Mark Pilgrim

Ponořme se do Python(u) 3

Python 3

Dive Into Python 3

Edice CZ.NIC

Mark Pilgrim
PONOŘME SE DO PYTHON(U) 3

Vydavatel:
CZ.NIC, z. s. p. o.
Americká 23, 120 00 Praha 2
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2011
Kniha vyšla jako 3. publikace v Edici CZ.NIC.
ISBN 978-80-904248-2-1

© 2010 Mark Pilgrim
Uvedené dílo podléhá licenci Creative Commons Uveďte autora-Zachovejte licenci 3.0 Unported.

CZ.NIC | SPRÁVCE DOMÉNY CZ

ISBN 978-80-904248-2-1

— Mark Pilgrim

Ponořme se do Python(u) 3
Dive Into Python 3

— Edice CZ.NIC

Předmluva a ediční poznámka

Vážení čtenáři,

po úspěchu naší předchozí publikace ProGit jsme se rozhodli, že třetí kniha v Edici CZ.NIC bude tak trochu na podobné téma a v podobném duchu. Opět jde o překlad velice kvalitní zahraniční publikace a také v tomto případě se dá očekávat, že jej ocení hlavně programátoři. Samozřejmě jsme i tentokrát sáhli po knize, která je pod volnou licencí a tedy filozofie její distribuce je blízká naší edici.

Podobně jako v případě nástroje Git je i Python technologie, která je mým kolegům velice dobře známa. Právě v programovacím jazyce Python je napsána podstatná část našeho centrálního registru pro správu domén, který se jmenuje FRED. Toto je jen jeden z mnoha důkazů, proč je nutné se tímto programovacím jazykem vážně zabývat.

Autor knihy **Mark Pilgrim** není ve světě Pythonu rozhodně žádným neznámým jménem. Své renomé si vybudoval již napsáním předchůdce této knihy s téměř stejným jménem. Právě úspěch dřívějšího díla je pro nás zárukou, že i tato verze si najde své čtenáře.

Ať už jste tedy v Pythonu nováčky nebo si jen chcete rozšířit své dosavadní znalosti, přeji Vám příjemnou četbu.

Ondřej Filip
Praha 17. listopadu 2010

Ediční poznámka autora

Ponořme se do Pythonu 3 pokrývá vlastnosti jazyka Python 3 a popisuje rozdíly proti jazyku Python 2. Ve srovnání s **Dive Into Python** zde naleznete asi 20 % revidovaného textu a asi 80 % nového materiálu. Knihu považuji za dokončenou, ale zpětná vazba je vždy vítána.

Obsah

— Přehled kapitol

-1. **Co najdete v „Ponořme se do Pythonu 3“ nového** — 17
0. **Instalujeme Python** — 21
1. **Váš první pythonovský program** — 45
2. **Přirozené datové typy** — 61
3. **Generátorová notace** — 91
4. **Řetězce** — 105
5. **Regulární výrazy** — 123
6. **Uzávěry a generátory** — 143
7. **Třídy a iterátory** — 159
8. **Iterátory pro pokročilé** — 173
9. **Unit Testing** — 193
10. **Refaktorizace** — 219
11. **Soubory** — 235
12. **XML** — 255
13. **Serializace pythonovských objektů** — 277
14. **Webové služby nad HTTP** — 297
15. **Případová studie: Přepis chardet pro Python 3** — 329
16. **Balení pythonovských knihoven** — 359
A. **Přepis kódu do Python 3 s využitím 2to3** — 377
B. **Jména speciálních metod** — 405
C. **Čím pokračovat** — 423
D. **Odstraňování problémů** — 427

-1. Co najdete v „Ponořme se do Pythonu 3“ nového — 17
-1.1. aneb „záporná úroveň” — **19**

0. Instalujeme Python — 21
0.1. Ponořme se — **23**
0.2. Který Python je pro vás ten správný? — **23**
0.3. Instalace pod Microsoft Windows — **24**
0.4. Instalace pod Mac OS X — **29**
0.5. Instalace pod Ubuntu Linux — **36**
0.6. Instalace na jiných platformách — **40**
0.7. Použití Python Shell — **41**
0.8. Editory a vývojová prostředí pro Python — **43**

1. Váš první pythonovský program — 45
1.1. Ponořme se — **47**
1.2. Deklarace funkcí — **48**
1.2.1. Nepovinné a pojmenované argumenty — **49**
1.3. Psaní čitelného kódu — **51**
1.3.1. Dokumentační řetězce — **51**
1.4. Vyhledávací cesta pro `import` — **52**
1.5. Všechno je objekt — **53**
1.5.1. Co to vlastně je objekt? — **54**
1.6. Odsazování kódu — **54**
1.7. Výjimky — **55**
1.7.1. Obsluha chyb importu — **57**
1.8. Volné proměnné — **58**
1.9. Vše je citlivé na velikost písmen — **58**
1.10. Spouštění skriptů — **59**
1.11. Přečtěte si — **60**

2. Přirozené datové typy — 61
2.1. Ponořme se — **63**
2.2. Booleovský typ — **63**
2.3. Čísla — **64**
2.3.1. Vynucení převodu celých čísel na reálná a naopak — **65**
2.3.2. Běžné operace s čísly — **66**
2.3.3. Zlomky — **67**
2.3.4. Trigonometrie — **67**
2.3.5. Čísla v booleovském kontextu — **68**
2.4. Seznamy — **69**
2.4.1. Vytvoření seznamu — **69**
2.4.2. Vytváření podseznamů — **70**
2.4.3. Přidávání položek do seznamu — **71**
2.4.4. Vyhledávání hodnoty v seznamu — **73**
2.4.5 Odstraňování položek ze seznamu — **74**
2.4.6. Odstraňování položek ze seznamu: Bonusové kolo — **75**
2.4.7. Seznamy v booleovském kontextu — **75**
2.5. N-tice — **76**
2.5.1. N-tice v booleovském kontextu — **78**
2.5.2. Přiřazení více hodnot najednou — **78**
2.6. Množiny — **79**
2.6.1. Vytvoření množiny — **79**
2.6.2. Úprava množiny — **81**
2.6.3. Odstraňování položek z množiny — **82**
2.6.4. Běžné množinové operace — **83**
2.6.5. Množiny v booleovském kontextu — **85**
2.7. Slovníky — **86**
2.7.1. Vytvoření slovníku — **86**
2.7.2. Úprava slovníku — **87**
2.7.3. Slovníky se smíšeným obsahem — **87**
2.7.4. Slovníky v booleovském kontextu — **88**
2.8. `None` — **89**
2.8.1. None v booleovském kontextu — **90**
2.9. Přečtěte si — **90**

3. Generátorová notace — 91
3.1. Ponořme se — **93**
3.2. Práce se soubory a s adresáři — **93**
3.2.1. Aktuální pracovní adresář — **93**
3.2.2. Práce se jmény souborů a adresářů — **94**
3.2.3. Výpis adresářů — **96**
3.2.4. Získání dalších informací o souboru — **97**
3.2.5. Jak vytvořit absolutní cesty — **98**
3.3. Generátorová notace seznamu — **98**
3.4. Generátorová notace slovníku — **100**
3.4.1. Další legrácky s generátorovou notací slovníků — **102**
3.5. Generátorová notace množin — **103**
3.6. Přečtěte si — **103**

4. Řetězce — 105
4.1. Pár nudných věcí, kterým musíme rozumět dříve, než se budeme moci ponořit — **107**
4.2. Unicode — **109**
4.3. Ponořme se — **111**
4.4. Formátovací řetězce — **111**
4.4.1. Složená jména oblastí — **113**
4.4.2. Specifikátory formátu — **114**
4.5. Další běžné metody řetězců — **115**
4.5.1. Vykrajování podřetězců — **117**
4.6. Řetězce vs. bajty — **117**
4.7. Závěrečná poznámka: Kódování znaků v pythonovském zdrojovém textu — **120**
4.8. Přečtěte si — **121**

5. Regulární výrazy — 123
5.1. Ponořme se — **125**
5.2. Případová studie: Adresa ulice — **125**
5.3. Případová studie: Římská čísla — **128**
5.3.1. Kontrola tisícovek — **128**
5.3.2. Kontrola stovek — **129**
5.4. Využití syntaxe `{n,m}` — **131**
5.4.1. Kontrola desítek a jednotek — **132**
5.5. Víceslovné regulární výrazy — **134**
5.6. Případová studie: Analýza telefonních čísel — **136**
5.7. Shrnutí — **141**

6. Uzávěry a generátory — 143
6.1. Ponořme se — **145**
6.2. Já vím jak na to! Použijeme regulární výrazy! — **146**
6.3. Seznam funkcí — **148**
6.4. Seznam vzorků — **150**
6.5. Soubor vzorků — **152**
6.6. Generátory — **154**
6.6.1. Generátor Fibonacciho posloupnosti — **155**
6.6.2. Generátor pravidel pro množné číslo — **156**
6.7. Přečtěte si — **158**

7. Třídy a iterátory — 159
7.1. Ponořme se — **161**
7.2. Definice tříd — **161**
7.2.1. Metoda `__init__()` — **162**
7.3. Vytváření instancí tříd — **163**
7.4. Členské proměnné — **163**
7.5. Fibonacciho iterátor — **164**
7.6. Iterátor pro pravidla množného čísla — **166**
7.7. Přečtěte si — **172**

8. Iterátory pro pokročilé — 173
8.1. Ponořme se — **175**
8.2. Nalezení všech výskytů vzorku — **176**
8.3. Nalezení jedinečných prvků posloupnosti — **177**
8.4. Činíme předpoklady — **178**
8.5. Generátorové výrazy — **179**
8.6. Výpočet permutací (pro lenochy) — **180**
8.7. Další legrácky v modulu itertools — **182**
8.8. Nový způsob úpravy řetězce — **185**
8.9. Vyhodnocování libovolných řetězců zachycujících pythonovské výrazy — **187**
8.10. Spojme to všechno dohromady — **190**
8.11. Přečtěte si — **191**

9. Unit Testing — 193
9.1. (Ne)ponořme se — **195**
9.2. Jediná otázka — **196**
9.3. „Zastav a začni hořet“ — **202**
9.4. Více zastávek, více ohně — **206**
9.5. A ještě jedna věc... — **209**
9.6. Symetrie, která potěší — **211**
9.7. Více špatných vstupů — **215**

10. Refaktorizace — 219
10.1. Ponořme se — **221**
10.2. Zvládání měnících se požadavků — **223**
10.3. Refaktorizace — **228**
10.4. Shrnutí — **232**

11. Soubory — 235
11.1. Ponořme se — **237**
11.2. Čtení z textových souborů — **237**
11.2.1. Kódování znaků vystrkuje svou ošklivou hlavu — **237**
11.2.2. Objekty typu stream — **238**
11.2.3. Čtení dat z textového souboru — **239**
11.2.4. Zavírání souborů — **241**
11.2.5. Automatické zavírání souborů — **242**
11.2.6. Čtení dat po řádcích — **243**
11.3. Zápis do textových souborů — **245**
11.3.1. A znovu kódování znaků — **246**
11.4. Binární soubory — **246**
11.5. Objekty typu stream z nesouborových zdrojů — **247**
11.5.1. Práce s komprimovanými soubory — **249**
11.6. Standardní vstup, výstup a chybový výstup — **250**
11.6.1. Přesměrování standardního výstupu — **251**
11.7. Přečtěte si — **254**

12. XML — 255
12.1. Ponořme se — **257**
12.2. Pětiminutový rychlokurz XML — **258**
12.3. Struktura Atom Feed — **261**
12.4. Analýza XML — **263**
12.4.1. Elementy jsou reprezentovány seznamy — **264**
12.4.2. Atributy jsou reprezentovány slovníky — **264**
12.5. Vyhledávání uzlů v XML dokumentu — **265**
12.6. lxml jde ještě dál — **268**
12.7. Generování XML — **270**
12.8. Analýza porušeného XML — **273**
12.9. Přečtěte si — **275**

13. Serializace pythonovských objektů — 277
13.1. Ponořme se — **279**
13.1.1. Stručná poznámka k příkladům v této kapitole — **279**
13.2. Uložení dat do „pickle souboru“ — **280**
13.3. Načítání dat z „pickle souboru“ — **281**
13.4. „Piklení“ bez souboru — **283**
13.5. Bajty a řetězce znovu zvedají své ošklivé hlavy — **284**
13.6. Ladění „pickle souborů“ — **284**
13.7. Serializace pythonovských objektů pro čtení z jiných jazyků — **286**
13.8. Uložení dat do `JSON` souboru — **287**
13.9. Zobrazení pythonovských datových typů do `JSON`— **289**
13.10. Serializace datových typů, které `JSON` nepodporuje — **289**
13.11. Načítání dat z `JSON` souboru — **293**
13.12. Přečtěte si — **295**

14. Webové služby nad HTTP — 297
14.1. Ponořme se — **299**
14.2. Vlastnosti HTTP — **300**
14.2.1. Používání mezipaměti — **300**
14.2.2. Kontrola Last-Modified — **301**
14.2.3. Kontrola ETag — **303**
14.2.4. Komprese — **304**
14.2.5. Přesměrování — **304**
14.3. Jak se nedostat k datům přes HTTP — **305**
14.4. Co že to máme na drátě? — **306**
14.5. Představujeme `httplib2` — **309**
14.5.1. Krátká odbočka vysvětlující, proč `httplib2` vrací bajty místo řetězců — **311**
14.5.2. Jak `httplib2` zachází s mezipamětí — **312**
14.5.3. Jak `httplib2` zachází s hlavičkami Last-Modified a ETag — **315**
14.5.4. Jak `http2lib` pracuje s kompresí — **318**
14.5.5. Jak `httplib2` řeší přesměrování — **318**
14.6. Za hranicemi HTTP GET — **322**
14.7. Za hranicemi HTTP POST — **326**
14.8. Přečtěte si — **328**

15. Případová studie: Přepis chardet pro Python 3 — 329
15.1. Ponořme se — **331**
15.2. Co se rozumí autodetekcí znakového kódování? — **331**

15.2.1. Není to náhodou neproveditelné? — **331**
15.2.2. Existuje takový algoritmus? — **332**
15.3. Úvod do modulu `chardet` — **332**
15.3.1. `UTF-N` s `BOM` — **332**
15.3.2. Kódování escape sekvencemi — **333**
15.3.3. Vícebajtová kódování — **333**
15.3.4. Jednobajtová kódování — **334**
15.3.5. `windows-1252` — **334**
15.4. Spouštíme `2to3` — **335**
15.5. Krátká odbočka k vícesouborovým modulům — **338**
15.6. Opravme, co `2to3` neumí — **340**
15.6.1. False je syntaktická chyba — **340**
15.6.2. Nenalezen modul `constants` — **341**
15.6.3. Jméno `'file'` není definováno — **342**
15.6.4. Řetězcový vzorek nelze použít pro bajtové objekty — **343**
15.6.5. Objekt typu `'bytes'` nelze implicitně převést na `str` — **345**
15.6.6. Nepodporované typy operandů pro `+`: `'int'` a `'bytes'` — **348**
15.6.7. Funkce `ord()` očekávala řetězec o délce 1, ale byl nalezen `int` — **350**
15.6.8. Neuspořádatelné datové typy: `int() >= str()` — **352**
15.6.9. Globální jméno `'reduce'` není definováno — **355**
15.7. Shrnutí — **357**

16. Balení pythonovských knihoven — 359
16.1. Ponořme se — **361**
16.2. Věci, které za nás Distutils neudělají — **362**
16.3. Struktura adresáře — **363**
16.4. Píšeme svůj instalační skript — **364**
16.5. Přidáváme klasifikaci našeho balíčku — **366**
16.5.1. Příklady dobrých klasifikátorů balíčků — **367**
16.6. Určení dalších souborů prostřednictvím manifestu — **368**
16.7. Kontrola chyb v našem instalačním skriptu — **369**
16.8. Vytvoření distribuce obsahující zdrojové texty — **369**
16.9. Vytvoření grafického instalačního programu — **371**
16.9.1. Tvorba instalačních balíčků pro jiné operační systémy — **373**
16.10. Přidání našeho softwaru do Python Package Index — **373**
16.11. Více možných budoucností balení pythonovských produktů — **375**
16.12. Přečtěte si — **375**

A. Přepis kódu do Pythonu 3 s využitím 2to3 — 377
A.1. Ponořme se — **379**
A.2. Příkaz `print` — **379**
A.3. Literály Unicode řetězců — **380**
A.4. Globální funkce `unicode()` — **380**
A.5. Datový typ `long` — **380**
A.6. Porovnání `<>` — **381**
A.7. Slovníková metoda `has_key()` — **381**
A.8. Slovníkové metody, které vracejí seznamy — **382**
A.9. Moduly, které byly přejmenovány nebo reorganizovány — **383**
A.9.1. `http` — **383**
A.9.2. `urllib` — **384**
A.9.3. `dbm` — **385**
A.9.4. `xmlrpc` — **385**
A.9.5. Ostatní moduly — **386**
A.10. Relativní importy uvnitř balíčku — **387**
A.11. Metoda iterátoru `next()` — **388**
A.12. Globální funkce `filter()` — **388**
A.13. Globální funkce `map()` — **389**
A.14. Globální funkce `reduce()` — **390**
A.15. Globální funkce `apply()` — **390**
A.16. Globální funkce `intern()` — **390**
A.17. Příkaz `exec` — **391**
A.18. Příkaz `execfile` — **391**
A.19. `repr`-literály (zpětné apostrofy) — **392**
A.20. Příkaz `try...except` — **392**

A.21. Příkaz `raise` — **393**
A.22. Metoda generátorů `throw` — **393**
A.23. Globální funkce `xrange()` — **394**
A.24. Globální funkce `raw_input()` a `input()` — **395**
A.25. Atributy funkcí `func_*` — **395**
A.26. Metoda `xreadlines()` V/V objektů — **396**
A.27. `lambda` funkce, které akceptují n-tici místo více parametrů — **396**
A.28. Atributy speciálních metod — **397**
A.29. Speciální metoda `__nonzero__` — **397**
A.30. Oktalové literály — **398**
A.31. `sys.maxint` — **398**
A.32. Globální funkce `callable()` — **399**
A.33. Globální funkce `zip()` — **399**
A.34. Výjimka `StandardError` — **399**
A.35. Konstanty modulu `types` — **400**
A.36. Globální funkce `isinstance()` — **400**
A.37. Datový typ `basestring` — **401**
A.38. `itertools` module — **401**
A.39. `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback` — **401**
A.40. Generátory seznamů nad n-ticemi — **402**
A.41. Funkce `os.getcwdu()` — **402**
A.42. Metatřídy — **402**
A.43. Věci týkající se stylu — **403**
A.43.1. Množinové literály (`set()`; explicitně) — **403**
A.43.2. Globální funkce `buffer()` (explicitně) — **403**
A.43.3. Bílé znaky kolem čárek (explicitně) — **404**
A.43.4. Běžné obraty (explicitně) — **404**

B. Jména speciálních metod — 405
B.1. Ponořme se — **407**
B.2. Základy — **407**
B.3. Třídy, které se chovají jako iterátory — **407**
B.4. Vypočítávané atributy — **408**
B.5. Třídy, které se chovají jako funkce — **411**
B.6. Třídy, které se chovají jako množiny — **412**
B.7. Třídy, které se chovají jako slovníky — **413**
B.8. Třídy, které se chovají jako čísla — **414**
B.9. Třídy, které se dají porovnávat — **417**
B.10. Třídy, které podporují serializaci — **418**
B.11. Třídy, které mohou být použity v bloku `with` — **418**
B.12. Opravdu esoterické věci — **420**
B.13. Přečtěte si — **420**

C. Čím pokračovat — 423
C.1. Doporučuji k přečtení — **425**
C.2. Kde hledat kód kompatibilní s Pythonem 3 — **426**

D. Odstraňování problémů — 427
D.1. Ponořme se — **429**
D.2. Jak se dostat k příkazovému řádku — **429**
D.3. Spuštění Pythonu z příkazového řádku — **429**

“Isn’t this where we came in?”

— Pink Floyd, *The Wall*

-1. Co najdete v „Ponořme se do Pythonu 3“ nového

-1. Co najdete v „Ponořme se do Pythonu 3“ nového — 17
-1.1. aneb „záporná úroveň” — **19**

-1.1. aneb „záporná úroveň”

Už jste v jazyce Python programovali? Četli jste originální publikaci „Dive Into Python“? Koupili jste si ji v knižní podobě? (Pokud ano, díky!) Jste připraveni ponořit se do jazyka Python 3?... Pokud tomu tak je, čtěte dál. (Pokud nic z toho neplatí, měli byste raději začít od začátku.)

Kap. Python 3 se dodává se skriptem nazvaným `2to3`. Naučte se jej. Milujte jej. Používejte jej. Přepis kódu do Pythonu 3 s využitím `2to3` je referenční příručkou ke všem věcem, které skript `2to3` umí opravit automaticky. A protože řada těchto věcí souvisí se změnami syntaxe, je tato příručka dobrým výchozím bodem ke studiu syntaktických změn, které Python 3 přináší. (Z příkazu `print` se stala funkce, obrat `` `x` `` přestal fungovat atd.)

Kap. Případová studie: Přepis `chardet` pro Python 3 popisuje mé (nakonec úspěšné) úsilí o přepis netriviální knihovny z Pythonu 2 do Pythonu 3. Možná vám tato studie pomůže, možná ne. Učící křivka je zde poměrně strmá, protože nejdříve musíte porozumět knihovně samotné. Teprve potom můžete rozumět tomu, proč přestala fungovat a jakým způsobem jsem ji opravil. Řada problémů se váže na řetězce. Když už o nich mluvíme...

Řetězce. Uffff. Kde mám začít? Python 2 používal „řetězce“ a „řetězce v Unicode“. Python 3 rozlišuje „bajty“ a „řetězce“. Všechny řetězce se nyní stávají řetězci v Unicode. Pokud s obsahem chceme zacházet jako s bajty, musíme použít nový datový typ nazvaný `bytes`. Python 3 *nikdy* skrytě nepřevádí řetězce na bajty a naopak. Takže pokud si v každém momentě nejste jistí, zda používáte ten či onen typ, kód
Kap. vašeho programu téměř jistě přestane fungovat. Další podrobnosti naleznete v kapitole Řetězce.

Problém bajty versus řetězce se v textu této knihy vynořuje znovu a znovu.

Kap. • V kapitole Soubory se seznámíte s rozdílem mezi čtením souborů v „binárním“ a „textovém“ režimu. Při čtení (ale také při zápisu) souborů v textovém režimu se vyžaduje zadání parametru určujícího kódování (`encoding`). Některé metody textových souborů počítají znaky, ale jiné metody zase počítají bajty. Pokud ve svém zdrojovém kódu předpokládáte, že se jeden znak rovná jednomu bajtu, pak to při přechodu na vícebajtové znaky *přestane fungovat*.

Kap. • V kapitole Webové služby nad `HTTP` čte modul `httplib2` hlavičky a data prostřednictvím protokolu `HTTP`. Hlavičky se vracejí v podobě řetězců, ale těla se vracejí jako bajty.

Kap. • V kapitole Serializace pythonovských objektů se naučíte, proč modul `pickle` pro Python 3 definuje nový datový formát, který je zpětně nekompatibilní s verzí pro Python 2. (Nápověda: Důvodem jsou bajty a řetězce.) Python 3 podporuje také serializaci objektů do a z `json`, který dokonce nepracuje s typem `bytes`. Ukážeme si, jak se to dá obejít.

Kap. • V části Případová studie: Přepis `chardet` pro Python 3 se setkáte se zatraceným zmatkem mezi bajty a řetězci úplně všude.

Dokonce i kdyby vás Unicode nechával úplně chladné (ale ne, nenechá), budete si určitě chtít něco přečíst o formátování řetězců v jazyce Python 3. Zcela se liší od předpisu formátování řetězců v jazyce Python 2.

S iterátory se v Pythonu 3 setkáte všude. A teď už jim rozumím mnohem víc, než tomu bylo před pěti lety, kdy jsem napsal „Dive Into Python“. Snažte se jim porozumět také, protože mnoho funkcí, které v jazyce Python 2 vracely seznamy, vrací v Pythonu 3 právě iterátory. Přinejmenším byste si měli přečíst druhou polovinu kapitoly Iterátory a druhou polovinu kapitoly Iterátory pro pokročilé.

Kap. Na přání čtenářů jsem přidal přílohu Jména speciálních metod, která se podobá kapitole Data Model (Datový model) uvedené v dokumentaci jazyka Python.

V době, kdy jsem psal „Dive Into Python“, měly všechny dostupné knihovny pro práci s XML mizernou kvalitu. Pak ale Fredrik Lundh napsal modul **ElementTree**, který není *ale vůbec mizerný*. Pythonovští bohové moudře začlenili ElementTree do standardní knihovny, a tak se tento modul stal základem mé nové kapitoly o XML. Starší způsoby zpracování XML jsou stále podporované, ale měli byste se jim vyhnout, protože jsou zkrátka mizerné!

V Pythonu je nové také to — ne v jazyce, ale v komunitě uživatelů —, že se objevila úložiště kódu, jako je **Python Package Index (PyPI)**. Python se dodává s utilitami k zabalení vašeho kódu do standardního formátu a tyto balíčky pak mohou být zveřejněny na PyPI. O podrobnostech se dočtete v kapitole Kap. Balení pythonovských knihoven.

“Tempora mutantur nos et mutamur in illis. ”
(Časy se mění a my se měníme s nimi.)

— přísloví ze starého Říma

0. Instalujeme Python

0. Instalujeme Python — 21
0.1. Ponořme se — **23**
0.2. Který Python je pro vás ten správný? — **23**
0.3. Instalace pod Microsoft Windows — **24**
0.4. Instalace pod Mac OS X — **29**
0.5. Instalace pod Ubuntu Linux — **36**
0.6. Instalace na jiných platformách — **40**
0.7. Použití Python Shell — **41**
0.8. Editory a vývojová prostředí pro Python — **43**

0.1. Ponořme se
0.2. Který Python je pro vás ten správný?

0.1. Ponořme se

Než začneme programovat v jazyce Python 3, musíme si jej nainstalovat. Nebo ne?

0.2. Který Python je pro vás ten správný?

Pokud používáte účet na hostovaném serveru, mohl být Python 3 již nainstalován jeho správcem. Pokud provozujete Linux doma, můžete mít Python 3 již také k dispozici. Nejpopulárnější distribuce systému GNU/Linux obsahují v základní instalaci Python 2. Malá, ale zvětšující se skupina distribucí obsahuje také Python 3. Mac OS X se dodává s Pythonem 2 (verze spouštěná přes příkazový řádek), ale v době psaní této knihy neobsahoval Python 3. Microsoft Windows se nedodává s žádnou verzí Pythonu. Ale nepropadejte zoufalství! Nezávisle na tom, jaký operační systém používáte, můžete Python nainstalovat na několik kliknutí.

Nejjednodušší způsob ověření si, zda máte k dispozici Python 3 na svém systému Linux nebo Mac OS X, začíná tím, že se dostanete na příkazový řádek. Jakmile se nacházíte za vyzývacím řetězcem příkazového řádku, napište jednoduše `python3` (vše malými písmeny, bez mezer), stiskněte `ENTER` a uvidíte, co se stane. Na svém domácím systému Linux už mám Python 3.1 nainstalovaný. Uvedeným příkazem vstoupím do *pythonovského interaktivního shellu*.

```
mark@atlantis:~$ python3
Python 3.1 (r31:73572, Jul 28 2009, 06:52:23)
[GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(Až budete chtít pythonovský interaktivní shell opustit, napište `exit()` a stiskněte `ENTER`.)

Můj poskytovatel webového prostoru používá také Linux a umožňuje přístup přes příkazový řádek, ale Python 3 není na serveru nainstalován. (Bééé!)

```
mark@manganese:~$ python3
bash: python3: command not found
```

Takže zpět k otázce, kterou jsme tuto podkapitolu zahájili: „Který Python je pro vás ten správný?“ Ten, který poběží na počítači, který máte k dispozici.

> Následuje návod pro instalaci pod Windows, nebo přeskočte na Instalace pod Mac OS X, Instalace pod Ubuntu Linux nebo Instalace na jiných platformách.

0.3. Instalace pod Microsoft Windows

V dnešní době se Windows dodávají ve dvou architekturách: 32bitové a 64bitové. Máme tu samozřejmě řadu různých verzí Windows — XP, Vista, Windows 7 —, ale Python běží na všech. Rozlišení mezi 32bitovou a 64bitovou architekturou je důležitější. Pokud nemáte vůbec tušení, jakou architekturu používáte, pak je to pravděpodobně 32bitová.

Přejděte na stránku `python.org/download/` a stáhněte si windowsovský instalátor Python 3, který se hodí pro vaši architekturu. Možnosti vaší volby budou vypadat nějak takto:

- **Python 3.1 Windows installer** (Windows binary — does not include source)
- **Python 3.1 Windows AMD64 installer** (Windows AMD64 binary — does not include source)

Nechci zde uvádět konkrétní odkazy, protože Python neustále prochází drobnými úpravami a nechci být zodpovědný za to, že jste nějakou důležitou úpravu prošvihli. Vždy byste měli nainstalovat co nejnovější verzi Pythonu 3.x, tedy pokud nemáte nějaké esoterické důvody k tomu, abyste tak neučinili.

Jakmile se stahování dokončí, poklepejte na soubor s příponou `.msi`. Protože se snažíte o spuštění programu, zobrazí Windows bezpečnostní varování. Oficiální instalátor Pythonu je digitálně podepsán jménem organizace *Python Software Foundation*, která dohlíží na vývoj jazyka Python. Nepřijímejte imitace!

Instalaci Pythonu 3 zahájíme stisknutím tlačítka `Run`.

Nejdříve se vás instalátor zeptá, zda chcete Python 3 nainstalovat pro všechny uživatele, nebo jen pro sebe. Volba „instalovat pro všechny uživatele“ je přednastavena. Pokud nemáte nějaký dobrý důvod pro jinou volbu, pak toto je ta nejlepší. (Jeden možný důvod, proč byste mohli chtít „instalovat jen pro mne“, je ten, že si chcete nainstalovat Python na počítači v práci a váš účet ve Windows nemá oprávnění administrátora. Ale proč byste v takovém případě chtěli instalovat Python bez svolení svého správce Windows? Ne abyste mě dostali do potíží!)

Svoji volbu způsobu instalace potvrdíte stiskem tlačítka `Next`.

Instalátor vás poté vyzve k výběru instalačního adresáře. Pro všechny verze Python 3.1.x je přednastavena hodnota `C:\Python31\`, která by měla vyhovovat většině uživatelů. Pokud ovšem nemáte zvláštní důvod cestu změnit. Pokud instalujete všechny aplikace na disk označený jiným písmenem, můžete příslušnou cestu vybrat příslušnými ovládacími prvky. Nebo prostě cestu k adresáři napíšete do spodního pole. Python nemusíte instalovat jen na disk `C:`. Můžete si jej nainstalovat na libovolný disk a do libovolného adresáře.

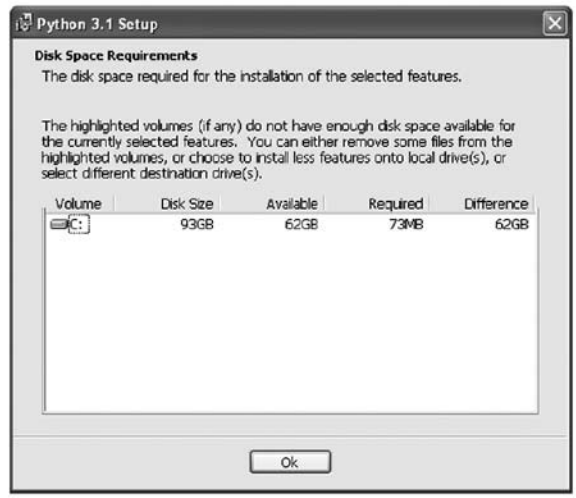
Volbu cílového adresáře potvrdíte stiskem tlačítka `Next`.

Další dialogová stránka vypadá komplikovaně, ale ve skutečnosti není. V případě Pythonu 3 máte možnost neinstalovat úplně všechny jeho komponenty — podobně jako u jiných instalačních programů. Pokud máte obzvlášť málo místa na disku, můžete některé komponenty vynechat.

- Volba **Register Extensions** (asociovat přípony) vám zajistí možnost spouštět pythonovské skripty (soubory s příponou `.py`) poklepáním na jejich ikonu. Je to sice doporučeno, ale není to nezbytné. (Tato volba nevyžaduje žádný diskový prostor, takže její potlačení není výhodné.)
- **Tcl/Tk** je grafická knihovna, kterou využívá pythonovský shell. Ten budeme používat v celé knize. Velmi doporučuji, abyste tuto volbu ponechali zapnutou.
- Volba **Documentation** vede k instalaci souborů s nápovědou, která obsahuje mnohé z informací uvedených na `docs.python.org`. Pokud máte omezený přístup k internetu nebo pokud používáte vytáčené připojení, doporučuji volbu ponechat zapnutou.
- Volba **Utility Scripts** v sobě zahrnuje i instalaci skriptu `2to3.py`, o kterém se budeme učit v této knize později. Pokud se chcete naučit přepisování existujícího kódu napsaného pro Python 2 do podoby pro Python 3, pak se zapnutí této volby vyžaduje. Pokud nemáte žádné programy napsané pro Python 2, můžete tuto volbu vypnout.
- Volba **Test Suite** zajistí instalaci sady skriptů, které se používají pro testování funkčnosti interpretu jazyka Python. V této knize je nebudeme používat. A nepoužíval jsem je nikdy ani během výuky programování v Pythonu. Volba je zcela na vás.

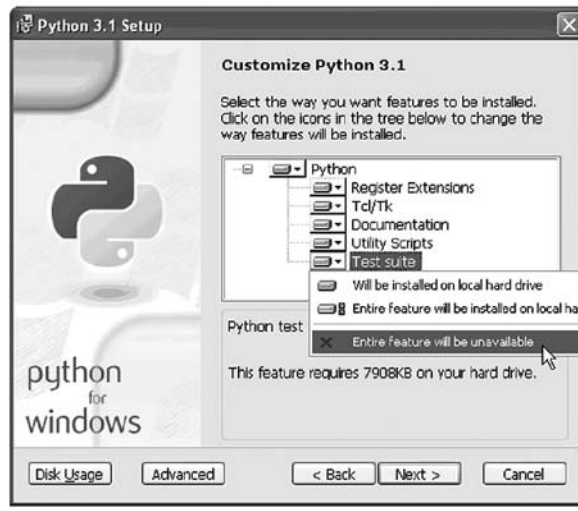
Pokud si nejste jisti, kolik máte místa na disku, klikněte na tlačítko `Disk Usage`. Instalátor zobrazí seznam písmen vašich disků, zjistí, kolik místa je na každém z nich, a vypočítá, kolik místa na nich zbude po instalaci.

Stiskem tlačítka `OK` se dostaneme na dialogovou stránku „Customizing Python“.



Pokud se rozhodnete volbu vynechat, stiskněte tlačítko pro rozbalení seznamu a vyberte „Entire feature will be unavailable“ (celá část bude nedostupná). Vynecháním Test Suite ušetříte na disku pěkných 7908 kb.

Výběr voleb potvrdíte stiskem tlačítka `Next`.

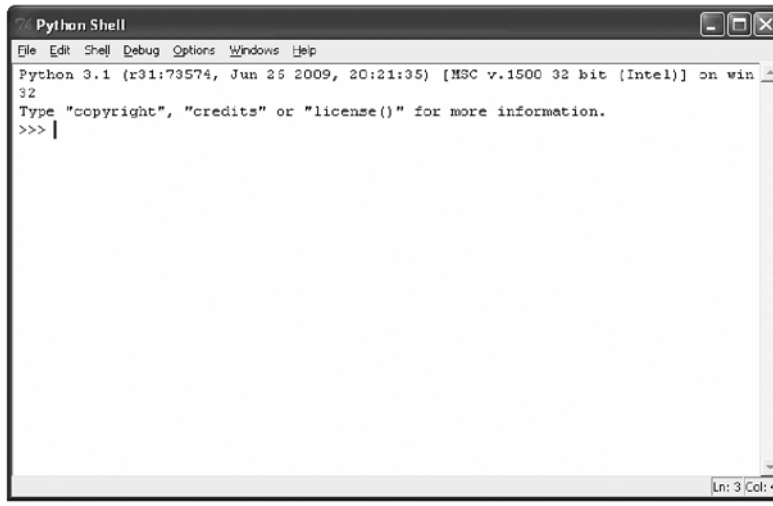


Instalátor nakopíruje všechny nezbytné soubory do vámi vybraného adresáře. (Proběhne to tak rychle, že jsem to musel zkusit třikrát, než se mi podařilo zachytit obrázek tohoto procesu.)

Stiskem tlačítka `Finish` ukončíme činnost instalátoru.

Ve vašem menu `Start` by se měla objevit položka s názvem `Python 3.1`. V ní se nachází program `IDLE`. Výběrem této položky spustíte interaktivní pythonovský shell. (Poznámka překladatele: Někdy ho autor označuje jako „grafický“ interaktivní shell. Jde o obdobu interaktivního pythonovského shellu, který se spouští v konzolovém okně. Tentokrát ale využívá prostředky grafického uživatelského

rozhraní (GUI) a v menu okna nalezneme i položky pro spuštění editoru nebo pro spuštění ladicího režimu. Dalo by se říct, že je to nástroj „téměř úplně, ale ne zcela naprosto nepodobný…“ klasickým IDE (integrované vývojové prostředí). Jenže to není soustředěné kolem editoru, ale spíš kolem shellu. Je to prostě IDLE. No zkrátka se na to podívejte a rozhodněte se sami, jak tomu budete říkat.)



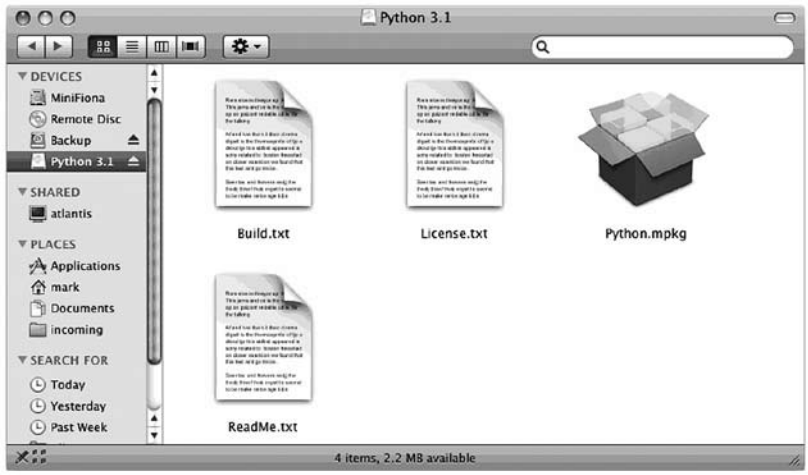
0.4. Instalace pod Mac OS X

Všechny moderní počítače Macintosh používají procesor firmy Intel (stejný jako většina osobních počítačů s Windows). Starší počítače Mac používají procesory PowerPC. Rozdílům rozumět nemusíte, protože existuje jen jeden jediný instalátor Pythonu pro všechny počítače Macintosh.

Přejděte na stránku `python.org/download/` a stáhněte si příslušný instalátor pro Mac. Bude u něj napsáno něco ve stylu **Python 3.1 Mac Installer Disk Image**, ačkoliv číslo verze se může lišit. Ujistěte se, že stahujete verzi 3.x a ne 2.x.

Váš prohlížeč by měl automaticky připojit obraz disku a otevřít okno Finder zobrazující jeho obsah. (Pokud se tak nestane, budete muset najít obraz disku ve svém adresáři pro stažené soubory a připojit jej poklepáním. Jmenuje se `python-3.1.dmg` nebo podobně.) Obraz disku obsahuje řadu textových souborů (`Build.txt, License.txt, ReadMe.txt`) a také skutečný instalační balík `Python.mpkg`.

Poklepejte na `Python.mpkg` a instalátor Mac Python se spustí.

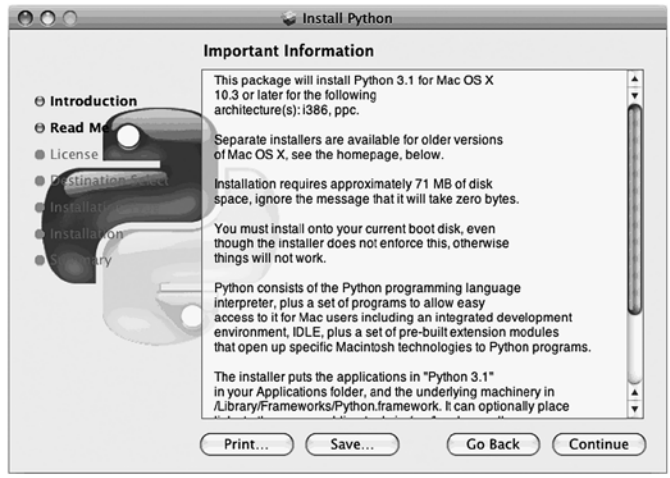


Na první stránce naleznete stručný popis jazyka Python a pro více detailů jste odkázáni na soubor `ReadMe.txt`. (...který jste nečetli. Nebo četli?)

Dál se posuneme stiskem tlačítka `Continue`.

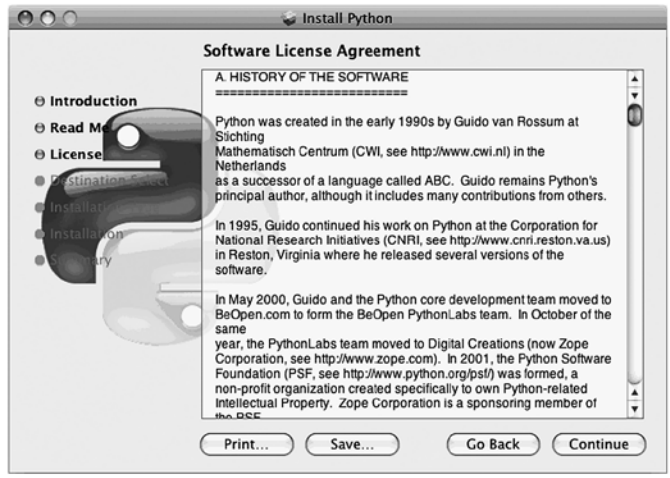
Následující stránka dialogu obsahuje některé důležité informace: Python vyžaduje Mac OS X 10.3 nebo novější. Pokud stále používáte Mac OS X 10.2, budete jej muset aktualizovat na vyšší verzi. Společnost Apple už pro váš operační systém neposkytuje bezpečnostní aktualizace a už při pouhém připojení na internet vystavujete svůj počítač riziku. A navíc nemůžete používat Python 3.

Pokračujeme stiskem tlačítka `Continue`.



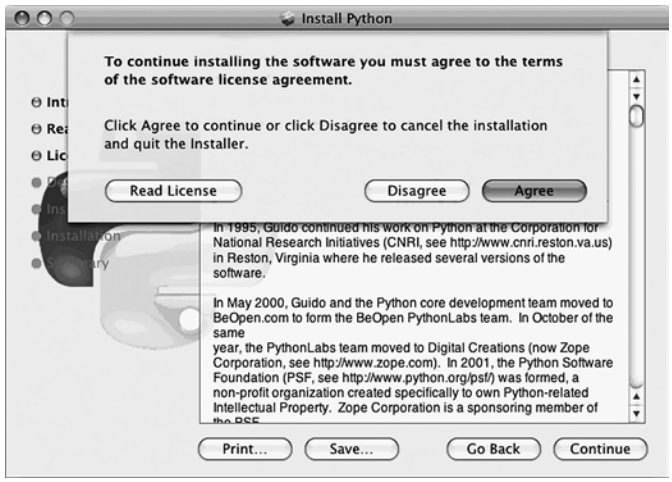
Tak jako všechny dobré instalátory, i ten pythonovský zobrazí licenční ujednání. Python je open source a jeho licence je schválena společností Open Source Initiative. Během historického vývoje měl Python řadu vlastníků a sponzorů. Každý z nich zanechal v jeho licenci svůj otisk. Ale konečný výsledek vypadá takto: Python je open source, můžete jej používat na libovolné platformě, pro libovolný účel, zdarma a bez závazku k protislužbě.

Stiskněte tlačítko `Continue` ještě jednou.



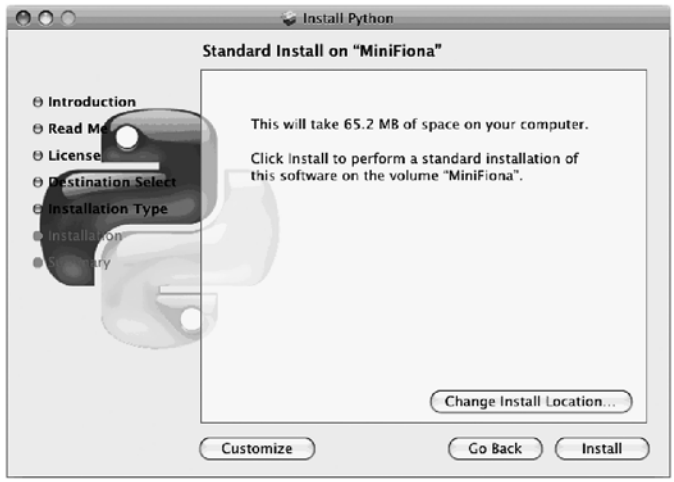
Abyste mohli instalaci dokončit, musíte kvůli manýrům v jádru applovského instalátoru projevit „souhlas“ se softwarovou licencí. Ale protože Python je open source, ve skutečnosti „souhlasíte“ s tím, že vám licence zaručuje práva navíc, než aby vás omezovala.

Pokračujeme stiskem tlačítka `Agree`.



Na další obrazovce můžete změnit umístění instalace. Python musíte instalovat na zaváděcí disk, ale kvůli omezením instalátoru to není vynuceno. Popravdě řečeno, nikdy jsem nepociťoval potřebu umístění instalace měnit.

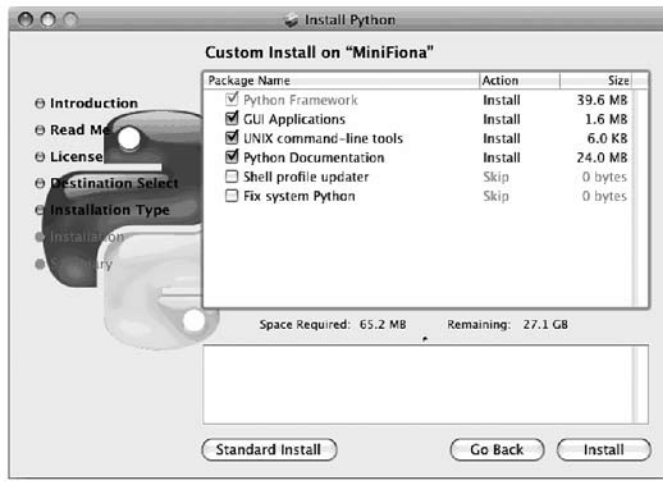
Na této obrazovce také můžete instalaci upravit vyloučením komponent, které nepotřebujete. Pokud tak chcete učinit, stiskněte tlačítko `Customize`. V opačném případě stiskněte tlačítko `Install`.



Pokud zvolíte uživatelskou úpravu instalace (`Custom Install`), nabídne vám instalátor následující seznam:

- **Python Framework.** Jde o jádro Pythonu. Proto je tato možnost předvolena a současně je zakázáno ji měnit. Tato část se nainstalovat musí.
- **GUI Applications** v sobě zahrnuje `IDLE`, což je grafický pythonovský shell. Budeme jej používat během celé knihy. Velmi doporučuji, abyste tuto volbu ponechali zapnutou.
- **UNIX command-line tools** v sobě obsahuje konzolovou aplikaci `python3`. Velmi doporučuji, abyste také tuto volbu ponechali zapnutou.
- **Python Documentation** obsahuje mnohé z informací uvedených na `docs.python.org`. Pokud máte omezený přístup k internetu nebo pokud používáte vytáčené připojení, doporučuji volbu ponechat zapnutou.
- **Shell profile updater** kontroluje, zda je nutné aktualizovat váš shellovský profil (použitý v `Terminal.app`) tak, aby bylo zajištěno, že umístění instalované verze Pythonu bude součástí prohledávaných cest. Tuto volbu pravděpodobně nebudete potřebovat měnit.
- Volbu **Fix system Python** byste měnit neměli. (Říká vašemu počítači, aby byl Python 3 použit jako preferovaný Python pro spouštění všech skriptů, včetně zabudovaných skriptů dodávaných firmou Apple. Dopadlo by to velmi špatně, protože většina těchto skriptů byla napsána pro Python 2 a pod verzí Python 3 by neběžely správně.)

Pokračujeme stiskem tlačítka `Install`.



Instalátor se vás zeptá na heslo správce, protože systémové binární soubory a nástroje se instalují do adresáře `/usr/local/bin/`. Bez administrátorských oprávnění Mac Python zkrátka nenainstalujete.

Stiskem tlačítka OK zahájíme instalaci.

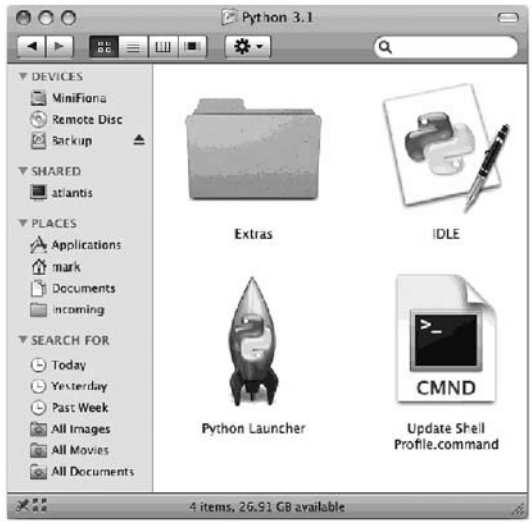
Během instalace částí, které jste si vybrali, instalátor indikuje postup instalace.

Pokud šlo všechno dobře, oznámí vám instalátor úspěšné dokončení instalace zobrazením zelené „fajfky“.

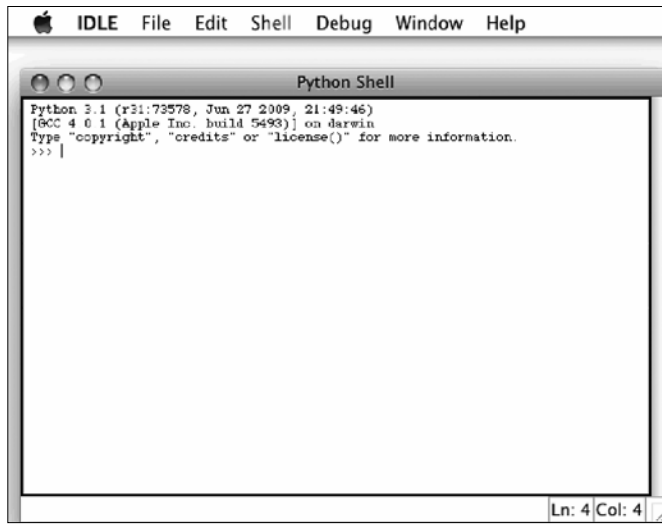
Stiskem tlačítka `Close` činnost instalátoru ukončíme.

Za předpokladu, že jste nezměnili umístění instalace, najdete nově nainstalované soubory v podadresáři `Python 3.1` uvnitř adresáře `/Applications`. Nejdůležitější součástí je zde grafický pythonovský shell zvaný idle.

Poklepejte na něj a pythonovský shell se spustí.

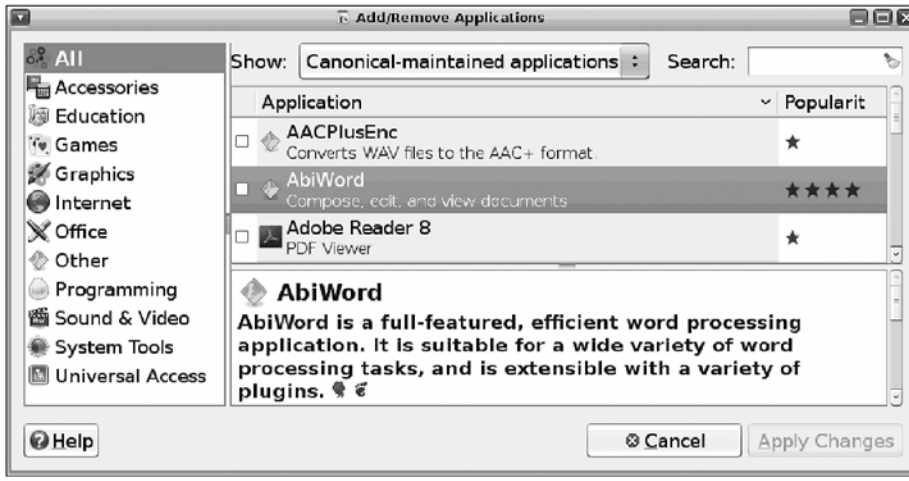


V pythonovském shellu strávíte při průzkumu jazyka Python nejvíce času. U příkladů budeme v této knize předpokládat, že se k pythonovskému shellu umíte dostat.



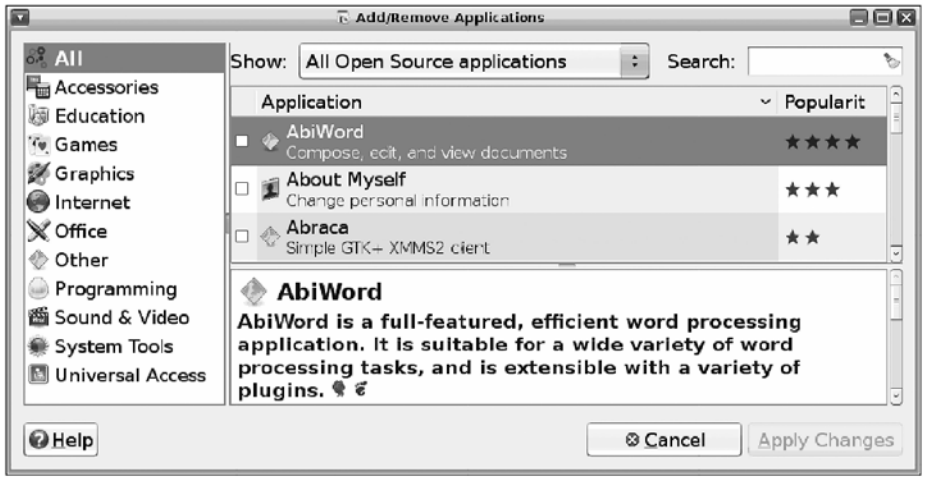
0.5. Instalace pod Ubuntu Linux

Moderní distribuce systému Linux jsou podepřeny ohromnými úložišti předkompilovaných aplikací, které jsou připraveny k okamžité instalaci. Detaily se pro konkrétní distribuce liší. Nejsnadnější způsob instalace Pythonu 3 pod Ubuntu Linux spočívá v použití nástroje `Add/Remove`, který najdete v menu `Applications`.



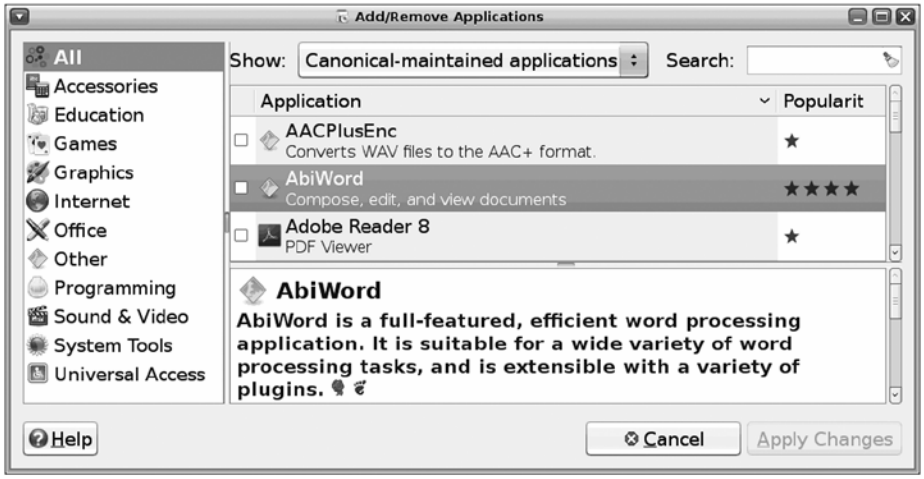
Když poprvé spustíte aplikaci `Add/Remove`, zobrazí vám seznam předvybraných aplikací v různých kategoriích. Některé z nich jsou již nainstalované, ale většina z nich ne. Protože úložiště obsahuje přes 10 tisíc aplikací, můžete pomocí různých filtrů omezit zobrazení jen na jeho malé části. Základem je filtr „Canonical-maintained applications“, což je malá podmnožina z celkového množství aplikací, které jsou oficiálně podporovány společností Canonical, která vytvořila a udržuje distribuci Ubuntu Linux.

Python 3 není společností Canonical udržován, takže jako první krok potlačíme činnost tohoto filtru a vybereme „All Open Source applications“ (všechny open source aplikace).



Jakmile změníte nastavení filtru tak, aby zahrnoval všechny open source aplikace, použijte k vyhledání `Pythonu 3` vyhledávací box nacházející se hned za nabídkou filtru.

V tom okamžiku se seznam aplikací zúží jen na ty, které souvisejí s Pythonem 3. Poté vybereme dva balíčky. Tím prvním je `Python (v3.0)`. Obsahuje vlastní interpret jazyka Python.



Druhý požadovaný balíček se nachází bezprostředně nad ním: `IDLE (using Python-3.0)`. Jde o grafický pythonovský shell, který budeme používat během celé knihy.

Po označení uvedených dvou balíčků pokračujte stiskem tlačítka `Apply Changes`.

Správce balíčků vás požádá o potvrzení, že chcete přidat jak `IDLE (using Python-3.0)`, tak `Python (v3.0)`. Pokračujeme stiskem tlačítka `Apply`.

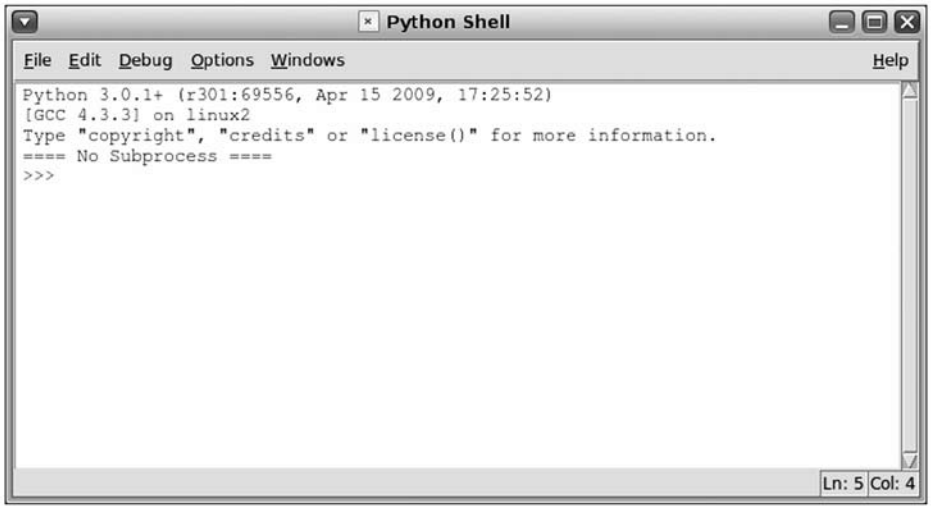
Během stahování potřebných balíčků z internetového úložiště společnosti Canonical zobrazuje správce balíčků indikátor postupu stahování.

Jakmile jsou balíčky staženy, zahájí správce balíčků automaticky jejich instalaci.

Pokud šlo všechno dobře, potvrdí správce balíčků, že byly oba úspěšně nainstalovány. V tomto okamžiku můžete poklepáním na `IDLE` spustit pythonovský shell, nebo můžete stiskem tlačítka `Close` ukončit činnost správce balíčků.

Pythonovský shell můžete spustit kdykoliv tím způsobem, že v menu `Applications` a v podmenu `Programming` vyberete `IDLE`.

V pythonovském shellu strávíte při průzkumu jazyka Python nejvíce času. U příkladů budeme v této knize předpokládat, že se k pythonovskému shellu umíte dostat.



0.6. Instalace na jiných platformách

Python 3 je dostupný pro řadu různých platforem. Abychom byli konkrétnější, je dostupný pro prakticky každou distribuci systému Linux, BSD a pro distribuce založené na systému Solaris. Takže například RedHat Linux používá správce balíčků yum. FreeBSD má svou sbírku ports and packages collection, SUSE má zypper a Solaris má pkgadd. Když zkusíte zběžně prohledat web při zadání `Python 3` + *váš operační systém*, dozvíte se, zda je balík s Pythonem 3 dostupný, a pokud ano, jak jej můžete nainstalovat.

0.7. Použití Python Shell

Python Shell (kvůli skloňování a zobecnění pohledu mu budeme říkat také *pythonovský shell*) bude nástrojem pro studium syntaxe jazyka Python, zdrojem interaktivní nápovědy k příkazům a prostředkem pro ladění krátkých programů. Grafický pythonovský shell (pojmenovaný `IDLE`) obsahuje navíc ucházející textový editor, který podporuje barevné zvýrazňování syntaxe a zajišťuje spolupráci s (konzolovým) pythonovským shellem. Pokud již nemáte nějaký svůj oblíbený textový editor, měli byste si `IDLE` vyzkoušet.

Ale proberme nejdříve hlavní věci. Samotný Python Shell je úžasné interaktivní prostředí, se kterým si vyhrajete. V celé knize se budete setkávat s příklady, jako je tento:

```
>>> 1 + 1
2
```

Tři úhlové závorky (`>>>`) jsou vyzývacím řetězcem pythonovského shellu. Tuto část neopisujte. Vyjadřuji tím to, že byste si příklad měli vyzkoušet v pythonovském shellu.

Vy budete psát pouze část `1 + 1`. V pythonovském shellu můžete napsat jakýkoliv platný pythonovský výraz nebo příkaz. Nestyďte se! Nekousne vás to! Přinejhorším se stane to, že se vám zobrazí chybové hlášení. Příkazy se provádějí okamžitě (jakmile stisknete `ENTER`). Také výrazy jsou vyhodnoceny okamžitě a pythonovský shell vytiskne jejich výsledek.

Takže zobrazená část `2` je výsledkem vyhodnocení předchozího výrazu. Protože se tak stalo, je `1 + 1` zjevně platným pythonovským výrazem. Jeho výsledek je samozřejmě `2`.

Vyzkoušejme něco dalšího.

```
>>> print('Hello world!')
Hello world!
```

Docela jednoduché, že? Ale v pythonovském shellu toho můžete dělat mnohem víc. Když se někdy zadrhnete — když si nemůžete vzpomenout na nějaký příkaz nebo si nemůžete vzpomenout na správné argumenty předávané nějaké funkci —, můžete se v pythonovském shellu dostat k interaktivní nápovědě. Napište prostě `help` a stiskněte `ENTER`.

```
>>> help
Type help() for interactive help, or help(object) for help about object.
```

Nápovědu můžeme používat ve dvou režimech. Můžeme získat nápovědu pro jeden objekt. Vytiskne se prostě jeho dokumentace a vrátíte se na vyzývací řádek pythonovského shellu. Nebo můžeme vstoupit do *režimu nápovědy*, ve kterém místo vyhodnocování pythonovských výrazů píšeme klíčová slova nebo jména příkazů a Python zobrazuje vše, co o těchto příkazech ví.

Pro vstup do interaktivního režimu nápovědy napište `help()` a stiskněte ENTER.

```
>>> help()
Welcome to Python 3.0!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

Všimněte si, že se vyzývací řetězec změnil z `>>>` na `help>`. Má vám to připomenout, že se nacházíte v interaktivním režimu nápovědy. V tomto okamžiku můžete napsat libovolné klíčové slovo, příkaz, jméno modulu, jméno funkce — v podstatě cokoliv, čemu Python rozumí — a přečtete si k tomu zobrazenou dokumentaci.

```
help> print                                                              [1]
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.

help> PapayaWhip                                                         [2]
no Python documentation found for 'PapayaWhip'

help> quit                                                               [3]

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
```

```
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>                                                                    [4]
```

[1] Abyste dostali dokumentaci k funkci `print()`, napište print a stiskněte ENTER. V interaktivním režimu nápovědy se zobrazí něco podobného jako manovská stránka: jméno funkce, stručný popis, argumenty funkce a jejich přednastavené hodnoty a tak dále. Pokud se vám zdá obsah dokumentace nejasný, nepropadejte panice. V následujících několika kapitolách se o těchto věcech dozvíte více.

[2] V interaktivním režimu nápovědy se samozřejmě nedozvíte všechno. Pokud zde napíšete něco, co není pythonovským příkazem, modulem, funkcí nebo nějakým zabudovaným klíčovým slovem, režim interaktivní nápovědy prostě pokrčí svými virtuálními rameny.

[3] Interaktivní režim nápovědy ukončíte tím, že napíšete `quit` a stisknete ENTER.

[4] Vyzývací řádek se změní zpět na `>>>`, čímž se dozvíte, že jste opustili režim interaktivní nápovědy a vrátili jste se do pythonovského shellu.

Grafický pythonovský shell IDLE navíc obsahuje textový editor šitý na míru jazyku Python.

0.8. Editory a vývojová prostředí pro Python

Pokud jde o psaní programů v jazyce Python, nepředstavuje idle jedinou možnost. Jakkoliv může být užitečný při seznamování se s jazykem jako takovým, mnozí vývojáři dávají přednost jiným textovým editorům nebo integrovaným vývojovým prostředím (Integrated Development Environment, čili IDE). Nebudu se zde jimi zabývat, ale komunita uživatelů jazyka Python udržuje seznam editorů podporujících jazyk Python, který pokrývá široké rozpětí podporovaných platforem a softwarových licencí.

Možná chcete nahlédnout i do seznamu ide podporujících jazyk Python, i když zatím pouze nemnohé z nich podporují Python 3. Jedním z těch, které jej podporují, je PyDev, zásuvný modul pro Eclipse, který změní Eclipse na plnohodnotné pythonovské integrované vývojové prostředí. Jak Eclipse, tak PyDev jsou multiplatformní a open source.

Z komerčních produktů jmenujme Komodo IDE společnosti ActiveState. Licence je vázána na uživatele. Studenti mohou získat slevu a k dispozici je i zkušební, časově omezená verze.

V jazyce Python programuji už devět let. Své programy edituji v prostředí GNU Emacs a ladím je v konzolovém pythonovském shellu. Při vývoji v jazyce Python není žádná cesta správnější nebo vyloženě špatná. Najděte si způsob, který vyhovuje právě vám!

> **“Don’t bury your burden in saintly silence. You have a problem? Great. Rejoice, dive in, and investigate.”**
> (Neutápějte své břímě ve svatém mlčení. Máte problém? Paráda. Radujte se, ponořte se do něj, bádejte.)
>
> — Ven. Henepola Gunaratana

1. Váš první pythonovský program

1. Váš první pythonovský program — 45
1.1. Ponořme se — **47**
1.2. Deklarace funkcí — **48**
1.2.1. Nepovinné a pojmenované argumenty — **49**
1.3. Psaní čitelného kódu — **51**
1.3.1. Dokumentační řetězce — **51**
1.4. Vyhledávací cesta pro `import` — **52**
1.5. Všechno je objekt — **53**
1.5.1. Co to vlastně je objekt? — **54**
1.6. Odsazování kódu — **54**
1.7. Výjimky — **55**
1.7.1. Obsluha chyb importu — **57**
1.8. Volné proměnné — **58**
1.9. Vše je citlivé na velikost písmen — **58**
1.10. Spouštění skriptů — **59**
1.11. Přečtěte si — **60**

1.1. Ponořme se

Konvence nám diktuje, že bych vás teď měl otravovat základními stavebními kameny, které s programováním souvisejí. A z nich bychom pak měli pomalu budovat něco užitečného. Přeskočme to. Tady máte úplný a funkční pythonovský program. Pravděpodobně vám bude zcela nepochopitelný. Žádné strachy. Rozpitváme ho řádek po řádku. Ale nejdříve si jej celý přečtěte a zjistěte, co z něj chápete (pokud vůbec něco).

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
    if size < 0:
        raise ValueError('number must be non-negative')

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('number too large')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

Spusťme program z příkazového řádku. Pod Windows to bude vypadat nějak takto:

```
c:\home\diveintopython3\examples> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

Pod Mac OS X nebo pod Linuxem to bude vypadat zase takhle:

```
you@localhost:~/diveintopython3/examples$ python3 humansize.py
1.0 TB
931.3 GiB
```

Co se to vlastně stalo? Spustili jste svůj první pythonovský program. Z příkazového řádku jste zavolali interpret jazyka Python a předali jste mu jméno skriptu, který měl být proveden. Uvedený skript definuje jedinou funkci, `approximate_size()`, která přebírá přesnou velikost souboru v bajtech a vypočítá velikost „v hezčím tvaru“ (ale přibližnou). (Pravděpodobně už jste něco podobného viděli v Průzkumníku Windows, v okně Finder na Mac OS X nebo v aplikacích Nautilus nebo Dolphin nebo Thunar na Linuxu. Když si necháte složku s dokumenty zobrazit v podobě vícesloupcového seznamu, uvidíte v tabulce ikonu dokumentu, jméno dokumentu, velikost, typ, datum poslední změny a tak dále. Pokud složka obsahuje soubor se jménem `TODO` a s velikostí 1093 bajtů, nezobrazí váš správce souborů `TODO 1093 bytes`. Místo toho se ukáže něco jako `TODO 1 KB`. A právě tohle dělá funkce `approximate_size()`.)

Podívejte se na konec skriptu a uvidíte dva řádky s voláním `print(approximate_size(argumenty))`. Jde o volání funkcí. Nejdříve se volá funkce `approximate_size()` a předávají se jí argumenty. Její návratová hodnota se předává přímo funkci `print()`. Funkce `print()` patří mezi zabudované (built-in). Její deklaraci nikdy neuvidíte. Můžete ji ale používat — kdykoliv a kdekoliv. (Zabudovaných funkcí existuje celá řada. A ještě mnohem více se jich nachází v různých *modulech*. Jen klid...)

Takže proč vlastně spuštěním skriptu z příkazového řádku získáme pokaždé stejný výstup? K tomu se ještě dostaneme. Nejdříve se podíváme na funkci `approximate_size()`.

1.2. Deklarace funkcí

Python pracuje s funkcemi podobně jako většina dalších jazyků, ale neodděluje hlavičkové soubory jako `c++` nebo sekce `rozhraní/implementace` jako Pascal. Pokud potřebujete nějakou funkci, prostě ji deklarujete, jako třeba zde:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

Deklarace funkce začíná klíčovým slovem `def`. Následuje jméno funkce a v závorce pak argumenty. Více argumentů se odděluje čárkami.

Pokud potřebujete nějakou funkci, prostě ji deklarujte.

Všimněte si, že funkce nedefinuje typ návratové hodnoty. Funkce v jazyce Python neurčují datový typ návratové hodnoty. Neurčují dokonce ani to, jestli vracejí hodnotu nebo ne. (Ve skutečnosti každá pythonovská funkce vrací hodnotu. Pokud funkce provede příkaz `return`, vrátí v něm uvedenou

hodnotu. V ostatních případech vrací `None`, což je pythonovský ekvivalent hodnoty null, nil, nic, žádná hodnota.)

> V některých jazycích funkce (které vracejí hodnotu) začínají slovem `function` a podprogramy (které nevracejí hodnotu) začínají slovem `sub`. Jazyk Python žádné podprogramy nezná. Vše jsou funkce, všechny funkce vracejí hodnotu (i když někdy je to `None`) a všechny funkce začínají slovem `def`.

Funkce `approximate_size()` přebírá dva argumenty — `size` a `a_kilobyte_is_1024_bytes` —, ale u žádného z nich není určen datový typ. V jazyce Python nemají proměnné explicitně určen typ nikdy. Python zjistí, jakého typu proměnná je, a vnitřně si to eviduje.

> V jazyce Java a v dalších jazycích se statickými datovými typy musíme určovat datový typ návratové hodnoty funkce a každého argumentu funkce. V jazyce Python nikdy explicitně neurčujeme datový typ čehokoliv. Python vnitřně sleduje datový typ podle toho, jakou hodnotu jsme přiřadili.

1.2.1. Nepovinné a pojmenované argumenty

Python umožňuje nastavit argumentům funkce implicitní hodnotu. Pokud funkci zavoláme bez zadání argumentu, získá argument svou implicitní hodnotu. Pokud použijeme pojmenované argumenty, můžeme je navíc (při volání funkce) zadat v libovolném pořadí.

Teď se na deklaraci funkce `approximate_size()` podíváme ještě jednou:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

U druhého argumentu, `a_kilobyte_is_1024_bytes`, je uvedena implicitní hodnota `True`. To znamená, že tento argument je *nepovinný*. Funkci můžeme zavolat, aniž bychom ho zadali. Python se bude chovat, jako kdybychom při volání funkce zadali na místě druhého argumentu hodnotu `True`.

Teď se podívejte na konec skriptu:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))               [1]
    print(approximate_size(1000000000000))                      [2]
```

[1] Zde se funkce `approximate_size()` volá s dvěma argumenty. Protože jsme druhému argumentu explicitně předali hodnotu `False`, nabývá `a_kilobyte_is_1024_bytes` uvnitř funkce `approximate_size()` hodnotu `False`.

[2] Zde se funkce `approximate_size()` volá pouze s jedním argumentem. Ale je to v pořádku, protože druhý argument je volitelný! A protože ho volající neurčil, nabývá druhý argument implicitní hodnoty `True` — přesně jak bylo určeno v deklaraci funkce.

Hodnotu argumentu můžeme do funkce předat také jako pojmenovanou.

```
>>> from humansize import approximate_size
>>> approximate_size(4000, a_kilobyte_is_1024_bytes=False)       [1]
'4.0 KB'
>>> approximate_size(size=4000, a_kilobyte_is_1024_bytes=False)  [2]
'4.0 KB'
>>> approximate_size(a_kilobyte_is_1024_bytes=False, size=4000)  [3]
'4.0 KB'
>>> approximate_size(a_kilobyte_is_1024_bytes=False, 4000)       [4]
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> approximate_size(size=4000, False)                           [5]
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

[1] Zde se funkce `approximate_size()` volá s hodnotou prvního argumentu `4000` (`size`) a s hodnotou `False` pro pojmenovaný argument `a_kilobyte_is_1024_bytes`. (Shodou okolností je to druhý argument, ale na tom nezáleží — jak uvidíte o chvíli později.)

[1] Zde se funkce `approximate_size()` volá s hodnotou `4000` pro pojmenovaný argument `size` a s hodnotou `False` pro pojmenovaný argument `a_kilobyte_is_1024_bytes`. (Pojmenované argumenty jsou zde shodou okolností uvedeny ve stejném pořadí, v jakém jsou uvedeny v deklaraci funkce, ale na tom rovněž nezáleží.)

[3] Zde se funkce `approximate_size()` volá s hodnotou `False` pro pojmenovaný argument `a_kilobyte_is_1024_bytes` a s hodnotou `4000` pro pojmenovaný argument `size`. (Vidíte? Já jsem vám říkal, že na pořadí nezáleží.)

[4] Toto volání selhalo, protože jsme použili pojmenovaný argument a teprve po něm následoval nepojmenovaný (poziční) argument. Tohle nefunguje nikdy. Při čtení seznamu argumentů zleva doprava se po použití prvního pojmenovaného argumentu musí všechny následující argumenty uvést také jako pojmenované.

[5] Toto volání rovněž selhává — ze stejného důvodu jako předchozí volání. Je to tak překvapivé? Když se to tak vezme, předáváme hodnotu `4000` pro pojmenovaný argument `size` a je „zřejmé“, že hodnota `False` byla myšlena jako hodnota argumentu `a_kilobyte_is_1024_bytes`. Ale Python tímto způsobem nefunguje. Jakmile použijeme pojmenovaný argument, všechny argumenty uvedené napravo od něj musí být také pojmenované.

1.3. Psaní čitelného kódu

Nebudu vás zde nudit dlouhým proslovem o důležitosti dokumentování vašeho kódu. Jen si uvědomte, že kód se píše jednou, ale čte se mnohokrát. A nejdůležitějším čtenářem vašeho zdrojového textu budete vy sami — šest měsíců poté, co jste jej napsali (to znamená poté, co už jste o něm všechno zapomněli a máte v něm něco opravit). V jazyce Python se čitelný kód píše snadno, takže toho využijte. Za šest měsíců mi poděkujete.

1.3.1. Dokumentační řetězce

Pythonovskou funkci můžete zdokumentovat tím, že jí přidělíte dokumentační řetězec (zkráceně `docstring`). V našem programu je u funkce `approximate_size()` dokumentační řetězec uveden:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
```

Každá funkce si zaslouží decentní docstring.

Tři apostrofy uvozují víceřádkový řetězec. Vše mezi počátečními a koncovými apostrofy (nebo uvozovkami) se stává součástí jediného řetězce, včetně konců řádků, úvodních bílých znaků a jednoduchých apostrofů. Víceřádkové řetězce můžete použít kdekoliv, ale nejčastěji se s nimi setkáte při zápisech dokumentačních řetězců.

> Použití ztrojených apostrofů představuje rovněž jednoduchý způsob pro zápis řetězců, ve kterých se vyskytují jak apostrofy, tak uvozovky. Chovají se jako zápis `qq/.../` v jazyce Perl 5.

Vše, co se nachází mezi ztrojenými apostrofy, je dokumentační řetězec, který popisuje, co funkce dělá. Pokud `docstring` existuje, pak to musí být první věc, která se v těle funkce objeví. (To znamená, že musí být uveden na řádku následujícím za deklarací funkce.) Z technického pohledu není nutné `docstring` funkci vůbec přidělovat, ale prakticky byste to měli udělat vždy. Já vím, že jste o tom slyšeli v každém kurzu programování, který jste navštěvovali. Ale u jazyka Python máme jeden motivační faktor navíc: `docstring` je dostupný za běhu programu v podobě atributu (vlastnosti) funkce.

> Mnohá pythonovská integrovaná vývojová prostředí používají docstring pro účely kontextově citlivé nápovědy. To znamená, že po napsání jména funkce se její docstring zobrazí v podobě tooltipu (tj. malého informačního okénka zobrazovaného poblíž daného místa). Může to být velmi užitečné, ale bude to dobré jen tak, jak dobře napíšete dokumentační řetězce.

1.4. Vyhledávací cesta pro import

Než půjdeme dál, chtěl bych se stručně zmínit o vyhledávací cestě pro knihovny (library search path). Když se pokoušíte importovat modul, hledá jej Python na několika místech. Přesněji řečeno, hledá jej ve všech adresářích, které jsou definovány proměnnou sys.path. Jde o běžný seznam a jeho obsah můžete snadno zobrazit nebo měnit prostřednictvím standardních metod seznamu. (O seznamech se dozvíme více v kapitole Přirozené datové typy.)

```
>>> import sys                                                    [1]
>>> sys.path                                                      [2]
['',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
>>> sys                                                           [3]
<module 'sys' (built-in)>
>>> sys.path.insert(0, '/home/mark/diveintopython3/examples')     [4]
>>> sys.path                                                      [5]
['/home/mark/diveintopython3/examples',
 '',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
```

[1] Importováním modulu sys zpřístupníme všechny jeho funkce a atributy.

[2] sys.path je seznam adresářů, které tvoří aktuální vyhledávací cestu. (U vás to bude vypadat jinak v závislosti na vašem operačním systému, na verzi Pythonu, který používáte, a na tom, kam byl nainstalován.) Pokud se pokoušíte o import, hledá Python soubor s daným jménem a příponou .py právě v těchto adresářích (v uvedeném pořadí).

[3] No, ve skutečnosti jsem trochu zalhal. Pravda je o něco komplikovanější, protože ne všechny moduly jsou uloženy v podobě souborů s příponou .py. U některých jde o *zabudované (built-in)*

moduly. Ve skutečnosti jsou součástí programu Python. Zabudované moduly se chovají úplně stejně jako běžné moduly, ale není k nim k dispozici pythonovský zdrojový kód, protože nejsou napsány v jazyce Python! Zabudované moduly jsou napsány v jazyce C, stejně jako samotný Python.

[4] K pythonovské vyhledávací cestě můžete za běhu přidat nový adresář tím, že jeho jméno přidáte do sys.path. Kdykoliv se od toho okamžiku pokusíte importovat nějaký modul, Python bude prohledávat i tento adresář. Efekt trvá tak dlouho, dokud Python běží.

[5] Použitím příkazu `sys.path.insert(0, new_path)` jsme vložili nový adresář jako první položku seznamu `sys.path`, což znamená, že se ocitla na začátku pythonovské vyhledávací cesty. Většinou potřebujeme právě tohle. V případě konfliktu jmen (například když se Python dodává s konkrétní knihovnou verze 2, ale my chceme použít tutéž knihovnu ve verzi 3) uvedeným obratem zajistíme, že námi požadované moduly budou nalezeny dříve než moduly dodané s Pythonem.

1.5. Všechno je objekt

Pokud vám to náhodou uniklo, řekli jsme si, že pythonovské funkce mají atributy a tyto atributy jsou přístupné za běhu programu. Funkce, stejně jako všechno ostatní v Pythonu, je objektem.

Spusťme interaktivní pythonovský shell a vyzkoušejme si:

```
>>> import humansize                                [1]
>>> print(humansize.approximate_size(4096, True))  [2]
4.0 KiB
>>> print(humansize.approximate_size.__doc__)      [3]
Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string
```

[1] Na prvním řádku importujeme program `humansize` jako modul — kus kódu, který můžeme používat interaktivně nebo z většího pythonovského programu. Jakmile je import modulu proveden, můžeme se odkazovat na jeho veřejné funkce, třídy nebo atributy. Moduly mohou dělat totéž, čímž si zpřístupňují funkčnost z jiných modulů. A my to můžeme udělat v interaktivním pythonovském shellu také. Tato koncepce je důležitá a v knize se s ní potkáme ještě mnohokrát.

[2] Pokud chceme použít funkce definované v importovaných modulech, musíme uvést i jméno modulu. Takže nestačí napsat jen `approximate_size`. Musíme uvést `humansize.approximate_size`. Pokud jste používali třídy v jazyce Java, mělo by vám to něco připomínat.

[3] Zde se místo očekávaného volání funkce ptáme na jeden z jejích atributů, který je nazván `__doc__`.

> Pythonovský příkaz `import` se podobá příkazu `require` v jazyce Perl. Jakmile provedeme `import` pythonovského modulu, vyjadřujeme přístup k jeho funkcím zápisem `modul.funkce`. Jakmile v jazyce Perl provedeme příkaz `require`, dostaneme se na jeho funkce zápisem `modul::funkce`.

1.5.1. Co to vlastně je objekt?

V Pythonu je objektem všechno. A vše může mít atributy a metody. Všechny funkce mají zabudovaný atribut `__doc__`, který vrací dokumentační řetězec funkce definovaný ve zdrojovém souboru. Modul `sys` je objekt, který (mimo jiné) má atribut zvaný `path`. A tak dále.

Tím ale stále neodpovídáme na základnější otázku: Co je to vlastně objekt? Různé programovací jazyky definují „objekt“ různým způsobem. V některých jazycích to znamená, že *všechny* objekty *musí* mít atributy a metody. V jiných jazycích to znamená, že všechny objekty lze rozdělit do tříd. Jazyk Python definuje objekt volněji. Některé objekty nemusí mít ani atributy ani metody, *ale mohou je mít*. Ne všechny objekty mají svou třídu. Ale vše je objektem v tom smyslu, že to může být přiřazeno do proměnné nebo předáno jako argument funkce.

V jiných souvislostech s programováním jste už možná slyšeli pojem „prvotřídní objekt“ („first-class object“). Kvůli lepší srozumitelnosti mu říkejme (opisem) *plnohodnotný objekt*. V jazyce Python je *plnohodnotným objektem* i funkce. Funkci můžeme předat jako argument jiné funkci. Moduly jsou rovněž *plnohodnotnými objekty*. Funkci můžeme předat jako argument celý modul. Třídy jsou také plnohodnotné objekty a jednotlivé instance třídy jsou rovněž plnohodnotnými objekty.

To je velmi důležité, takže pro případ, že by vám to na začátku párkrát uteklo, zopakuji znovu: *V jazyce Python je všechno objektem.* Řetězce jsou objekty. Seznamy jsou objekty. Funkce jsou objekty. Třídy jsou objekty. Instance tříd jsou objekty. Dokonce moduly jsou objekty.

1.6. Odsazování kódu

V jazyce Python se pro označování míst, kde kód funkce začíná a kde končí, nepoužívají slova `begin` a `end` a ani žádné složené závorky. Jediným oddělovačem těla je dvojtečka `(:)` a odsazení kódu.

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):  [1]
    if size < 0:                                            [2]
        raise ValueError('number must be non-negative')     [3]
                                                            [4]
```

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
 for suffix in SUFFIXES[multiple]:                          [5]
     size /= multiple
     if size < multiple:
         return '{0:.1f} {1}'.format(size, suffix)

 raise ValueError('number too large')
```

[1] Bloky kódu (bloky zdrojového textu) jsou určeny jejich odsazením. „Blokem kódu“ zde rozumím volání funkcí, příkazy `if`, cykly `for`, cykly `while` a další. Blok je zahájen odsazením (odskočením řádku vpravo) a končí předsazením (odskočením následujícího řádku vlevo). Nenajdeme zde žádné explicitní závorky nebo klíčová slova. To ale znamená, že používání bílých znaků má svůj význam a že je musíme užívat důsledně. V tomto příkladu je kód funkce odsazen o čtyři mezery. Nemusí to být zrovna čtyři mezery, ale musíme použít stejné odsazení. První řádek, který není odsazený, označuje konec funkce.

[2] V Pythonu za příkazem `if` následuje blok kódu. Pokud výraz za `if` nabývá hodnoty `true`, provede se následující odsazený blok. V opačném případě se provede blok za `else` (pokud je uveden). Povšimněte si, že kolem výrazu chybí závorky.

[3] Tento řádek se nachází v bloku kódu, který je uvnitř příkazu `if`. Příkaz `raise` vyvolá výjimku (typu `ValueError`), ale jen v případě, kdy platí `size < 0`.

[4] Zde ještě není konec funkce. Zcela prázdné řádky se nepočítají. Díky nim může být kód čitelnější, ale nepovažují se za oddělovače bloků kódu. Na dalším řádku funkce pokračuje.

[5] Rovněž příkaz cyklu `for` zahajuje blok kódu. Bloky kódu se mohou skládat z mnoha řádků, ale všechny musí být odsazeny stejně. Tento cyklus `for` má blok s třemi řádky kódu. Pro víceřádkové bloky kódu se nepoužívá žádná jiná zvláštní syntaxe. Prostě odsadíme a jedeme dál.

Po počátečních protestech a sarkastických přirovnáních k Fortranu si na to zvyknete a zjistíte, jaké to má výhody. Jedna z největších výhod spočívá v tom, že všechny pythonovské programy vypadají podobně, protože odsazování je vynuceno samotným jazykem a není jen věcí stylu. Pythonovský kód napsaný někým jiným se proto snadněji čte a je srozumitelnější.

> Python používá k oddělování příkazů konec řádku. Oddělení bloku kódu se vyjadřuje dvojtečkou a odsazením. Jazyky `c++` a Java používají k oddělování příkazů středník a k oddělování bloku kódu složené závorky.

1.7. Výjimky

V jazyce Python najdete výjimky všude. Používá je prakticky každý modul standardní pythonovské knihovny a samotný Python je vyvolává při mnoha různých okolnostech. V celé této knize se s nimi budete opakovaně setkávat.

Co to vlastně je výjimka? Obvykle jde o projev nějaké chyby. Vyjadřuje, že něco nedopadlo dobře. (Ne všechny výjimky jsou vyjádřením chyby. Ale v tomto okamžiku na tom nezáleží.) V některých programovacích jazycích jsme vedeni k používání návratových chybových kódů, které pak *kontrolujeme*. Python nás vede k používání výjimek, které pak *obsluhujeme*.

Když se v pythonovském shellu objeví chyba, vypíše nějaké podrobnosti o výjimce a jak k ní došlo. A to je právě ono. Říkáme tomu *neobsloužená* výjimka. V okamžiku vyvolání výjimky se v okolí nenacházel žádný kód, který by si toho všímal a který by se jí zabýval. Takže výjimka probublala zpět až do horních úrovní pythonovského shellu. Ten vyplivnul nějaké ladicí informace a považoval to za vyřešené. Pokud se to stane při práci v shellu, není to žádná pohroma. Ale pokud by se to stalo u vašeho skutečného pythonovského programu, pak by za předpokladu, že výjimku nic neobsloužilo, došlo ke skřípavému zastavení jeho běhu. Možná by vám to vyhovovalo, možná ne.

> V Pythonu nemusí funkce deklarovat, jaké výjimky mohou vyvolat — na rozdíl od jazyka Java. Rozhodnutí o tom, jaké možné výjimky potřebujete odchytávat, záleží zcela na vás.

Ale výjimka nemusí vést k úplnému krachu programu. Výjimky mohou být *obslouženy*. Někdy je výjimka opravdu důsledkem chyby ve vašem programu (když se například pokoušíte použít proměnnou, která neexistuje), ale někdy je výjimka výsledkem něčeho, co se dalo předvídat. Když otvíráte soubor, nemusí třeba existovat. Když importujete modul, nemusel být nainstalován. Když se připojujete k databázi, může být nedostupná nebo k ní nemůžete přistupovat kvůli nedostatečným bezpečnostním oprávněním. Pokud víte, že na nějakém řádku může vzniknout výjimka, měli byste ji obsloužit pomocí konstrukce `try...except`.

> Python používá bloky `try...except` k obsluze výjimek. Příkaz `raise` používá k jejich generování. Jazyky Java a c++ používají k obsloužení výjimek bloky `try...catch`. K jejich generování používají příkaz `throw`.

Funkce `approximate_size()` vyvolává výjimky ve dvou různých případech: když je zadaná velikost (`size`) větší, než pro jakou byla funkce navržena, nebo když je zadaná velikost menší než nula.

```
if size < 0:
    raise ValueError('number must be non-negative')
```

Syntaxe pro vyvolání výjimky je poměrně jednoduchá. Použijeme příkaz `raise`, za kterým uvedeme jméno výjimky a nepovinný, pro člověka srozumitelný řetězec usnadňující ladění. Zápis se podobá volání funkce. (Ve skutečnosti jsou výjimky implementovány jako třídy. Příkaz raise zde vytváří instanci třídy `ValueError` a její inicializační metodě předává řetězec `'number must be non-negative'` (číslo nesmí být záporné). Ale nepředbíhejme!)

> Výjimka nemusí být obsloužena ve funkci, která ji vyvolala. Pokud ji jedna funkce neobslouží, výjimka bude předána volající funkci, pak funkci, která vyvolala zase ji a tak dále, „nahoru po zásobníku“. Pokud není výjimka obsloužena vůbec, program zhavaruje a Python vypíše „traceback“

(trasovací výpis) na standardní chybový výstup a tím to končí. Znovu opakuji, možná takové chování požadujeme. Záleží to na tom, k čemu je náš program určen.

1.7.1. Obsluha chyb importu

Jednou ze zabudovaných výjimek jazyka Python je `ImportError`. Ta je vyvolána v okamžiku, kdy se pokoušíme o import modulu a tato operace selže. Může k tomu dojít z různých důvodů, ale v nejjednodušším případě modul nebyl nalezen ve vaší vyhledávací cestě pro import. Toho můžete využít pro zabudování nepovinných vlastností svého programu. Tak například knihovna `chardet` umožňuje autodetekci znakového kódování. Možná byste chtěli, aby váš program tuto knihovnu využil *v případě, že existuje.* Pokud ji uživatel nemá nainstalovanou, měl by program bez mrknutí oka pokračovat. Můžeme toho dosáhnout použitím bloku `try..except`.

```
try:
  import chardet
except ImportError:
  chardet = None
```

Později můžete otestovat, zda je modul `chardet` přítomen — jednoduše, příkazem `if`:

```
if chardet:
  # do something
else:
  # continue anyway
```

Další běžný případ použití výjimky `ImportError` souvisí se situací, kdy dva moduly implementují společné aplikační programové rozhraní (`API`), ale jeden z nich chceme používat přednostně. (Možná je rychlejší nebo používá méně paměti.) Můžeme zkusit importovat jeden modul, ale pokud import selže, vezmeme zavděk tím druhým. Tak například kapitola o XML pojednává o dvou modulech, které implementují společné rozhraní zvané `ElementTree`. Prvním z nich je lxml, což je modul třetí strany, který si musíte sami stáhnout a nainstalovat. Tím druhým je xml.etree.ElementTree, který je sice pomalejší, ale je součástí standardní knihovny jazyka Python 3.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

Na konci bloku `try..except` máte zpřístupněný některý z těchto modulů a máte jej pojmenovaný `etree`. Protože oba moduly implementují stejné rozhraní (`API`), nemusíte ve zbytku svého kódu neustále testovat, který modul se vlastně naimportoval. A protože se modul, který se opravdu naimportoval, vždy jmenuje `etree`, nemusí být zbytek vašeho kódu zaneřáděný příkazy `if`, ve kterých se volají různě pojmenované moduly.

1.8. Volné proměnné
1.9. Vše je citlivé na velikost písmen

1.8. Volné proměnné

Podívejme se znovu na následující řádek kódu funkce `approximate_size()`:

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

Proměnnou `multiple` (násobek) jsme nikde nedeklarovali. Pouze jsme do ní přiřadili hodnotu. To je v pořádku, protože Python vám tohle dovolí. Co už vám ale Python *nedovolí*, je pokus o odkaz na proměnnou, které nebyla nikdy přiřazena hodnota. Pokud se o to pokusíme, bude vyvolána výjimka `NameError`.

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 1
>>> x
1
```

Jednoho dne za to Pythonu poděkujete.

1.9. Vše je citlivé na velikost písmen

V jazyce Python je zápis všech jmen citlivý na velikost písmen. Týká se to jmen proměnných, jmen funkcí, jmen tříd, jmen modulů, jmen výjimek. Pokud to můžete zpřístupnit, nastavit, zavolat, importovat nebo to vyvolat, je to citlivé na velikost písmen.

```
>>> an_integer = 1
>>> an_integer
1
>>> AN_INTEGER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'AN_INTEGER' is not defined
>>> An_Integer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'An_Integer' is not defined
>>> an_inteGer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'an_inteGer' is not defined
```

A tak dále.

1.10. Spouštění skriptů

V Pythonu je objektem všechno.

V Pythonu je objektem i modul a moduly definují několik užitečných atributů. Při psaní vašich modulů toho můžeme využít k jejich snadnému testování. Vložíme do nich speciální blok kódu, který se provede v případě, kdy pythonovský soubor spustíte z příkazového řádku. Podívejte se na poslední řádky v souboru `humansize.py`:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

> Python — stejně jako jazyk `C` — používá == pro porovnání a = pro přiřazení. Na rozdíl od jazyka `C` ale Python nepodporuje přiřazovací výraz, takže odpadá možnost nechtěného přiřazení hodnoty v situaci, kdy jste měli na mysli test na rovnost.

Takže čím je vlastně tento příkaz `if` zvláštní? Tak tedy, moduly jsou objekty a všechny moduly mají zabudovaný atribut `__name__`. Jeho hodnota závisí na tom, jakým způsobem modul používáte. Pokud provádíte import modulu, pak je v atributu `__name__` zachyceno jméno jeho souboru bez cesty do adresáře a bez přípony.

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

Ale modul můžete spustit také přímo, jako samostatný program. V takovém případě bude `__name__` nabývat speciální přednastavené hodnoty `__main__`. Python tuto skutečnost otestuje příkazem `if`, zjistí, že výraz platí, a provede blok kódu uvnitř `if`. V našem případě se vytisknou dvě hodnoty.

```
c:\home\diveintopython3> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

A tohle všechno dělá váš první pythonovský program!

1.11. Přečtěte si

- **PEP 257: Docstring Conventions.** Najdete zde vysvětlení, čím se liší dobrý `docstring` od vynikajícího `docstringu`.
  (http://www.python.org/dev/peps/pep-0257/)
- **Python Tutorial: Documentation Strings** — dotýká se stejného tématu.
  (http://docs.python.org/py3k/tutorial/controlflow.html)
- **PEP 8: Style Guide for Python Code** pojednává o vhodných způsobech odsazování.
  (http://www.python.org/dev/peps/pep-0008/)
- **Python Reference Manual** vysvětluje co to znamená, když se řekne, že vše v Pythonu je objekt, protože někteří lidé jsou puntičkáři a rádi o takových věcech dlouze diskutují.
  (http://docs.python.org/py3k/reference/)

“Wonder is the foundation of all philosophy, inquiry its progress, ignorance its end.”
(Zvědavost je základem celé filozofie, hledání odpovědí na otázky ji žene vpřed, ignorance ji zabíjí.)

— Michel de Montaigne

2. Přirozené datové typy

2. Přirozené datové typy — 61
2.1. Ponořme se — **63**
2.2. Booleovský typ — **63**
2.3. Čísla — **64**
2.3.1. Vynucení převodu celých čísel na reálná a naopak — **65**
2.3.2. Běžné operace s čísly — **66**
2.3.3. Zlomky — **67**
2.3.4. Trigonometrie — **67**
2.3.5. Čísla v booleovském kontextu — **68**
2.4. Seznamy — **69**
2.4.1. Vytvoření seznamu — **69**
2.4.2. Vytváření podseznamů — **70**
2.4.3. Přidávání položek do seznamu — **71**
2.4.4. Vyhledávání hodnoty v seznamu — **73**
2.4.5 Odstraňování položek ze seznamu — **74**
2.4.6. Odstraňování položek ze seznamu: Bonusové kolo — **75**
2.4.7. Seznamy v booleovském kontextu — **75**
2.5. N-tice — **76**
2.5.1. N-tice v booleovském kontextu — **78**
2.5.2. Přiřazení více hodnot najednou — **78**
2.6. Množiny — **79**
2.6.1. Vytvoření množiny — **79**
2.6.2. Úprava množiny — **81**
2.6.3. Odstraňování položek z množiny — **82**
2.6.4. Běžné množinové operace — **83**
2.6.5. Množiny v booleovském kontextu — **85**
2.7. Slovníky — **86**
2.7.1. Vytvoření slovníku — **86**
2.7.2. Úprava slovníku — **87**
2.7.3. Slovníky se smíšeným obsahem — **87**
2.7.4. Slovníky v booleovském kontextu — **88**
2.8. `None` — **89**
2.8.1. `None` v booleovském kontextu — **90**
2.9. Přečtěte si — **90**

2.1. Ponořme se
2.2. Booleovský typ

2.1. Ponořme se

Datové typy. Přestaňme si na chvíli všímat našeho prvního pythonovského programu a pojďme si popovídat o datových typech. Každá hodnota v Pythonu je určitého datového typu, ale u proměnných nemusíme datový typ deklarovat. Jak to tedy funguje? Při každém přiřazení hodnoty do proměnné si Python zjistí, jakého typu hodnota je, a vnitřně si to eviduje.

Python používá mnoho přirozených datových typů (ve smyslu „přirozených pro Python“). Uveďme zde ty hlavní:

1. **Boolean** (booleovský typ) nabývá buď hodnoty `True` nebo `False`.
2. **Čísla** mohou být celá (integer; `1` a `2`), reálná (float; `1.1` a `1.2`), zlomky (fraction; `1/2` and `2/3`), nebo dokonce čísla komplexní.
3. **Řetězce** jsou posloupnosti Unicode znaků. Tuto podobu může mít například `HTML` dokument.
4. **Bajty a pole bajtů**, například soubor s obrázkem ve formátu `JPEG`.
5. **Seznamy** jsou uspořádané posloupnosti hodnot.
6. **N-tice** jsou uspořádané, neměnné posloupnosti hodnot.
7. **Množiny** jsou neuspořádané kolekce hodnot.
8. **Slovníky** jsou neuspořádané kolekce dvojic klíč-hodnota.

Těch typů je samozřejmě víc. V Pythonu je vše objektem, proto musí existovat také typy jako *modul, funkce, třída, metoda, soubor,* a dokonce *přeložený kód.* S některými z nich už jsme se setkali: moduly mají jména, funkce mají docstring atd. O třídách se dozvíte v kapitole Třídy a iterátory, o souborech v kapitole Soubory.

Kap.
Kap.

Řetězce a bajty jsou důležité do té míry — a jsou také dost komplikované —, že jim je věnována samostatná kapitola. Nejdříve se podívejme na ty zbývající.

2.2. Booleovský typ

Objekt booleovského typu nabývá buď hodnoty true (pravda) nebo false (nepravda). Pro přímé přiřazení booleovských hodnot definuje Python dvě konstanty, příhodně pojmenované `True` a `False`. Booleovská hodnota může vzniknout také vyhodnocením výrazu. Na některých místech (jako u příkazu `if`) Python dokonce předpokládá, že se výraz vyhodnotí do podoby booleovské hodnoty. Těmto místům se říká *booleovský kontext.* V booleovském kontextu můžeme použít téměř libovolný výraz. Python se pokusí získat jeho pravdivostní hodnotu. Pravidla, podle kterých se v booleovském kontextu výsledek chápe jako pravdivý nebo nepravdivý (true nebo false), jsou pro různé datové typy různá. (Jakmile uvidíte dále v této kapitole konkrétní příklady, bude vám to dávat větší smysl.)

V booleovském kontextu můžete použít téměř libovolný výraz.

Vezměme si například následující úryvek z humansize.py:

```
if size < 0:
    raise ValueError('number must be non-negative')
```

Proměnná size obsahuje celé číslo, 0 je celé číslo a < je číselný operátor. Výsledek výrazu size < 0 má vždy booleovskou hodnotu. V pythonovském shellu si vyzkoušejte následující:

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
>>> size = -1
>>> size < 0
True
```

V důsledku problematického dědictví z Pythonu 2 se s booleovskými hodnotami může zacházet jako s čísly. True je 1; False je 0.

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Ajajaj! Takové věci nedělejte. Zapomeňte, že jsem se o tom vůbec zmínil.

2.3. Čísla

Čísla jsou obdivuhodná. Můžete si je vybrat z tak ohromného množství. Python podporuje jak celá čísla (integer), tak čísla reálná (floating point). Nerozlišují se deklarací datového typu. Python je od sebe poznává podle přítomnosti nebo nepřítomnosti desetinné tečky.

```
>>> type(1)                 [1]
<class 'int'>
>>> isinstance(1, int)      [2]
True
>>> 1 + 1                   [3]
2
>>> 1 + 1.0                 [4]
2.0
>>> type(2.0)
<class 'float'>
```

[1] Pro ověření typu libovolné hodnoty nebo proměnné můžeme použít funkci `type()`. Jak se dalo čekat, hodnota `1` je typu `int`.

[2] Podobně můžeme voláním funkce `isinstance()` ověřit, zda hodnota či proměnná odpovídá zadanému typu.

[3] Přidáním `int` k `int` vzniká výsledek typu `int`.

[4] Přidáním `int` k `float` vzniká výsledek typu `float`. Aby mohl Python provést sčítání, vynutí si převod typu `int` na `float`. Poté vrátí výsledek typu `float`.

2.3.1. Vynucení převodu celých čísel na reálná a naopak

Jak jste zrovna viděli, některé operátory (například sčítání) mohou podle potřeby vynutit převod celého čísla na číslo reálné. Ale k převodu je můžete donutit taky vy sami.

```
>>> float(2)                [1]
2.0
>>> int(2.0)                [2]
2
>>> int(2.5)                [3]
2
>>> int(-2.5)               [4]
-2
>>> 1.12345678901234567890  [5]
1.1234567890123457
>>> type(1000000000000000)  [6]
<class 'int'>
```

[1] Voláním funkce `float()` můžeme explicitně vynutit převod `int` (typ pro celé číslo) na `float` (typ pro reálné číslo).

[2] A nebude asi moc překvapivé, že voláním `int()` můžeme vynutit převod `float` na `int`.

[3] Funkce `int()` nezaokrouhluje, ale odsekává.

[4] Funkce `int()` odsekává desetinnou část u záporných čísel směrem k nule. Jde o funkci

opravdového odsekávání, ne o funkci `floor` (tj. u záporných čísel dojde ke zvětšení čísla, protože například `-2.5` se změní na `-2`).

[5] Čísla typu `float` jsou uložena s přesností na 15 desetinných míst.

[6] Celá čísla mohou být libovolně velká.

> Python 2 měl oddělené typy `int` a `long`. Datový typ `int` byl omezen konstantou `sys.maxint`, která byla platformově závislá, ale obvykle nabývala hodnoty $2^{32}-1$. Python 3 má pouze jeden celočíselný typ, který se chová většinou jako původní typ `long` z Pythonu 2. Detaily naleznete v PEP 237.

2.3.2. Běžné operace s čísly

S čísly můžete dělat všechno možné.

```
>>> 11 / 2       [1]
5.5
>>> 11 // 2      [2]
5
>>> -11 // 2     [3]
-6
>>> 11.0 // 2    [4]
5.0
>>> 11 ** 2      [5]
121
>>> 11 % 2       [6]
1
```

[1] Operátor `/` provádí dělení. Vrací výsledek typu `float` dokonce i v případě, že činitel i jmenovatel jsou typu `int`.

[2] Operátor `//` provádí svým způsobem podivné celočíselné dělení. Pokud je výsledek kladný, můžete o něm uvažovat, že vznikl odseknutím desetinných míst (tedy nikoliv zaokrouhlením). Ale pozor na to.

[3] Při celočíselném dělení záporných čísel provede operátor `//` zaokrouhlení „nahoru“ k nejbližšímu celému číslu. Z matematického hlediska zaokrouhluje „dolů“, protože `-6` je menší než `-5`. Ale pokud byste očekávali, že dojde k odseknutí na `-5`, tak byste se nachytali.

[4] Operátor `//` nevrací celé číslo vždy. Pokud je čitatel nebo jmenovatel typu `float`, bude výsledek sice opět zaokrouhlen na celé číslo, ale výsledná hodnota bude typu `float`.

[5] Operátor `**` znamená „umocněno na“. 11^2 je 121.

[6] Operátor `%` vrací zbytek po celočíselném dělení. 11 děleno 2 je 5 a zbytek je 1. Takže výsledkem bude 1.

> V Pythonu 2 obvykle operátor `/` prováděl celočíselné dělení. Ale když jste ve svém kódu použili speciální direktivu, mohli jste jeho význam přepnout na reálné dělení. V Pythonu 3 operátor `/`

vyjadřuje vždy dělení s reálným výsledkem (floating point division). Na detaily se podívejte do PEP 238.

2.3.3. Zlomky

Python vás neomezuje jen na celá a reálná čísla. Zvládne celou tu fantastickou matiku, kterou jste se učili na střední škole a rychle jste ji zapomněli.

```
>>> import fractions                    [1]
>>> x = fractions.Fraction(1, 3)        [2]
>>> x
Fraction(1, 3)
>>> x * 2                               [3]
Fraction(2, 3)
>>> fractions.Fraction(6, 4)            [4]
Fraction(3, 2)
>>> fractions.Fraction(0, 0)            [5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fractions.py", line 96, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(0, 0)
```

[1] Používání zlomků zahájíme importem modulu `fractions`.

[2] Zlomek definujeme tak, že vytvoříme objekt třídy `Fraction` a předáme mu čitatele a jmenovatele.

[3] Se zlomky můžeme provádět obvyklé matematické operace. Ty vracejí nový objekt třídy `Fraction`. `2 * (1/3) = (2/3)`

[4] Objekt třídy `Fraction` zlomky automaticky krátí. `(6/4) = (3/2)`

[5] Python má dost rozumu na to, aby nevytvořil zlomek s nulovým jmenovatelem.

2.3.4. Trigonometrie

Python zvládne i základy trigonometrie.

```
>>> import math
>>> math.pi                     [1]
3.1415926535897931
>>> math.sin(math.pi / 2)       [2]
1.0
>>> math.tan(math.pi / 4)       [3]
0.99999999999999989
```

[1] Modul `math` definuje konstantu `ϖ`, čili poměr mezi obvodem kružnice a jejím průměrem.
[2] Modul `math` zvládá všechny základní trigonometrické funkce včetně `sin()`, `cos()`, `tan()` a varianty jako `asin()`.
[3] Ale pozor na to, že Python neoplývá nekonečnou přesností. Funkce `tan(ϖ / 4)` by měla vrátit `1.0` a ne `0.99999999999999989`.

2.3.5. Čísla v booleovském kontextu

Nulová hodnota se interpretuje jako false, nenulová jako true.

Čísla můžete použít v booleovském kontextu — například v příkazu `if`. Nulové hodnoty se interpretují jako false, nenulové jako true.

```
>>> def is_it_true(anything):                  [1]
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(1)                              [2]
yes, it's true
>>> is_it_true(-1)
yes, it's true
>>> is_it_true(0)
no, it's false
>>> is_it_true(0.1)                            [3]
yes, it's true
>>> is_it_true(0.0)
no, it's false
>>> import fractions
>>> is_it_true(fractions.Fraction(1, 2))       [4]
yes, it's true
>>> is_it_true(fractions.Fraction(0, 1))
no, it's false
```

[1] A to víte, že své vlastní funkce můžete definovat i v pythonovském interaktivním shellu? Stačí zmáčknout `ENTER` na konci každého řádku a vše ukončit stiskem `ENTER` na prázdném řádku.
[2] V booleovském kontextu se nenulová celá čísla chápou jako true a nula jako false.
[3] Nenulová reálná čísla se chápou jako true, `0.0` se chápe jako false. Ale bacha na tu poslední hodnotu! Pokud dojde k sebemenší zaokrouhlovací chybě (což není nemožné, jak jste si

mohli všimnout v předchozí podkapitole), pak bude Python testovat místo nuly například `0.0000000000001` a vrátí hodnotu `True`.

[5] Zlomky můžeme také použít v booleovském kontextu. Hodnota `Fraction(0, n)` se pro všechny hodnoty n vyhodnotí jako false. Všechny ostatní zlomky se vyhodnotí jako true.

2.4. Seznamy

Seznamy jsou v Pythonu nejpoužívanějšími datovými typy. Když řeknu „seznam“ (anglicky list [list]), může vás napadnout „pole, jehož velikost musím předem deklarovat, které může obsahovat jen prvky stejného typu atd.“. Tímto směrem neuvažujte. Seznamy jsou mnohem lepší.

> Pythonovský seznam se podobá poli (array) v Perl 5. Proměnné polí v jazyce Perl 5 vždycky začínají znakem `@`. Pythonovské proměnné můžou být pojmenovány zcela libovolně. Python si vnitřně eviduje jejich datový typ.

> Pythonovský seznam má větší možnosti než pole (array) v jazyce Java. (Ačkoliv pokud je to vše, co od života očekáváte, můžete jej tímto způsobem používat.) Podobnější je mu třída `ArrayList`, která umožňuje uchovávání libovolných objektů a při přidání nových položek se může dynamicky zvětšit.

2.4.1. Vytvoření seznamu

Seznam můžeme vytvořit snadno. Čárkami oddělené hodnoty uzavřeme do hranatých závorek.

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example']  [1]
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[0]                                        [2]
'a'
>>> a_list[4]                                        [3]
'example'
>>> a_list[-1]                                       [4]
'example'
>>> a_list[-3]                                       [5]
'mpilgrim'
```

[1] Nejdříve jsme nadefinovali seznam s pěti položkami. Všimněte si, že zachovávají své původní pořadí. Není to náhoda. Seznam je uspořádaná kolekce položek.

[2] Seznam můžeme používat jako pole s indexováním od nuly. První prvek každého neprázdného seznamu zpřístupníme vždy zápisem `a_list[0]`.

[3] Poslední prvek tohoto pětiprvkového seznamu je `a_list[4]`, protože indexování začíná nulou.

[4] Záporným indexem zpřístupňujeme položky ve směru od konce seznamu k začátku. Poslední prvek každého neprázdného seznamu zpřístupníme vždy zápisem `a_list[-1]`.

[5] Pokud se vám zdá použití záporného indexu matoucí, uvažujte o něm takto: `a_list[-n] == a_list[len(a_list) - n]`. Takže pro náš seznam pak platí `a_list[-3] == a_list[5 - 3] == a_list[2]`.

2.4.2. Vytváření podseznamů

a_list[0] je vždy první položkou seznamu a_list.

Jakmile máme vytvořen seznam, můžeme získat jakoukoliv jeho část. Anglicky se tomu říká „*slicing* the list“, což můžeme přeložit jako „*vykrajování* ze seznamu“ nebo „výřez ze seznamu“ nebo — z pohledu abstraktního záměru — vytváření podseznamu.

```
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[1:3]                          [1]
['b', 'mpilgrim']
>>> a_list[1:-1]                         [2]
['b', 'mpilgrim', 'z']
>>> a_list[0:3]                          [3]
['a', 'b', 'mpilgrim']
>>> a_list[:3]                           [4]
['a', 'b', 'mpilgrim']
>>> a_list[3:]                           [5]
['z', 'example']
>>> a_list[:]                            [6]
['a', 'b', 'mpilgrim', 'z', 'example']
```

[1] Část seznamu, výřez (slice), můžeme získat zadáním dvou indexů. Návratovou hodnotou je nový seznam, který obsahuje položky od prvního indexu výřezu (v tomto případě `a_list[1]`) až po položku (ale vyjma) s druhým indexem výřezu (v našem případě `a_list[3]`).

[2] Výřez funguje i v případě, kdy je hodnota jednoho nebo obou indexů výřezu záporná. Můžete si pomoci následujícím způsobem uvažování. Když se na seznam díváme zleva doprava, pak první index výřezu určuje první položku, kterou chceme, a druhý index výřezu určuje první položku, kterou nechceme. Vrací se vše mezi tím.

[3] Seznamy se indexují od nuly, takže zápis `a_list[0:3]` vrací první tři položky seznamu počínaje položkou `a_list[0]` až po `a_list[3]` vyjma (ta už se nevrací).

[4] Pokud je levý index výřezu roven nule, můžeme nulu vynechat a Python si ji tam dosadí. Takže zápis `a_list[:3]` vede ke stejnému výsledku jako `a_list[0:3]`, protože počáteční nula se dosadí jako implicitní hodnota.

[5] Podobně, pokud by pravý index výřezu měl mít hodnotu rovnou délce seznamu, můžeme jej vynechat. Protože náš seznam má pět položek, vede zápis `a_list[3:]` ke stejnému výsledku jako `a_list[3:5]`. A najdeme zde potěšitelnou symetrii. V našem pětiprvkovém seznamu vrací zápis `a_list[:3]` první tři položky a `a_list[3:]` vrací zbývající dvě. Obecně platí, že `a_list[:n]` vždy vrátí prvních n položek a `a_list[n:]` vrátí zbytek — nezávisle na délce seznamu.

[6] Pokud vynecháme oba indexy výřezu, jsou ve výsledku zahrnuty všechny položky původního seznamu. Ale není to totéž jako původní proměnná `a_list`. Jde o nový seznam, který má shodou okolností stejné položky. Zápis `a_list[:]` je tedy zkratkou pro získání úplné kopie seznamu.

2.4.3. Přidávání položek do seznamu

Položku můžeme do seznamu přidat čtyřmi způsoby.

```
>>> a_list = ['a']
>>> a_list = a_list + [2.0, 3]          [1]
>>> a_list                              [2]
['a', 2.0, 3]
>>> a_list.append(True)                 [3]
>>> a_list
['a', 2.0, 3, True]
>>> a_list.extend(['four', 'Ω'])        [4]
>>> a_list
['a', 2.0, 3, True, 'four', 'Ω']
>>> a_list.insert(0, 'Ω')               [5]
>>> a_list
['Ω', 'a', 2.0, 3, True, 'four', 'Ω']
```

[1] Operátor + spojí seznamy a vytvoří nový seznam. Seznam může obsahovat libovolný počet položek. Neexistuje zde žádný limit (pouze velikost dostupné paměti). Ale co se týká paměti, měli bychom si dát pozor na to, že spojením seznamů vzniká v paměti další seznam. V našem případě je nový seznam ihned přiřazen do existující proměnné `a_list`. Takže tento řádek kódu ve skutečnosti představuje dvoufázový proces — spojení (konkatenace) a přiřazení —, který může u rozsáhlých seznamů (dočasně) spotřebovat velké množství paměti.

[2] Seznam může obsahovat položky libovolného datového typu a v jednom seznamu nemusí být všechny položky stejného typu. V našem případě máme seznam obsahující řetězec, reálné číslo a celé číslo.

[3] Metoda `append()` přidává jednu položku na konec seznamu. (Teď už máme v seznamu položky se *čtyřmi* rozdílnými datovými typy!)

[4] Seznamy jsou implementovány formou třídy. „Vytvoření“ seznamu tedy znamená vytvoření instance třídy. V tomto smyslu mají seznamy metody, které nad nimi pracují. Metoda `extend()` přebírá jeden argument, kterým je seznam. Každý jeho prvek připojí na konec původního seznamu (append).

[5] Metoda `insert()` vloží do seznamu jednu položku. Prvním argumentem je index první položky seznamu, která bude z této pozice odsunuta. Položky seznamu nemusí být jedinečné. Například v našem případě teď seznam obsahuje dvě samostatné položky s hodnotou `'Ω'`: první položku (`a_list[0]`) a poslední položku (`a_list[6]`).

> Volání metody `a_list.insert(0, value)` se podobá použití funkce `unshift()` v jazyce Perl. Vloží prvek na začátek seznamu a index všech ostatních položek se zvýší, aby vzniklo potřebné místo.

Podívejme se podrobněji na rozdíly mezi `append()` a `extend()`.

```
>>> a_list = ['a', 'b', 'c']
>>> a_list.extend(['d', 'e', 'f'])       [1]
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(a_list)                          [2]
6
>>> a_list[-1]
'f'
>>> a_list.append(['g', 'h', 'i'])       [3]
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
>>> len(a_list)                          [4]
7
>>> a_list[-1]
['g', 'h', 'i']
```

[1] Metoda `extend()` přebírá jeden argument, kterým je vždy seznam, a přidá každý jeho prvek do seznamu `a_list`.

[2] Pokud začnete se seznamem o třech položkách a rozšíříte jej voláním `extend()` o seznam s dalšími třemi položkami, dostanete seznam s šesti položkami.

[3] Ve srovnání s tím metoda `append()` přebírá jeden argument, který může být libovolného datového typu. Na tomto řádku předáváme metodě `append()` seznam s třemi položkami.

[4] Pokud jsme začali se seznamem o šesti položkách a předaný seznam připojíme na konec, dostaneme seznam se sedmi položkami. Proč se sedmi? Protože poslední položkou (kterou jsme právě připojili) je *celý seznam*. Seznam může obsahovat data libovolného typu, včetně seznamu. Může to být právě to, co jste chtěli. Nebo možná nechtěli. Každopádně jste si o to řekli, a proto jste to dostali.

2.4.4. Vyhledávání hodnoty v seznamu

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list.count('new')                  [1]
2
>>> 'new' in a_list                      [2]
True
>>> 'c' in a_list
False
>>> a_list.index('mpilgrim')             [3]
3
>>> a_list.index('new')                  [4]
2
>>> a_list.index('c')                    [5]
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
```

[1] Metoda `count()` vrací počet výskytů určité hodnoty v seznamu (což se dalo čekat).

[2] Pokud se chcete dozvědět jen to, jestli nějaká hodnota v seznamu je nebo ne, pak je použití operátoru `in` o něco rychlejší než volání metody `count()`. Operátor `in` vždy vrací `True` nebo `False`. Neřekne vám, kolikrát se daná hodnota v seznamu vyskytuje.

[3] Ani operátor `in` ani metoda `count()` vám ale neřeknou, kde se v seznamu hodnota vyskytuje. Pokud chcete zjistit, *kde* se hodnota v seznamu nachází, použijte metodu `index()`. Pokud neřeknete jinak, bude prohledávat celý seznam. Ale nepovinným druhým argumentem můžete zadat index (od nuly), na kterém má hledání začít. A můžeme dokonce zadat nepovinný třetí argument s indexem místa, kde má hledání skončit.

[4] Metoda `index()` najde *první* výskyt zadané hodnoty v seznamu. V tomto případě se hodnota `'new'` vyskytuje v seznamu dvakrát: `a_list[2]` a `a_list[4]`. Ale metoda `index()` vrátí jen index prvního výskytu.

[5] Co byste ale možná *nečekali*, je to, že v případě nenalezení hodnoty v seznamu vyvolá metoda `index()` výjimku.

Počkat! Co? Je to tak. Pokud metoda `index()` nenajde v seznamu zadanou hodnotu, vyvolá výjimku. Jde o zjevně odlišné chování ve srovnání s jinými jazyky, které vracejí nějakou neplatnou hodnotu indexu (jako například `-1`). Ze začátku se vám to může zdát protivné, ale myslím, že to časem oceníte. Znamená to, že program zhavaruje v místě vzniku problému místo toho, aby potichu a divně selhal o chvíli později. Vzpomeňte si, že hodnota `-1` je platným indexem prvku v seznamu. Kdyby metoda `index()` místo výjimky vracela hodnotu `-1`, mohlo by to vést k poměrně nezábavným zážitkům při ladění.

2.4.5 Odstraňování položek ze seznamu

V seznamech nikdy nevznikají díry.

Seznamy se mohou automaticky nafukovat a smršťovat. Jejich expanzi už jsme si ukázali. Odstraňování položek ze seznamu můžeme také provést několika způsoby.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list[1]
'b'
>>> del a_list[1]                          [1]
>>> a_list
['a', 'new', 'mpilgrim', 'new']
>>> a_list[1]                              [2]
'new'
```

[1] Pro odstranění určené položky ze seznamu můžeme použít příkaz `del`.

[2] Pokud se pokoušíme o přístup k položce s indexem 1 poté, co jsme položku s indexem 1 odstranili, *nedojde* k chybě. Poziční index všech položek, které následují za rušenou položkou, bude posunut tak, aby byla vzniklá mezera zaplněna.

Že neznáte ten správný poziční index? Žádný problém. Odstranění položek můžete předepsat také jejich hodnotou.

```
>>> a_list.remove('new')                   [1]
>>> a_list
['a', 'mpilgrim', 'new']
>>> a_list.remove('new')                   [2]
>>> a_list
['a', 'mpilgrim']
>>> a_list.remove('new')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

[1] K odstranění položky ze seznamu můžete použít metodu `remove()`. Metoda `remove()` přebírá zadanou *hodnotu* a odstraní ze seznamu její první výskyt. A opět. Všechny položky, které následují za rušenou položkou, budou posunuty tak, aby byla vzniklá mezera zaplněna. V seznamech nikdy nevznikají díry.

[2] Metodu `remove()` můžete volat, kdykoliv se vám to hodí. Ale pokud se pokusíte o odstranění položky s hodnotou, která se v seznamu nevyskytuje, bude vyvolána výjimka.

2.4.6. Odstraňování položek ze seznamu: Bonusové kolo

Další zajímavou metodou seznamu je `pop()`. Metoda `pop()` představuje další způsob odstraňování položek ze seznamu, ale s malou fintou.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim']
>>> a_list.pop()                          [1]
'mpilgrim'
>>> a_list
['a', 'b', 'new']
>>> a_list.pop(1)                         [2]
'b'
>>> a_list
['a', 'new']
>>> a_list.pop()
'new'
>>> a_list.pop()
'a'
>>> a_list.pop()                          [3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

[1] Pokud voláme metodu `pop()` bez argumentů, odstraní poslední položku seznamu a *vrátí hodnotu*, která byla odstraněna.

[2] Metodou `pop()` můžeme ze seznamu odstranit libovolnou položku. Jednoduše jí předáme poziční index. Odstraní požadovanou položku, posune následující položky tak, aby zaplnila mezeru, a vrátí odstraněnou hodnotu.

[3] Pokud voláme `pop()` pro prázdný seznam, vznikne výjimka.

> Volání metody seznamu `pop()` bez argumentu se podobá volání funkce `pop()` v jazyce Perl. Odstraní poslední položku seznamu a vrátí hodnotu, která byla odstraněna. V jazyce Perl existuje také funkce `shift()`, která odstraní první položku a vrátí její hodnotu. Jde o ekvivalent pythonovského volání `a_list.pop(0)`.

2.4.7. Seznamy v booleovském kontextu

Prázdné seznamy se vyhodnocují jako false, ostatní seznamy jako true.

Seznam můžeme použít také v booleovském kontextu, jako například v příkazu `if`.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true([])             [1]
no, it's false
>>> is_it_true(['a'])          [2]
yes, it's true
>>> is_it_true([False])        [3]
yes, it's true
```

[1] Prázdný seznam se v booleovském kontextu vyhodnocuje jako false.
[2] Libovolný seznam, který obsahuje aspoň jednu položku, se vyhodnocuje jako true.
[3] Libovolný neprázdný seznam se vyhodnocuje jako true. Hodnota položek je nepodstatná.

2.5. N-tice

N-tice (anglicky tuple) se chová jako neměnitelný seznam. Jakmile je n-tice jednou vytvořena, nedá se nijak změnit.

```
>>> a_tuple = ("a", "b", "mpilgrim", "z", "example")  [1]
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple[0]                                        [2]
'a'
>>> a_tuple[-1]                                       [3]
'example'
>>> a_tuple[1:3]                                      [4]
('b', 'mpilgrim')
```

[1] N-tice se definuje stejným způsobem jako seznam. Jediný rozdíl spočívá v tom, že posloupnost prvků neuzavřeme do hranatých závorek, ale do kulatých.
[2] Prvky n-tice mají definované pořadí, stejně jako u seznamu. N-tice se indexují od nuly (jako seznam), takže první element neprázdné n-tice se zapisuje vždy `a_tuple[0]`.
[3] Záporné indexy se vyhodnocují od konce n-tice, stejně jako u seznamu.
[4] Dají se z nich získávat výřezy (slice), stejně jako u seznamů. Když získáte výřez se seznamu, má podobu nového seznamu. Když předepíšete výřez z n-tice, dostanete novou n-tici.

Hlavní rozdíl mezi n-ticemi a seznamy je ten, že n-tice nemohou být změněny. Z technického pohledu říkáme, že n-tice jsou neměnitelné (anglicky immutable). Prakticky se to projevuje tak, že neposkytu-

jí žádnou metodu, která by nám je dovolila změnit. Seznamy mají metody jako `append()`, `extend()`, `insert()`, `remove()` a `pop()`. N-tice žádnou z těchto metod nemají. Z n-tice můžeme vytvořit výřez (protože se vytváří nová n-tice), můžeme zjišťovat, zda n-tice obsahuje určitou hodnotu (protože tím ke změně n-tice nedochází) a... to je všechno.

```
# Pokračování předchozího příkladu

>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple.append("new")               [1]
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> a_tuple.remove("z")                 [2]
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> a_tuple.index("example")            [3]
4
>>> "z" in a_tuple                      [4]
True
```

[1] Do n-tice nemůžeme přidávat další prvky. N-tice nemají ani metodu `append()` ani `extend()`.

[2] Z n-tice nemůžeme prvky odstranit. N-tice nemají žádnou z metod `remove()` nebo `pop()`.

[3] V n-tici *můžeme* prvky vyhledávat, protože tím nedochází k její změně.

[4] Můžeme také použít operátor `in` pro testování, zda n-tice obsahuje zadaný prvek.

Takže na co jsou n-tice dobré?

- N-tice jsou rychlejší než seznamy. Pokud potřebujete nadefinovat konstantní sadu hodnot a vše, co s nimi budete kdy chtít dělat, bude jejich procházení, použijte místo seznamu n-tici.
- Pokud data nepotřebujete měnit a učiníte je „chráněnými proti zápisu“, bude váš kód bezpečnější. Pokud použijete místo seznamu n-tici, je to, jako kdybyste použili příkaz `assert`, který by kontroloval, zda jsou data konstantní. Překonat to můžeme jen záměrně (a s využitím specifické funkce).
- Některé n-tice mohou být použity jako slovníkové klíče (přesněji řečeno, n-tice, které obsahují *neměnitelné* (immutable) hodnoty jako jsou řetězce, čísla a jiné n-tice). V roli slovníkových klíčů nemůžou nikdy vystupovat seznamy, protože seznamy nejsou neměnitelné (immutable).

> N-tice mohou být převedeny na seznamy a naopak. Zabudovaná funkce `tuple()` může převzít seznam a vrací n-tici se stejnými prvky. A naopak funkce `list()` může převzít zadanou n-tici a vrací seznam. Z pohledu účinku tedy funkce `tuple()` seznam zmrazí a funkce naopak `list()` rozpustí n-tici.

2.5.1. N-tice v booleovském kontextu

N-tice můžeme použít v booleovském kontextu, jako například v příkazu `if`.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(())              [1]
no, it's false
>>> is_it_true(('a', 'b'))      [2]
yes, it's true
>>> is_it_true((False,))        [3]
yes, it's true
>>> type((False))               [4]
<class 'bool'>
>>> type((False,))
<class 'tuple'>
```

[1] Prázdná n-tice se v booleovském kontextu vyhodnocuje jako false.

[2] Libovolná n-tice s alespoň jednou položkou se vyhodnocuje jako true.

[3] Libovolná n-tice s alespoň jednou položkou se vyhodnocuje jako true. Hodnota položek je nepodstatná. Ale co tady dělá ta čárka?

[4] Pokud chceme vytvořit n-tici s jedinou položkou, pak musíme za hodnotu připsat čárku. Pokud bychom čárku nepřidali, Python by si myslel, že jsme jednoduše přidali nadbytečnou dvojici závorek. Je to sice neškodné, ale n-tice se tím nevytvoří.

2.5.2. Přiřazení více hodnot najednou

Následuje parádní programátorská zkratka. V Pythonu můžete n-tici použít pro přiřazení více hodnot najednou.

```
>>> v = ('a', 2, True)
>>> (x, y, z) = v               [1]
>>> x
'a'
>>> y
2
>>> z
True
```

[1] v je n-tice o třech prvcích a (x, y, z) je n-tice s třemi proměnnými. Přiřazení jedné do druhé vede k přiřazení každé z hodnot n-tice v do jednotlivých proměnných v uvedeném pořadí.

Využít se toho dá všemožnými způsoby. Dejme tomu, že chcete pojmenovat řadu hodnot. K rychlému přiřazení po sobě jdoucích hodnot můžete využít zabudovanou funkci range() a vícenásobné přiřazení.

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7)  [1]
>>> MONDAY                                                                       [2]
0
>>> TUESDAY
1
>>> SUNDAY
6
```

[1] Zabudovaná funkce range() vytváří posloupnost celých čísel. (Z technického hlediska nevrací funkce range() seznam ani n-tici, ale **iterátor**. Odlišnosti se naučíme později.) MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, a SUNDAY jsou proměnné, které definujeme. (Tento příklad pochází z modulu calendar, což je malý zábavný modul, který tiskne kalendář podobně jako unixový program cal. Modul calendar definuje pro dny v týdnu celočíselné konstanty.)

[2] V tomto okamžiku má každá z proměnných svou hodnotu: Proměnná MONDAY je rovna 0, TUESDAY má hodnotu 1 a tak dále.

Současného přiřazení více proměnným můžeme využít také pro vytváření funkcí, které vracejí více hodnot najednou. Jednoduše v nich vrátíme n-tici se všemi požadovanými hodnotami. Ve volajícím kódu se k výsledku můžeme chovat jako k jedné n-tici, nebo jej můžeme přiřadit do více jednotlivých proměnných. Tento obrat používá řada standardních pythonovských knihoven, včetně modulu os. O něm si něco řekneme v následující kapitole.

2.6. Množiny

Množina (set) je neuspořádanou kolekcí jedinečných hodnot. Jedna množina může obsahovat hodnoty libovolného neměnitelného (immutable) datového typu. Pokud máme k dispozici dvě množiny, můžeme s nimi provádět standardní množinové operace, jako je sjednocení, průnik a rozdíl množin.

2.6.1. Vytvoření množiny

Ale nejdříve proberme základy. Množinu vytvoříme snadno.

```
>>> a_set = {1}      [1]
>>> a_set
{1}
>>> type(a_set)      [2]
<class 'set'>
>>> a_set = {1, 2}   [3]
>>> a_set
{1, 2}
```

[1] Pokud chceme vytvořit množinu s jednou hodnotou, uzavřeme hodnotu do složených závorek ({}).

[2] Množiny jsou ve skutečnosti implementovány jako třídy, ale tím se teď nebudeme zatěžovat.

[3] Pokud chceme vytvořit množinu s více hodnotami, oddělíme hodnoty čárkami a vše uzavřeme do složených závorek.

Množinu můžeme vytvořit i ze seznamu.

```
>>> a_list = ['a', 'b', 'mpilgrim', True, False, 42]
>>> a_set = set(a_list)                                  [1]
>>> a_set                                                [2]
{'a', False, 'b', True, 'mpilgrim', 42}
>>> a_list                                               [3]
['a', 'b', 'mpilgrim', True, False, 42]
```

[1] K vytvoření množiny ze seznamu použijeme funkce `set()`. (Puntičkáři, kteří vědí, jak jsou množiny implementovány, by zde podotkli, že ve skutečnosti nejde o volání funkce, ale o vytváření instance třídy. Já vám *slibuji*, že se o tomto rozdílu dozvíte v této knize později. Prozatím nám bude stačit vědět, že `set()` se chová jako funkce a že vrací množinu.)

[2] Jak už jsem se zmínil dříve, jedna množina může obsahovat hodnoty libovolného datového typu. A zmínil jsem se také, že množiny jsou *neuspořádané*. Tato množina si nepamatuje původní pořadí prvků v seznamu, který byl použit k jejímu vytvoření. Pokud byste do množiny přidávali další prvky, nebude si množina pamatovat pořadí, v jakém jste je vkládali.

[3] Původní seznam zůstává nezměněn.

Že zatím nemáte k dispozici žádné hodnoty? Žádný problém. Můžeme vytvořit prázdnou množinu.

```
>>> a_set = set()    [1]
>>> a_set            [2]
set()
>>> type(a_set)      [3]
<class 'set'>
>>> len(a_set)       [4]
0
>>> not_sure = {}    [5]
```

```
>>> type(not_sure)
<class 'dict'>
```

[1] K vytvoření prázdné množiny zavoláme `set()` bez argumentů.
[2] Zobrazená reprezentace prázdné množiny vypadá trochu divně. Očekávali jste spíš něco jako `{}`? Tímto způsobem se vyjadřuje prázdný slovník a ne množina. O slovnících se dozvíme později, ale ještě v této kapitole.
[3] Navzdory podivnosti zobrazené reprezentace to skutečně *je* množina...
[4] ...a tato množina neobsahuje žádné prvky.
[5] Prázdnou množinu nelze vytvořit zápisem dvou složených závorek kvůli historickým způsobům přeneseným z Pythonu 2. Tímto způsobem se vyjadřuje prázdný slovník a ne množina.

2.6.2. Úprava množiny

Do existující množiny můžeme přidávat hodnoty dvěma různými způsoby: metodou `add()` a metodou `update()`.

```
>>> a_set = {1, 2}
>>> a_set.add(4)          [1]
>>> a_set
{1, 2, 4}
>>> len(a_set)            [2]
3
>>> a_set.add(1)          [3]
>>> a_set
{1, 2, 4}
>>> len(a_set)            [4]
3
```

[1] Metoda `add()` přebírá jeden argument, který může být libovolného datového typu, a přidává zadanou hodnotu do množiny.
[2] Množina teď má tři členy.
[3] Množiny jsou kolekcemi jedinečných hodnot. Pokud do množiny zkusíme přidat hodnotu, která se v ní již nachází, neudělá to nic. Nevznikne chyba. Jde zkrátka o prázdnou operaci.
[4] Množina má *pořád* jen tři členy.

```
>>> a_set = {1, 2, 3}
>>> a_set
{1, 2, 3}
>>> a_set.update({2, 4, 6})          [1]
>>> a_set                            [2]
{1, 2, 3, 4, 6}
```

```
>>> a_set.update({3, 6, 9}, {1, 2, 3, 5, 8, 13})  [3]
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 13}
>>> a_set.update([10, 20, 30])                    [4]
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}
```

[1] Metoda `update()` přebírá jeden argument, rovněž množinu, a přidá všechny její členy do původní množiny. Je to, jako kdybychom volali metodu `add()` pro všechny členy množiny předané argumentem.

[2] Protože cílová množina nemůže obsahovat jednu hodnotu dvakrát, duplicitní hodnoty se ignorují.

[3] Ve skutečnosti můžete metodu `update()` volat s libovolným počtem argumentů. Pokud ji zavoláte s dvěma množinami, metoda `update()` přidá všechny členy z každé z předaných množin do původní množiny (duplicitní hodnoty se přeskočí).

[4] Metoda `update()` umí zpracovat objekty různých datových typů, včetně seznamů. Pokud jí předáte seznam, pak metoda `update()` přidá do původní množiny všechny členy seznamu.

2.6.3. Odstraňování položek z množiny

Jednotlivé hodnoty lze z množiny odstranit třemi způsoby. První dva, `discard()` a `remove()`, se liší v jedné malé drobnosti.

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set
{1, 3, 36, 6, 10, 45, 15, 21, 28}
>>> a_set.discard(10)                             [1]
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.discard(10)                             [2]
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.remove(21)                              [3]
>>> a_set
{1, 3, 36, 6, 45, 15, 28}
>>> a_set.remove(21)                              [4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 21
```

[1] Metoda `discard()` přebírá jeden argument a zadanou hodnotu odebere z množiny.

[2] Pokud metodu `discard()` voláme s hodnotou, která v množině neexistuje, neprovede se nic. Nevznikne chyba. Jde o prázdnou operaci.

[3] Metoda `remove()` také přebírá hodnotu jediného argumentu a také odstraňuje hodnotu z množiny.

[4] Odlišnost se projeví v případě, kdy se zadaná hodnota v množině nenachází. V takovém případě metoda `remove()` vyvolá výjimku `KeyError`.

Množiny, stejně jako seznamy, podporují metodu `pop()`.

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set.pop()                                    [1]
1
>>> a_set.pop()
3
>>> a_set.pop()
36
>>> a_set
{6, 10, 45, 15, 21, 28}
>>> a_set.clear()                                  [2]
>>> a_set
set()
>>> a_set.pop()                                    [3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

[1] Metoda `pop()` odstraní jeden prvek z množiny a vrátí jeho hodnotu. Ale množiny jsou neuspořádané a neexistuje u nich nic takového jako „poslední“ hodnota. Proto také neexistuje možnost ovlivnit, která hodnota bude odstraněna. Je to zcela náhodné.

[2] Metoda `clear()` *odstraní* všechny prvky množiny a množina se stane prázdnou. Ve výsledku je to stejné jako provedení příkazu `a_set = set()`, který vytvoří novou prázdnou množinu a přepíše původní hodnotu proměnné `a_set`.

[3] Pokus o volání metody `pop()` pro prázdnou množinu vede k vyvolání výjimky `KeyError`.

2.6.4. Běžné množinové operace

Pythonovský datový typ `set` podporuje několik běžných množinových operací.

```
>>> a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
>>> 30 in a_set                                                    [1]
True
>>> 31 in a_set
False
>>> b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
```

```
>>> a_set.union(b_set)                                          [2]
{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}
>>> a_set.intersection(b_set)                                   [3]
{9, 2, 12, 5, 21}
>>> a_set.difference(b_set)                                     [4]
{195, 4, 76, 51, 30, 127}
>>> a_set.symmetric_difference(b_set)                           [5]
{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
```

[1] Pokud chceme otestovat, zda je daná hodnota prvkem množiny, použijeme operátor `in`. Funguje stejným způsobem jako u seznamů.

[2] Metoda `union()` (sjednocení) vrací novou množinu, která obsahuje všechny prvky jak z jedné, tak z druhé množiny.

[3] Metoda `intersection()` (průnik) vrací novou množinu, která obsahuje všechny prvky nacházející se v obou množinách současně.

[4] Metoda `difference()` (rozdíl) vrací novou množinu obsahující všechny prvky, které se nacházejí v množině `a_set`, ale nenacházejí se v množině `b_set`.

[5] Metoda `symmetric_difference()` (symetrický rozdíl) vrací novou množinu obsahující všechny prvky, které se nacházejí právě v jedné z množin.

Tři z těchto metod jsou symetrické.

```
# Pokračování předchozího příkladu

>>> b_set.symmetric_difference(a_set)                                    [1]
{3, 1, 195, 4, 6, 8, 76, 15, 17, 18, 51, 30, 127}
>>> b_set.symmetric_difference(a_set) == a_set.symmetric_difference(b_set)  [2]
True
>>> b_set.union(a_set) == a_set.union(b_set)                             [3]
True
>>> b_set.intersection(a_set) == a_set.intersection(b_set)               [4]
True
>>> b_set.difference(a_set) == a_set.difference(b_set)                   [5]
False
```

[1] Symetrický rozdíl množin `a_set` od `b_set` *vypadá* jinak než symetrický rozdíl množin `b_set` od `a_set`. Ale pamatujte na to, že množiny jsou neuspořádané. Jakékoliv dvě množiny, jejichž všechny hodnoty se shodují (žádná nesmí být vynechána), se považují za shodné.

[2] A přesně tento případ nastal zde. Nenechte se zmást reprezentacemi těchto množin zobrazenými pythonovským shellem. Obsahují stejné hodnoty, takže jsou shodné.

[3] Sjednocení dvou množin je také symetrické.

[4] Průnik dvou množin je rovněž symetrický.

[5] Rozdíl dvou množin symetrický není. Ono to dává smysl. Podobá se to odčítání jednoho čísla od druhého. Na pořadí operandů zde záleží.

A nakonec tu máme několik otázek, které můžeme množinám položit.

```
>>> a_set = {1, 2, 3}
>>> b_set = {1, 2, 3, 4}
>>> a_set.issubset(b_set)                [1]
True
>>> b_set.issuperset(a_set)              [2]
True
>>> a_set.add(5)                         [3]
>>> a_set.issubset(b_set)
False
>>> b_set.issuperset(a_set)
False
```

[1] Množina `a_set` je podmnožinou množiny `b_set` — všechny prvky množiny `a_set` jsou současně prvky množiny `b_set`.

[2] Stejnou otázku můžeme položit obráceně. Množina b_set je nadmnožinou množiny a_set, protože všechny prvky množiny `a_set` jsou současně prvky množiny `b_set`.

[3] Jakmile do množiny `a_set` přidáme hodnotu, která se v množině `b_set` nenachází, oba testy vrátí hodnotu `False`.

2.6.5. Množiny v booleovském kontextu

Množiny můžeme použít v booleovském kontextu, například v příkazu `if`.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(set())                    [1]
no, it's false
>>> is_it_true({'a'})                    [2]
yes, it's true
>>> is_it_true({False})                  [3]
yes, it's true
```

[1] Prázdná množina se v booleovském kontextu vyhodnocuje jako false.

[2] Libovolná množina s alespoň jedním prvkem se vyhodnocuje jako true.

[3] Libovolná množina s alespoň jedním prvkem se vyhodnocuje jako true. Hodnota prvků je nepodstatná.

2.7. Slovníky

Slovník (dictionary) je neuspořádaná kolekce dvojic klíč-hodnota. Když do slovníku přidáme klíč, musíme do něj současně přidat i hodnotu, která ke klíči patří. (Hodnotu můžeme v budoucnu kdykoliv změnit.) Pythonovské slovníky jsou optimalizované pro získávání hodnoty k zadanému klíči, ale ne naopak.

> Pythonovský slovník se chová jako hash (čti [heš]; vyhledávací tabulka) v Perl 5. V jazyce Perl 5 začínají proměnné typu hash vždy znakem %. Pythonovské proměnné mohou být pojmenovány zcela libovolně. Python si vnitřně eviduje jejich datový typ.

2.7.1. Vytvoření slovníku

Slovník vytvoříme snadno. Syntaxe se podobá množinové, ale místo pouhé hodnoty zadáváme dvojice klíč-hodnota. Jakmile slovník existuje, můžeme v něm vyhledávat hodnoty podle jejich klíče.

```
>>> a_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'}  [1]
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['server']                                                    [2]
'db.diveintopython3.org'
>>> a_dict['database']                                                  [3]
'mysql'
>>> a_dict['db.diveintopython3.org']                                    [4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'db.diveintopython3.org'
```

[1] Nejdříve vytvoříme slovník s dvěma položkami a přiřadíme ho do proměnné `a_dict`. Každá položka je tvořena dvojicí klíč-hodnota a celý výčet položek je uzavřen ve složených závorkách.

[2] Řetězec `'server'` je zde klíčem a k němu přidruženou hodnotou, na kterou se odkážeme zápisem `a_dict['server']`, je `'db.diveintopython3.org'`.

[3] Řetězec `'database'` je zde klíčem. K němu přidruženou hodnotou, na kterou se odkážeme zápisem `a_dict['database'], je 'mysql'`.

[4] Hodnoty můžeme získat na základě klíče, ale klíče nemůžeme získat na základě znalosti hodnoty. Takže `a_dict['server']` obsahuje `'db.diveintopython3.org'`, ale `a_dict['db.diveintopython3.org']` vyvolá výjimku, protože `'db.diveintopython3.org'` není klíčem.

2.7.2. Úprava slovníku

Slovníky nemají žádné předem určené omezení velikosti. Dvojici klíč-hodnota můžeme do slovníku přidat kdykoliv. Nebo můžeme měnit hodnotu příslušející ke klíči. Pokračování předchozího příkladu:

```
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['database'] = 'blog'          [1]
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'blog'}
>>> a_dict['user'] = 'mark'              [2]
>>> a_dict                               [3]
{'server': 'db.diveintopython3.org', 'user': 'mark', 'database': 'blog'}
>>> a_dict['user'] = 'dora'              [4]
>>> a_dict
{'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
>>> a_dict['User'] = 'mark'              [5]
>>> a_dict
{'User': 'mark', 'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
```

[1] Ve slovníku se nemohou nacházet duplicitní klíče. Pokud přiřadíme hodnotu k existujícímu klíči, dojde k přepsání původní hodnoty.

[2] Dvojici klíč-hodnota můžeme přidat kdykoliv. Tato syntaxe se shoduje s případem změny existujících hodnot.

[3] Nová položka slovníku (klíč `'user'`, hodnota `'mark'`) se objevila uprostřed. To, že se u prvního příkladu položky objevily seřazené, byla pouhá náhoda. Stejná náhoda je to, že se nyní jeví jako rozházené.

[4] Přiřazení hodnoty k existujícímu klíči slovníku vede k prosté náhradě staré hodnoty novou.

[5] Změní se tímto příkazem hodnota klíče `user` zpět na „mark“? Nikoliv! Prohlédněte si klíč pořádně. V řetězci `User` je napsáno velké `U`. Klíče slovníků jsou citlivé na velikost písmen, takže tento příkaz vytváří novou dvojici klíč-hodnota a existující hodnotu nepřepíše. Klíč se vám sice může zdát podobný, ale z pohledu Pythonu je úplně jiný.

2.7.3. Slovníky se smíšeným obsahem

Slovníky nejsou určeny jen pro řetězce. Hodnoty ve slovníku mohou být libovolného datového typu včetně celých čísel, booleovských hodnot, libovolných objektů nebo dokonce slovníků. Uvnitř jednoho slovníku nemusí být všechny hodnoty stejného typu. Můžeme je míchat podle potřeby. Klíče slovníků mají větší omezení, ale mohou být typu řetězec, celé číslo a několika dalších typů. Datové typy klíčů v jednom slovníku můžeme také míchat.

Se slovníky s neřetězcovými klíči a hodnotami jsme se vlastně už setkali v kapitole Váš první pythonovský program.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

Teď to v interaktivním shellu rozkucháme.

```
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
...              1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>> len(SUFFIXES)                          [1]
2
>>> 1000 in SUFFIXES                       [2]
True
>>> SUFFIXES[1000]                         [3]
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> SUFFIXES[1024]                         [4]
['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
>>> SUFFIXES[1000][3]                      [5]
'TB'
```

[1] Funkce `len()`, podobně jako u seznamů a množin, vrací počet klíčů ve slovníku.

[2] A stejně jako u seznamů a množin můžeme použít operátor `in` k testování, zda je zadaný klíč ve slovníku definován.

[4] Číslo `1000` *je* klíčem ve slovníku `SUFFIXES`. Jeho hodnotou je seznam osmi položek (osmi řetězců, abychom byli přesní).

[5] A podobně i číslo `1024` je klíčem ve slovníku `SUFFIXES`. Jeho hodnotou je také seznam s osmi položkami.

[6] A protože `SUFFIXES[1000]` obsahuje seznam, můžeme jeho jednotlivé prvky zpřístupňovat prostřednictvím indexu (od nuly).

2.7.4. Slovníky v booleovském kontextu

Slovník můžeme použít v booleovském kontextu, jako například v příkazu `if`.

Prázdné slovníky se vyhodnocují jako false, všechny ostatní slovníky jako true.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true({})             [1]
no, it's false
>>> is_it_true({'a': 1})       [2]
yes, it's true
```

[1] Prázdný slovník se v booleovském kontextu vyhodnocuje jako false.
[2] Slovník s alespoň jednou dvojicí klíč-hodnota se vyhodnocuje jako true.

2.8. None

None [nan] je speciální pythonovskou konstantou. Vyjadřuje žádnou hodnotu. Ale None není totéž co `False`. None není nula. None není prázdný řetězec. Pokud porovnáme `None` s čímkoliv jiným než s `None`, vždycky dostaneme `False`.

`None` je jedinou „žádnou“ hodnotou. Má svůj vlastní datový typ (`NoneType`). Hodnotu `None` můžeme přiřadit do libovolné proměnné, ale nemůžeme vytvořit jiný objekt typu `NoneType`. Všechny proměnné, jejichž hodnota je `None`, jsou vzájemně shodné.

```
>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ''
False
>>> None == None
True
>>> x = None
>>> x == None
True
>>> y = None
>>> x == y
True
```

2.8.1. None v booleovském kontextu

V booleovském kontextu se None vyhodnocuje jako false a not None jako true.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(None)
no, it's false
>>> is_it_true(not None)
yes, it's true
```

2.9. Přečtěte si

- **Boolean operations** (booleovské operace)
- **Numeric types** (číselné typy)
- **Sequence types** (typy posloupností)
- **Set types** (množinové typy)
- **Mapping types** (mapovací typy, vyhledávací tabulky)
  (vše na http://docs.python.org/py3k/library/stdtypes.html)
- **modul `fractions`** (zlomky)
  (http://docs.python.org/py3k/library/fractions.html)
- **modul `math`** (matematický)
  (http://docs.python.org/py3k/library/math.html)
- **PEP 237: Unifying Long Integers and Integers** (sjednocení velkých celých čísel a celých čísel)
  (http://www.python.org/dev/peps/pep-0237/)
- **PEP 238: Changing the Division Operator** (změna operátoru dělení)
  (http://www.python.org/dev/peps/pep-0238/)

> **“ Our imagination is stretched to the utmost, not, as in fiction, to imagine things which are not really there, but just to comprehend those things which are. ”**
> (Naše představivost je napjatá do krajnosti. Ne jako u fikce, abychom si představili věci, které zde nejsou, ale proto, abychom jen obsáhli věci, které jsou zde.)
>
> — Richard Feynman

3. Generátorová notace

3. Generátorová notace — 91
3.1. Ponořme se — **93**
3.2. Práce se soubory a s adresáři — **93**
3.2.1. Aktuální pracovní adresář — **93**
3.2.2. Práce se jmény souborů a adresářů — **94**
3.2.3. Výpis adresářů — **96**
3.2.4. Získání dalších informací o souboru — **97**
3.2.5. Jak vytvořit absolutní cesty — **98**
3.3. Generátorová notace seznamu — **98**
3.4. Generátorová notace slovníku — **100**
3.4.1. Další legrácky s generátorovou notací slovníků — **102**
3.5. Generátorová notace množin — **103**
3.6. Přečtěte si — **103**

3.1. Ponořme se
3.2. Práce se soubory a s adresáři

3.1. Ponořme se

V každém programovacím jazyce najdeme určitý rys, který záměrně zjednodušuje nějakou komplikovanou věc. Pokud přicházíte se zkušenostmi z jiného jazyka, můžete to snadno přehlédnout, protože váš starý jazyk právě tu určitou věc nezjednodušoval (protože dalo práci místo toho zjednodušit něco jiného). V této kapitole se seznámíme s generátorovou notací seznamů (list comprehensions), s generátorovou notací slovníků (dictionary comprehensions) a s generátorovou notací množin (set comprehensions). Jde o tři související koncepty, jejichž jádrem je jedna velmi mocná technika. Ale nejdříve si uděláme malou odbočku ke dvěma modulům, které vám usnadní orientaci ve vašem lokálním souborovém systému.

3.2. Práce se soubory a s adresáři

Python 3 se dodává s modulem zvaným `os`, což je zkratka pro „operační systém“. Modul os obsahuje spoustu funkcí pro získávání informací o lokálních adresářích, souborech, procesech a proměnných prostředí — a v některých případech s nimi umožňuje manipulovat. Python se snaží co nejlépe, aby pro všechny podporované operační systémy nabízel jednotné `API` (aplikační programové rozhraní). Cílem je, aby vaše programy běžely na libovolném počítači a aby přitom obsahovaly co nejméně kódu, který by byl závislý na platformě.

3.2.1. Aktuální pracovní adresář

Pokud s Pythonem právě začínáte, strávíte ještě hodně času v pythonovském shellu. V celé knize se budete setkávat s příklady, jako je tento:

1. Importujte jeden z modulů nacházejících se v adresáři `examples` (příklady).
2. Zavolejte funkci z tohoto modulu.
3. Vysvětlete výsledky.

Vždy existuje to, čemu se říká aktuální pracovní adresář.

Pokud o aktuálním pracovním adresáři nic nevíte, pak krok 1 pravděpodobně selže a objeví se výjimka `ImportError`. Proč? Protože Python se bude po modulu dívat ve vyhledávací cestě pro import, ale nenajde jej, protože adresář `examples` se v žádném adresáři z vyhledávací cesty nenachází. Aby to prošlo, můžete udělat jednu ze dvou věcí:

1. Adresář `examples` přidáte do vyhledávací cesty pro import.
2. Změníte aktuální pracovní adresář na `examples`.

Aktuální pracovní adresář je neviditelný údaj, který si Python neustále udržuje v paměti. Aktuální pracovní adresář existuje vždy — ať už jste v pythonovském shellu, spouštíte svůj vlastní pythonovský skript z příkazového řádku nebo spouštíte pythonovský CGI skript na nějakém webovém serveru.

Pro vypořádání se s aktuálním pracovním adresářem nabízí modul os dvě funkce.

```
>>> import os                                                [1]
>>> print(os.getcwd())                                      [2]
C:\Python31
>>> os.chdir('/Users/pilgrim/diveintopython3/examples')     [3]
>>> print(os.getcwd())                                      [4]
C:\Users\pilgrim\diveintopython3\examples
```

[1] Modul os je součástí Pythonu. Můžete jej importovat kdykoliv a kdekoliv.

[2] Informaci o aktuálním pracovním adresáři získáte použitím funkce os.getcwd(). Pokud používáte grafický pythonovský shell, pak se aktuální pracovní adresář zpočátku nachází v adresáři, ve kterém je umístěn spustitelný program pythonovského shellu. Při práci pod Windows to záleží na tom, kam jste Python nainstalovali. Výchozí adresář je c:\Python31. Pokud používáte konzolový pythonovský shell, pak se aktuální pracovní adresář zpočátku nachází v adresáři, ve kterém jste spustili python3.

[3] Aktuální pracovní adresář můžeme měnit použitím funkce os.chdir().

[4] Při volání funkce os.chdir() jsem použil cestu v linuxovém stylu (normální lomítka, žádné písmeno disku), i když pracuji pod Windows. To je právě jedno z míst, kde se Python snaží zamaskovat rozdíly mezi operačními systémy.

3.2.2. Práce se jmény souborů a adresářů

Když už se bavíme o adresářích, chtěl bych vás upozornit na modul os.path. Ten obsahuje funkce pro manipulace se jmény souborů a adresářů.

```
>>> import os
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples/', 'humansize.py'))       [1]
/Users/pilgrim/diveintopython3/examples/humansize.py
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples', 'humansize.py'))        [2]
/Users/pilgrim/diveintopython3/examples\humansize.py
>>> print(os.path.expanduser('~'))                                                        [3]
c:\Users\pilgrim
>>> print(os.path.join(os.path.expanduser('~'), 'diveintopython3', 'examples', 'humansize.py')) [4]
c:\Users\pilgrim\diveintopython3\examples\humansize.py
```

[1] Funkce os.path.join() sestaví cestu z jedné nebo více částí cesty. V tomto případě jednoduše spojí řetězce.

[2] Tento příklad už není tak jednoduchý. Funkce `os.path.join()` před napojením jména souboru navíc přidá k cestě jedno lomítko. Místo obyčejného lomítka použila zpětné lomítko, protože jsem tento příklad pouštěl pod Windows. Pokud byste stejný příklad zkoušeli na systémech Linux nebo Mac OS X, použilo by se normální lomítko. Nepárejte se s lomítky. Používejte vždy `os.path.join()` a nechejte na Pythonu, aby udělal, co je správné.

[3] Funkce `os.path.expanduser()` rozepíše cestu, která pro vyjádření domácího adresáře aktuálního uživatele používá znak ~. Funguje to na libovolné platformě, kde mají uživatelé přidělený svůj domácí adresář, tedy na Linuxu, Mac OS X a ve Windows. Vrácená cesta neobsahuje koncové lomítko, ale to funkci `os.path.join()` nevadí.

[4] Kombinováním těchto technik můžeme snadno konstruovat cesty do adresářů a k souborům, které se nacházejí v uživatelově domácím adresáři. Funkce `os.path.join()` přebírá libovolný počet argumentů. Jakmile jsem to zjistil, skákal jsem radostí, protože při přípravě mých nástrojů v nějakém novém jazyce je `addSlashIfNecessary()` (přidejLomítkoPokudJeToNutné) jednou z těch otravných malých funkcí, které si musím vždy znovu napsat. V Pythonu takovou funkci *nepište*. Chytří lidé už se o to postarali za vás.

Modul `os.path` obsahuje také funkce, které umí rozdělit plné cesty, jména adresářů a souborů na jejich podstatné části.

```
>>> pathname = '/Users/pilgrim/diveintopython3/examples/humansize.py'
>>> os.path.split(pathname)                                          [1]
('/Users/pilgrim/diveintopython3/examples', 'humansize.py')
>>> (dirname, filename) = os.path.split(pathname)                    [2]
>>> dirname                                                          [3]
'/Users/pilgrim/diveintopython3/examples'
>>> filename                                                         [4]
'humansize.py'
>>> (shortname, extension) = os.path.splitext(filename)              [5]
>>> shortname
'humansize'
>>> extension
'.py'
```

[1] Funkce `split` rozdělí plnou cestu a vrátí n-tici, která obsahuje zvlášť cestu a zvlášť jméno souboru.

[2] Pamatujete si, že jsme se bavili o možnosti vracet více hodnot z funkce přiřazením hodnot více proměnným najednou? Funkce `os.path.split()` dělá přesně tohle. Výsledek funkce `split` přiřadíme do n-tice s dvěma proměnnými. Každá z proměnných získá hodnotu odpovídajícího prvku vracené dvojice.

[3] První proměnná, dirname, obdrží hodnotu prvního prvku n-tice, kterou vrací funkce os.path.`split()`, a sice cestu k souboru.

[4] Druhá proměnná, `filename`, obdrží hodnotu druhého prvku n-tice vracené funkcí `os.path.split()`, jméno souboru.

[5] Modul `os.path` obsahuje také funkci `os.path.splitext()`, která rozdělí jméno souboru a vrací dvojici obsahující jméno souboru bez přípony a příponu. Pro jejich přiřazení do oddělených proměnných použijeme stejnou techniku.

3.2.3. Výpis adresářů

Modul `glob` používá shellovské zástupné znaky.

Dalším nástrojem z pythonovské standardní knihovny je modul `glob`. Umožní nám z programu snadno získat obsah nějakého adresáře. Používá typ zástupných znaků (wildcards), které už asi znáte z práce na příkazovém řádku.

```
>>> os.chdir('/Users/pilgrim/diveintopython3/')
>>> import glob
>>> glob.glob('examples/*.xml')                    [1]
['examples\\feed-broken.xml',
 'examples\\feed-ns0.xml',
 'examples\\feed.xml']
>>> os.chdir('examples/')                          [2]
>>> glob.glob('*test*.py')                         [3]
['alphameticstest.py',
 'pluraltest1.py',
 'pluraltest2.py',
 'pluraltest3.py',
 'pluraltest4.py',
 'pluraltest5.py',
 'pluraltest6.py',
 'romantest1.py',
 'romantest10.py',
 'romantest2.py',
 'romantest3.py',
 'romantest4.py',
 'romantest5.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

[1] Modul `glob` zpracovává masku se zástupným znakem a vrací cesty ke všem souborům a adresářům, které masce se zástupným znakem odpovídají. V tomto příkladu je maska složena z cesty do adresáře a z „`*.xml`“. Budou jí odpovídat všechny `.xml` soubory v podadresáři examples.

[2] Teď jako aktuální pracovní adresář zvolíme podadresář examples. Funkce `os.chdir()` umí pracovat i s relativními cestami.

[3] Ve vzorku pro funkci `glob` můžeme použít více zástupných znaků. Tento příklad nalezne v aktuálním pracovním adresáři všechny soubory, které končí příponou `.py` a kdekoliv ve jméně souboru obsahují slovo `test`.

3.2.4. Získání dalších informací o souboru

Každý moderní souborový systém ukládá o každém souboru metadata, jako jsou: datum vytvoření, datum poslední modifikace, velikost souboru atd. Pro zpřístupnění těchto metadat poskytuje Python jednotné `API`. Soubor se nemusí otevírat. Vše, co potřebujete znát, je jeho jméno.

```
>>> import os
>>> print(os.getcwd())                          [1]
c:\Users\pilgrim\diveintopython3\examples
>>> metadata = os.stat('feed.xml')              [2]
>>> metadata.st_mtime                           [3]
1247520344.9537716
>>> import time                                 [4]
>>> time.localtime(metadata.st_mtime)           [5]
time.struct_time(tm_year=2009, tm_mon=7, tm_mday=13, tm_hour=17,
  tm_min=25, tm_sec=44, tm_wday=0, tm_yday=194, tm_isdst=1)
```

[1] Aktuálním pracovním adresářem je složka `examples`.

[2] `feed.xml` je soubor ve složce examples. Voláním funkce `os.stat()` získáme objekt, který obsahuje několik různých typů informací o souboru (metadat).

[3] `st_mtime` zachycuje čas poslední modifikace, ale není uložen ve tvaru, který by byl moc použitelný. (Z technického pohledu je to počet sekund od Epochy, kde Epocha je definována jako první sekunda 1. ledna 1970. Vážně!)

[4] Modul `time` je součástí standardní pythonovské knihovny. Obsahuje funkce pro převody mezi různými reprezentacemi času, pro formátování času do řetězcové podoby a pro hraní si s časovými zónami.

[5] Funkce `time.localtime()` převádí hodnotu času ze sekund-od-Epochy (z položky `st_mtime` objektu vraceného funkcí `os.stat()`) na použitelnější strukturu obsahující rok, měsíc, den, hodinu, minutu, sekundu atd. Tento soubor byl naposledy změněn 13. července 2009 přibližně v 17 hodin a 25 minut.

```
# pokračování předchozího příkladu

>>> metadata.st_size                              [1]
3070
>>> import humansize
>>> humansize.approximate_size(metadata.st_size)  [2]
'3.0 KiB'
```

[1] Funkce `os.stat()` vrací také velikost souboru, a to v položce `st_size`. Soubor `feed.xml` obsahuje `3070` bajtů.

[2] Položku `st_size` můžeme předat funkci `approximate_size()`.

3.2.5. Jak vytvořit absolutní cesty

V předcházející podkapitole jsme voláním funkce `glob.glob()` získali seznam s relativními cestami. V prvním příkladu jsme získali cesty jako `'examples\feed.xml'`. V druhém příkladu jsme získali dokonce ještě kratší relativní cesty jako `'romantest1.py'`. Za předpokladu, že zůstaneme ve stejném pracovním adresáři, můžeme tyto relativní cesty používat pro otevření souborů nebo pro získávání jejich metadat. Ale pokud chceme vytvořit absolutní cestu — tj. takovou, která obsahuje jména všech adresářů až po kořenový adresář nebo včetně jména disku —, budeme potřebovat funkci `os.path.realpath()`.

```
>>> import os
>>> print(os.getcwd())
c:\Users\pilgrim\diveintopython3\examples
>>> print(os.path.realpath('feed.xml'))
c:\Users\pilgrim\diveintopython3\examples\feed.xml
```

3.3. Generátorová notace seznamu

Generátorová notace seznamu (anglicky list comprehension [list komprihenšn]) umožňuje stručný zápis vytvoření seznamu z jiného seznamu aplikováním funkce na všechny prvky zdrojového seznamu. (Poznámka překladatele: Pojem „list comprehension“ je znám z deklarativních jazyků a má charakter syntaktické konstrukce. V jazyce Python se „vnitřku“ deklarativního zápisu podobá generátorový výraz. Tímto způsobem byl odvozen český pojem „generátorová notace“. Někdy je pojem „list comprehension“ použit v procedurálním, dynamickém smyslu. V takové situaci můžeme uvažovat o pojmu „generátor seznamu“. Pokud se bavíme o jeho výsledku, můžeme uvažovat i o pojmu „generovaný

V generátorové notaci seznamu můžeme použít libovolný pythonovský výraz.

seznam“. Vzhledem k tomu, že zavedený český pojem pro tuto konstrukci asi neexistuje — studentům příslušných oborů vysokých škol přijde po krátké chvíli anglický pojem srozumitelný —, budu volněji používat některou z uvedených variant. Někdy budu poněkud dlouhý pojem „generátorová notace seznamu“ zkracovat. Kritériem volby bude dobrá srozumitelnost.)

```
>>> a_list = [1, 9, 8, 4]
>>> [elem * 2 for elem in a_list]           [1]
[2, 18, 16, 8]
>>> a_list                                  [2]
[1, 9, 8, 4]
>>> a_list = [elem * 2 for elem in a_list]  [3]
>>> a_list
[2, 18, 16, 8]
```

[1] Aby nám to začalo dávat smysl, podívejme se na zápis zprava doleva. Seznam `a_list` je zde zdrojem zobrazení. Interpret jazyka Python prochází seznam `a_list` po jednom prvku a dočasně přiřazuje jeho hodnotu do proměnné `elem`. Poté Python aplikuje funkci `elem * 2` a připojí výsledek na konec cílového seznamu.

[2] Generátorová notace produkuje nový seznam. Původní seznam zůstává nezměněný.

[3] Výsledek generátoru seznamu můžeme bezpečně přiřadit do proměnné, která zachycovala původní seznam. Python nejdříve vytvoří nový seznam v paměti a teprve po dokončení jeho generování přiřadí výsledek do původní proměnné.

V generátorové notaci seznamu můžeme využít libovolný pythonovský výraz, včetně funkcí z modulu os, které slouží k manipulaci se soubory a adresáři.

```
>>> import os, glob
>>> glob.glob('*.xml')                                      [1]
['feed-broken.xml', 'feed-ns0.xml', 'feed.xml']
>>> [os.path.realpath(f) for f in glob.glob('*.xml')]       [2]
['c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml']
```

[1] Toto volání vrací seznam všech `.xml` souborů v aktuálním pracovním adresáři.

[2] Tato generátorová notace přebírá předchozí seznam `.xml` souborů a transformuje jej na seznam jmen s plnou cestou.

Generátorová notace seznamu může navíc předepisovat i filtraci položek. To znamená, že může vyprodukovat výsledek, který bude kratší než původní seznam.

```
>>> import os, glob
>>> [f for f in glob.glob('*.py') if os.stat(f).st_size > 6000]          [1]
['pluraltest6.py',
 'romantest10.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

[1] Filtraci seznamu provedeme vložením podmínky `if` na konec generátorové notace. Pro každou položku seznamu bude vyhodnocen výraz za klíčovým slovem `if`. Pokud je výsledkem výrazu `True`, pak bude položka zahrnuta do výstupu. Tato generátorová notace seznamu předepisuje zpracování všech souborů s příponou `.py` v aktuálním adresáři. Výraz za `if` zajišťuje filtraci seznamu testováním, zda je velikost každého souboru větší než `6000` bajtů. Takových souborů je šest, takže generátorová notace produkuje seznam se šesti jmény souborů.

Všechny předchozí příklady generátorové notace seznamu používaly jen jednoduché výrazy — násobení čísla konstantou, volání jedné funkce, nebo jednoduše vracely původní položky seznamu (po filtraci). Ale generátorová notace seznamu může být libovolně složitá.

```
>>> import os, glob
>>> [(os.stat(f).st_size, os.path.realpath(f)) for f in glob.glob('*.xml')]          [1]
[(3074, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml'),
 (3386, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml'),
 (3070, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml')]
>>> import humansize
>>> [(humansize.approximate_size(os.stat(f).st_size), f) for f in glob.glob('*.xml')]  [2]
[('3.0 KiB', 'feed-broken.xml'),
 ('3.3 KiB', 'feed-ns0.xml'),
 ('3.0 KiB', 'feed.xml')]
```

[1] Tato generátorová notace nalezne v aktuálním pracovním adresáři všechny soubory s příponou `.xml`, zjistí velikost každého z nich (voláním funkce `os.stat()`) a vytvoří dvojice obsahující jméno souboru a absolutní cestu k souboru (voláním funkce `os.path.realpath()`).

[2] Tento generátorový zápis seznamu vychází z předchozího. Pro velikost každého `.xml` souboru se volá funkce `approximate_size()`.

3.4. Generátorová notace slovníku

Generátorová notace slovníku (anglicky dictionary comprehension [dikšenri komprihenšn]) se podobá generátorové notaci seznamu, ale místo seznamu popisuje vytvoření slovníku.

```
>>> import os, glob
>>> metadata = [(f, os.stat(f)) for f in glob.glob('*test*.py')]                [1]
>>> metadata[0]                                                                 [2]
('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0, st_dev=0,
 st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
 st_mtime=1247520344, st_ctime=1247520344))
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*test*.py')}              [3]
>>> type(metadata_dict)                                                         [4]
<class 'dict'>
>>> list(metadata_dict.keys())                                                  [5]
['romantest8.py', 'pluraltest1.py', 'pluraltest2.py', 'pluraltest5.py',
 'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
 'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
 'romantest3.py', 'romantest5.py', 'romantest6.py', 'alphameticstest.py',
 'pluraltest4.py']
>>> metadata_dict['alphameticstest.py'].st_size                                 [6]
2509
```

[1] Toto není generátorová notace slovníku, ale generátorová notace seznamu. Nalezne všechny soubory s příponou `.py`, které ve svém jméně obsahují podřetězec `test`. Pak se vytvoří dvojice obsahující jméno souboru a jeho metadata (voláním funkce `os.stat()`).

[2] Každá položka výsledného seznamu je dvojice.

[3] Ale toto už je generátorová notace slovníku. Až na dva rozdíly se syntaxe podobá generátorové notaci seznamu. Zaprvé, místo do hranatých závorek je celá uzavřena do složených závorek. Zadruhé, pro každou položku místo jednoho výrazu obsahuje dva výrazy oddělené dvojtečkou. Výraz před dvojtečkou (v našem případě `f`) představuje klíč slovníku. Výraz za dvojtečkou (v našem případě `os.stat(f)`) je hodnota.

[4] Generátorová notace slovníku produkuje slovník.

[5] Klíče uvedeného slovníku zachycují jména souborů, která se vrátila z volání `glob.glob('*test*.py')`.

[6] Hodnotou přidruženou ke každému klíči je hodnota vrácená funkcí `os.stat()`. To znamená, že v tomto slovníku můžeme na základě jména souboru „vyhledat“ jeho metadata. Jednou z částí metadat je `st_size`, zachycující velikost souboru. Soubor `alphameticstest.py` obsahuje `2509` bajtů.

Také u generátorové notace slovníků (podobně jako u generátorové notace seznamů) můžeme přidat podmínku `if`, která zajistí filtraci vyhodnocením výrazu pro každou položku vstupní posloupnosti.

```
>>> import os, glob, humansize
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*')}                    [1]
>>> humansize_dict = {os.path.splitext(f)[0]:humansize.approximate_size(meta.st_size) \
...                   for f, meta in metadata_dict.items() if meta.st_size > 6000}   [2]
>>> list(humansize_dict.keys())                                               [3]
['romantest9', 'romantest8', 'romantest7', 'romantest6', 'romantest10', 'pluraltest6']
>>> humansize_dict['romantest9']                                              [4]
'6.5 KiB'
```

[1] Tato generátorová notace konstruuje seznam všech souborů v aktuálním pracovním adresáři (`glob.glob('*')`), získává metadata každého souboru (`os.stat(f)`) a vytváří slovník, jehož klíči jsou jména souborů a k nim přiřazené hodnoty jsou metadata každého souboru.

[2] Tato generátorová notace vychází z předchozí. Odfiltrovává soubory menší než `6000` bajtů (`if meta.st_size > 6000`) a takto přefiltrovaný seznam používá k vytvoření slovníku. Jeho klíče tvoří jména souborů bez přípony (`os.path.splitext(f)[0]`) a hodnotami jsou přibližné velikosti těchto souborů (`humansize.approximate_size(meta.st_size)`).

[3] V předchozím příkladu jsme si ukázali, že těchto souborů je šest. Z toho vyplývá, že slovník bude mít šest položek.

[4] Hodnotou každého klíče je řetězec vrácený funkcí `approximate_size()`.

3.4.1. Další legrácky s generátorovou notací slovníků

Následující trik využívající generátorové notace slovníku se nám jednoho dne může hodit. Jde o vzájemnou záměnu klíčů a hodnot slovníku.

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
{1: 'a', 2: 'b', 3: 'c'}
```

Bude to samozřejmě fungovat jen v případě, kdy jsou hodnoty ve slovníku neměnitelného typu (immutable), jako jsou řetězce nebo n-tice. Pokud totéž zkusíte se slovníkem, který obsahuje seznamy, dojde k velkolepé havárii.

```
>>> a_dict = {'a': [1, 2, 3], 'b': 4, 'c': 5}
>>> {value:key for key, value in a_dict.items()}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <dictcomp>
TypeError: unhashable type: 'list'
```

3.5. Generátorová notace množin
3.6. Přečtěte si

3.5. Generátorová notace množin

Neměli bychom opomenout, že i syntaxe pro množiny zahrnuje generátorovou notaci. Pozoruhodně se podobá syntaxi pro generátorový zápis slovníků. Jediný rozdíl spočívá v tom, že množiny mají místo párů *klíč: hodnota* jen hodnoty.

```
>>> a_set = set(range(10))
>>> a_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {x ** 2 for x in a_set}              [1]
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> {x for x in a_set if x % 2 == 0}     [2]
{0, 8, 2, 4, 6}
>>> {2**x for x in range(10)}            [3]
{32, 1, 4, 2, 64, 8, 16, 128, 256, 512}
```

[1] Vstupem generátorové notace množiny může být množina. Tato generátorová notace množiny vyhodnocuje druhé mocniny prvků z množiny čísel od `0` do `9`.

[2] Generátorové notace množin (stejně jako generátorové notace seznamů a slovníků) mohou obsahovat podmínku `if`, která vstupní položky před zařazením do výsledné množiny filtruje.

[3] Vstupem generátorové notace množiny ale nemusí být množina. Může jí být jakákoliv posloupnost.

3.6. Přečtěte si

- **modul `os`** (standardní dokumentace)
  (http://docs.python.org/py3k/library/os.html)
- **`os` — Portable access to operating system specific features**
  (přenositelné zpřístupnění specifických vlastností vázaných na operační systém)
  (http://www.doughellmann.com/PyMOTW/os/)
- **modul `os.path`** (standardní dokumentace)
  (http://docs.python.org/py3k/library/os.path.html)
- **`os.path` — Platform-independent manipulation of file names**
  (platformově nezávislá manipulace se jmény souborů)
  (http://www.doughellmann.com/PyMOTW/ospath/)
- **modul `glob`** (standardní dokumentace)
  (http://docs.python.org/py3k/library/glob.html)
- **`glob` — Filename pattern matching** (vyhledávání souborů podle vzorků)
  (http://www.doughellmann.com/PyMOTW/glob/)
- **modul `time`** (standardní dokumentace)
  (http://docs.python.org/py3k/library/time.html)

- `time` — **Functions for manipulating clock time** (funkce pro manipulaci času hodin) (http://www.doughellmann.com/PyMOTW/time/)
- **List comprehensions** (standardní tutorial)
- **Nested list comprehensions** (vnořená generátorová notace seznamů; standardní tutorial)
- **Looping techniques** (techniky zápisu cyklů; standardní tutorial) (vše na http://docs.python.org/py3k/tutorial/datastructures.html)

“I’m telling you this ’cause you’re one of my friends.
My alphabet starts where your alphabet ends!”
(Protože jedním z mých přátel jsi, tak říkám ti:
Má abeceda začíná tam, kde tvá končí!)

— Dr. Seuss, On Beyond Zebra!

4. Řetězce

4. Řetězce — 105
4.1. Pár nudných věcí, kterým musíme rozumět dříve, než se budeme moci ponořit — **107**
4.2. Unicode — **109**
4.3. Ponořme se — **111**
4.4. Formátovací řetězce — **111**
4.4.1. Složená jména oblastí — **113**
4.4.2. Specifikátory formátu — **114**
4.5. Další běžné metody řetězců — **115**
4.5.1. Vykrajování podřetězců — **117**
4.6. Řetězce vs. bajty — **117**
4.7. Závěrečná poznámka: Kódování znaků v pythonovském zdrojovém textu — **120**
4.8. Přečtěte si — **121**

4.1. Pár nudných věcí, kterým musíte rozumět dříve, než se budeme moci ponořit

Přemýšlí o tom jen málo lidí, ale text je neuvěřitelně komplikovaný. Začněme s abecedou. Obyvatelé Bougainville používají nejmenší abecedu na světě. Jejich abeceda Rotokas se skládá z pouhých 12 písmen: A, E, G, I, K, O, P, R, S, T, U a V. Na opačném konci spektra najdeme jazyky, jako jsou čínština, japonština a korejština, které používají tisíce znaků. Angličtina používá 26 písmen — nebo 52, pokud počítáte zvlášť malá a velká písmena — a k tomu pár interpunkčních znaků, jako jsou `!@#$%&`.

Pokud v souvislosti s počítači mluvíte o „textu“, pak pravděpodobně myslíte „znaky a symboly na počítačové obrazovce“. Ale počítače nepracují se znaky a symboly. Pracují s bity a bajty. Každý kousek textu, který jste kdy spatřili na počítačové obrazovce, byl ve skutečnosti uložen v určitém *znakovém kódování*. Zhruba řečeno, kódování znaků zachycuje vztah mezi tím, co vidíte na obrazovce, a tím, co je ve skutečnosti uloženo v paměti počítače a na disku. Znakových kódování se používá velmi mnoho. Některá jsou optimalizována pro konkrétní jazyk, jakým je ruština, čínština nebo angličtina. Jiná kódování se mohou používat pro více jazyků.

Ve skutečnosti je to ještě mnohem komplikovanější. Řada znaků je společná pro více různých kódování, ale každé kódování může pro jejich uložení v paměti nebo na disku používat jinou posloupnost bajtů. Takže o znakovém kódování můžete uvažovat jako o dešifrovacím klíči. Kdykoliv vám někdo poskytne posloupnost bajtů — soubor, webovou stránku, cokoliv — a bude tvrdit, že to je „text“, budete k úspěšnému dekódování bajtů na znaky chtít vědět také to, jaké kódování znaků bylo použito. Pokud vám někdo poskytne špatný klíč nebo vám dokonce nedá žádný, postaví vás před nevyhnutelný úkol rozlousknout kód sami. Může se stát, že při tom uděláte chybu a výsledek bude zmatený.

Vše, co jste si mysleli, že o řetězcích víte, je vám k ničemu.

Určitě už jste viděli webové stránky s podivnými znaky podobnými otazníku na místech, kde měly být apostrofy. Obvykle to znamená, že autor stránky neuvedl jejich správné kódování a váš prohlížeč musel hádat. Výsledkem byla směs očekávaných a neočekávaných znaků. U anglického textu to vnímáme spíš jen rušivě, ale v jiných jazycích může být výsledek zcela nečitelný.

Každý význačný jazyk na světě má definováno své znakové kódování. Každé kódování znaků bylo kvůli rozdílům v jazycích optimalizováno pro konkrétní jazyk, protože paměťový a diskový prostor byly v minulosti velmi drahé. Mám tím na mysli to, že pro reprezentaci znaků jazyka používalo každé kódování stejný interval čísel (0–255). Pravděpodobně znáte například kódování ascii, které ukládá anglické znaky jako čísla z intervalu 0 až 127. (65 je velké „A“, 97 je malé „a“ atd.) Angličtina má velmi jednoduchou abecedu, která může být úplně vyjádřena méně než 128 čísly. Pro ty z vás, kteří umí počítat ve dvojkové soustavě, na to stačí 7 z 8 bitů v bajtu.

Západoevropské jazyky, jakou jsou francouzština, španělština a němčina, používají více znaků než angličtina. Přesněji řečeno, najdete v nich písmena kombinovaná s různými diakritickými značkami, jako například u znaku ñ používaného ve španělštině. Nejběžnějším kódováním je u těchto jazyků CP-1252. Označuje se také „windows-1252“, protože se široce používá v Microsoft Windows. Kódování CP-1252 sdílí znaky v intervalu 0–127 s ASCII, ale rozpíná se i do intervalu 128–255. Nalezneme v něm takové znaky jako n-s-vlnkou (241), u-s-přehláskou (252) atd. Pořád ale jde o jednobajtové kódování. Největší možné číslo (255) se pořád vejde do jednoho bajtu.

Pak tu ale máme jazyky, jako je čínština, japonština a korejština, které používají takové množství znaků, že vyžadují vícebajtové znakové sady. Každý jejich „znak“ je vyjádřen dvoubajtovým číslem v intervalu 0–65535. Ale u různých vícebajtových kódování se pořád setkáváme se stejným problémem, jako u různých jednobajtových kódování. Každé z nich používá stejná čísla pro vyjádření různých věcí. Používají jen širší interval čísel, protože musí vyjádřit mnohem více znaků.

Ve světě, který ještě nebyl propojen sítí a kde „text“ bylo něco, co jste si sami napsali a příležitostně vytiskli, to většinou bylo přijatelné. „Prostého textu“ jste ale moc nenašli. Zdrojové texty byly v ASCII a všichni ostatní používali textové procesory, které definovaly své vlastní (netextové) formáty. Ty si spolu s informacemi o stylu ukládaly také informaci o znakovém kódování. Lidé tyto dokumenty četli prostřednictvím stejných textových procesorů, jaké použil původní autor, takže všechno víceméně fungovalo.

Teď si představte vzestup globálních sítí s elektronickou poštou a s webem. Spousty „prostých textů“ létají kolem zeměkoule — byly napsány na jednom počítači, přeneseny přes druhý a zobrazovány na třetím počítači. Počítače vidí jen čísla. Ale čísla mohou znamenat různé věci. Ach ne! Co budeme dělat? Takže systém musel být navržen tak, aby si každý „prostý text“ s sebou nesl informaci o kódování. Připomeňme si, že jde o dešifrovací klíč, který převádí čísla srozumitelná počítači na znaky čitelné člověkem. Chybějící dešifrovací klíč vede ke zkreslenému textu, zmatkům nebo k něčemu horšímu.

Teď si představte, že bychom více kusů textu chtěli uložit na stejném místě, jako například ve stejné databázové tabulce uchovávající doručenou elektronickou poštu. Pro každý kousek musíme stejně uložit i znakové kódování, abychom text dokázali správně zobrazit. Myslíte si, že je to příliš tvrdý požadavek? Zkuste ve své e-mailové databázi vyhledávat. To znamená, že budete muset za běhu provádět převody mezi různými kódováními. Tady přestává legrace, že?

Teď si představte, že byste měli vícejazyčné dokumenty, ve kterých se znaky z různých jazyků vyskytují vedle sebe, v tom samém dokumentu. (Nápověda: Programy, které se o to pokoušely, typicky používaly pomocné kódy (escape) pro přepínání „režimů“. Prásk, teď jste v ruském režimu koi8-r, takže 241 znamená Я; bum, teď jste řeckém režimu pro Mac, takže 241 znamená ώ.) I *v takových* dokumentech byste samozřejmě chtěli umět vyhledávat.

Tak a teď plačte, protože vše, co jste si mysleli, že o řetězcích víte, je vám k ničemu. Nic takového jako „prostý text“ neexistuje.

4.2. Unicode

Vstupte do světa Unicode.

Unicode je systém navržený tak, aby bylo možné vyjádřit *každý* znak z *každého* jazyka. Každé písmeno, znak nebo ideogram se v Unicode vyjadřují jako 4bajtové číslo. Každé číslo vyjadřuje jedinečný znak, který se používá alespoň v jednom jazyce našeho světa. (Ne všechna čísla jsou využita, ale těch použitých je více než 65535. To znamená, že dva bajty nestačí.) Znaky, které se používají ve více jazycích, mají obvykle stejné číslo — pokud neexistuje dobrý etymologický důvod, aby tomu tak nebylo. Bez ohledu na další okolnosti je ale pro každý znak vyhrazeno jedno číslo a pro každé číslo jen jeden znak. Jedno číslo vždy znamená jedinou věc. Nepoužívají se žádné dříve zmíněné „režimy“. `U+0041` znamená vždy `'A'`, a to i v případech, pokud by váš jazyk `'A'` nepoužíval.

Na první pohled to vypadá jako výborná myšlenka. Jedno kódování vládne všem. Více jazyků v jednom dokumentu. Už nikdy více „přepínání režimu“ uprostřed textu jen kvůli přepnutí kódování. Ale už v této chvíli by vás měla napadnout zjevná otázka. Čtyři bajty? Pro každý jeden znak **?** To vypadá jako hrozné plýtvání. Obzvlášť pro jazyky, jako jsou angličtina nebo španělština, které k vyjádření každého používaného znaku potřebují méně než jeden bajt (256 čísel). Ve skutečnosti je to plýtvání i pro jazyky založené na ideogramech (jako je čínština), které na jeden znak nepotřebují nikdy více než dva bajty.

Existuje kódování Unicode, které používá čtyři bajty na znak. Nazývá se UTF-32, podle počtu 32 bitů, což jsou 4 bajty. UTF-32 je přímočaré kódování. Každé číslo uložené na čtyřech bajtech se reprezentuje jako Unicode znak se stejným číslem. Má to své výhody. Nejdůležitější z nich je ta, že N-tý znak řetězce můžeme zpřístupnit v konstantním čase. N-tý znak totiž začíná na `4×N`-tém bajtu. Ale má to i nevýhody. Ta nejzjevnější je, že na každý podělaný znak potřebujeme čtyři bajty.

Znaků je v Unicode velmi mnoho, ale ukazuje se, že většina lidí nepoužije nikdy žádný, který by ležel mimo prvních 65535. Takže tu máme další kódování Unicode. Nazývá se UTF-16 (protože 16 bitů jsou 2 bajty). V UTF-16 se každý znak s číslem z intervalu 0–65535 kóduje do dvou bajtů. Pokud opravdu potřebujeme vyjádřit zřídka používané Unicode znaky z „astrální roviny“ (přesahující 65535), používá UTF-16 jisté špinavé triky. Nejzjevnější výhoda: UTF-16 je prostorově dvakrát efektivnější než UTF-32, protože pro uložení každého znaku potřebujeme jen dva bajty místo čtyř (s výjimkou těch, pro které to neplatí). A pokud budeme předpokládat, že řetězec neobsahuje žádné znaky z astrální roviny, můžeme snadno najít `N-tý` znak v konstantním čase. Ten předpoklad je docela dobrý, ale jen do doby, kdy to přestane platit.

Ale jak UTF-32, tak UTF-16 mají také méně zřejmé nevýhody. Různé počítačové systémy ukládají jednotlivé bajty různým způsobem. Tak například znak `U+4E2D` by mohl být v UTF-16 uložen buď jako `4E 2D` nebo `2D 4E`. Závisí to na tom, zda systém používá přístup big-endian (na menší adrese významnější bajt) nebo little-endian (na menší adrese méně významný bajt). (Pro UTF-32 existují dokonce ještě další možnosti uspořádání bajtů.) Pokud váš dokument nikdy neopustí váš počítač, je to v suchu — různé aplikace budou na stejném počítači používat stejné pořadí bajtů. Ale v okamžiku, kdy budete chtít dokument přenášet mezi systémy, třeba prostřednictvím webu nebo něčeho takového, budeme

potřebovat způsob, jak vyjádřit námi používané pořadí uložených bajtů. V opačném případě by cílový systém neuměl zjistit, zda dvoubajtová posloupnost `4E 2D` znamená `U+4E2D` nebo `U+2D4E`.

Vícebajtová kódování Unicode pro vyřešení tohoto problému definují „Byte Order Mark“ (značka pořadí bajtů; zkráceně BOM). Jde o speciální netisknutelný znak, který můžete vložit na začátek svého dokumentu, abyste dali najevo, v jakém pořadí jsou vaše bajty uvedeny. Pro UTF-16 je Byte Order Mark roven `U+FEFF`. Pokud obdržíte dokument v `utf-16` začínající bajty `FF FE`, pak víte, že jde o jedno z možných pořadí bajtů. Pokud začíná bajty `FE FF`, pak víte, že pořadí bajtů je obrácené.

Přesto UTF-16 není zcela ideální. Platí to zvláště v případech, kdy používáte velké množství `ASCII` znaků. Když o tom popřemýšlíte, dokonce i čínské webové stránky budou obsahovat velké množství `ASCII` znaků — všechny ty značky a atributy, které obklopují tisknutelné čínské znaky. Pokud umíme najít `N-tý` znak v konstantním čase, je to fajn. Ale pořád tu máme nepříjemný problém s těmi znaky z astrální roviny. To znamená, že nemůžete *zaručit*, že každý znak je uložen přesně na dvou bajtech. Takže ve skutečnosti nemůžete `N-tý` znak najít v konstantním čase — pokud si ovšem neudržujete oddělený index. A mezi námi, ve světě se nachází ohromné množství `ASCII` textů...

Těmito otázkami se už zabývali jiní a přišli s řešením:

UTF-8

`UTF-8` je kódovací systém *s proměnnou délkou*. To znamená, že různé Unicode znaky zabírají různý počet bajtů. Pro `ASCII` znaky (A-Z atd.) používá `UTF-8` jen jeden bajt na znak. Ve skutečnosti používá přesně tentýž bajt. Prvních 128 znaků (0–127) se v `UTF-8` nedá rozlišit od `ASCII`. Znaky z „rozšířené latinky“, jako jsou ñ a ö, budou zabírat dva bajty. (Bajty zde nevyjadřují kód z Unicode tak jednoduchým způsobem, jako je tomu u `UTF-16`. Je do toho zataženo trošku složitější hraní si s bity.) Čínské znaky jako 中 zabírají tři bajty. Zřídka používané znaky z „astrální roviny“ zabírají čtyři bajty.

Nevýhody: Protože každý znak zabírá různý počet bajtů, je nalezení N-tého znaku operací o složitosti O(N). To znamená, že čím je řetězec delší, tím déle budeme znak na určené pozici vyhledávat. Při kódování znaků na bajty a dekódování bajtů na znaky se musíme navíc zabývat dalšími manipulacemi s bity.

Výhody: Kódovaní běžných `ASCII` znaků je extrémně efektivní. Při kódování znaků z rozšířené latinky není horší než UTF-16. Pro čínské znaky je lepší než UTF-32. A díky jednoznačnému způsobu manipulace s bity zde neexistují žádné problémy s pořadím bajtů. (To mi musíte věřit, protože to tady nebudu matematicky zdůvodňovat.) Dokument kódovaný v `UTF-8` používá na každém počítači přesně stejnou posloupnost bajtů.

4.3. Ponořme se
4.4. Formátovací řetězce

4.3. Ponořme se

V Pythonu 3 jsou všechny řetězce posloupnostmi znaků v Unicode. Nenajdeme zde nic takového jako pythonovský řetězec kódovaný v UTF-8 nebo pythonovský řetězec kódovaný v CP-1252. „Je tento řetězec v UTF-8?“ — toto je nesmyslná otázka. UTF-8 představuje způsob kódování znaků do posloupnosti bajtů. Pokud chcete vzít řetězec a přeměnit jej na posloupnost bajtů v určitém znakovém kódování, může vám v tom Python 3 pomoci. Pokud chcete vzít posloupnost bajtů a přeměnit ji na řetězec, pomůže vám s tím Python 3 také. Ale bajty nejsou znaky. Bajty jsou prostě bajty. Znak je abstrakce. A řetězce jsou posloupnostmi těchto abstrakcí.

```
>>> s = '深入 Python'      [1]
>>> len(s)                 [2]
9
>>> s[0]                   [3]
'深'
>>> s + ' 3'               [4]
'深入 Python 3'
```

[1] Řetězec vytvoříme tak, že posloupnost znaků uzavřeme do uvozovacích znaků. Pythonovské řetězce mohou být definovány uzavřením buď do apostrofů (' ; single quotes) nebo do uvozovek (" ; double quotes).

[2] Zabudovaná funkce `len()` vrací délku řetězce, tj. počet znaků. Je to stejná funkce, jakou používáme pro nalezení délky seznamu, n-tice, množiny nebo slovníku. Řetězec připomíná n-tici znaků.

[3] S využitím indexové notace můžeme získat jednotlivé znaky řetězce, podobně jako u seznamu.

[4] Operátor + provádí konkatenaci řetězců (zřetězení, spojení), stejně jako u seznamů.

4.4. Formátovací řetězce

Podívejme se znovu na `humansize.py`:

Řetězce definujeme uzavřením do apostrofů nebo do uvozovek.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],          [1]
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.                           [2]
```

```
    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''                                                                    [3]
    if size < 0:
        raise ValueError('number must be non-negative')                    [4]

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)                      [5]

    raise ValueError('number too large')
```

[1] `'KB', 'MB', 'GB'`… to všechno jsou řetězce.

[2] Dokumentační řetězce funkcí jsou řetězce. Tento dokumentační řetězec se rozprostírá přes několik řádků. Proto je použita trojice apostrofů na začátku i na konci řetězce.

[3] Tato trojice apostrofů ukončuje dokumentační řetězec.

[4] Zde máme další řetězec, který předáváme objektu výjimky jako lidsky čitelnou podobu chybového hlášení.

[5] A tady máme… hej, co je sakra tohle?

Python 3 podporuje formátování hodnot do řetězců. Možné jsou i velmi komplikované výrazy, ale nejzákladnější použití spočívá ve vložení hodnoty do řetězce s jednou oblastí náhrad.

```
>>> username = 'mark'
>>> password = 'PapayaWhip'                              [1]
>>> "{0}'s password is {1}".format(username, password)   [2]
"mark's password is PapayaWhip"
```

[1] Ne, moje heslo doopravdy nezní `PapayaWhip`.

[2] Tady se děje spousta věcí. Zaprvé, voláme zde metodu řetězcového literálu. Řetězce jsou objekty a objekty mají metody. Zadruhé, vyhodnocením celého výrazu vznikne řetězec. Zatřetí, `{0}` a `{1}` jsou *oblasti náhrad* (replacement fields), do kterých budou dosazeny argumenty předané metodě `format()`.

4.4.1. Složená jména oblastí

Předchozí příklad ukazoval nejjednodušší případ, kdy jsou v oblastech náhrad použita pouze celá čísla. Celá čísla se v oblastech náhrad považují za indexy do seznamu argumentů metody `format()`. To znamená, že `{0}` je nahrazena prvním argumentem (v našem případě `username`), `{1}` je nahrazena druhým argumentem (`password`) atd. Můžeme použít tolik pozičních indexů, kolik máme argumentů. A argumentů můžeme mít tolik, kolik chceme. Ale oblasti náhrad jsou ještě mnohem mocnější.

```
>>> import humansize
>>> si_suffixes = humansize.SUFFIXES[1000]                [1]
>>> si_suffixes
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> '1000{0[0]} = 1{0[1]}'.format(si_suffixes)            [2]
'1000KB = 1MB'
```

[1] Místo volání funkce z modulu `humansize` si půjčíme jednu z datových struktur, která je v něm definována: seznam přípon jednotek „SI“ (mocniny čísla 1000).

[2] Vypadá to složitě, ale není to složité. `{0}` se odkazuje na první argument předaný metodě `format()`, tedy na `si_suffixes`. Ale `si_suffixes` má podobu seznamu. Takže `{0[0]}` odkazuje na první položku seznamu, který je prvním argumentem předaným metodě `format(): 'KB'`. Podobně `{0[1]}` odkazuje na druhou položku stejného seznamu: `'MB'`. Všechno vně složených závorek — včetně `1000`, rovnítka a mezer — zůstává nedotčeno. Konečným výsledkem je řetězec `'1000KB = 1MB'`.

{0} je nahrazena prvním argumentem metody format(). {1} je nahrazena druhým argumentem.

Tento příklad ukazuje, že *specifikátory formátu mohou pro zpřístupnění položek a vlastností datových struktur používat (téměř) pythonovskou syntaxi*. Říká se tomu *složená jména oblastí* (compound field names).

Funkční jsou následující složená jména oblastí:

- Předání seznamu a zpřístupnění položky seznamu indexem (jako v předchozím příkladu).
- Předání slovníku a zpřístupnění jeho hodnoty uvedením klíče.
- Předání modulu a zpřístupnění jeho proměnných a funkcí jménem.
- Předání instance třídy a zpřístupnění jejích vlastností a metod jménem.
- Libovolná kombinace výše uvedeného.

Abych vás ohromil, tady máte příklad, který vše kombinuje:

```
>>> import humansize
>>> import sys
>>> '1MB = 1000{0.modules[humansize].SUFFIXES[1000][0]}'.format(sys)
'1MB = 1000KB'
```

A teď si popíšeme, jak to funguje:

- Modul `sys` v sobě udržuje informace o momentálně běžící pythonovské instanci. Protože jsme provedli jeho import, můžeme celý modul `sys` předat jako argument metody `format()`. Takže pole náhrad `{0}` odkazuje na modul `sys`.
- `sys.modules` je slovník všech modulů, které byly importovány touto instancí Pythonu. V roli klíčů vystupují jména modulů uvedená jako řetězce. Hodnotami jsou vlastní objekty modulů. Takže oblast náhrad `{0.modules}` odkazuje na slovník importovaných modulů.
- `sys.modules['humansize']` odkazuje na modul humansize module, který jsme právě importovali. Oblast náhrad `{0.modules[humansize]}` odkazuje na modul humansize. Povšimněte si zde malého rozdílu v syntaxi. Ve skutečném pythonovském kódu jsou klíči slovníku `sys.modules` řetězce. Abychom se jimi mohli odkázat, musíme jméno modulu uzavřít do apostrofů (jako například `'humansize'`). Jenže uvnitř oblasti náhrad apostrofy kolem slovníkového klíče vynecháváme (tj. `humansize`). Citujme PEP 3101: Advanced String Formatting, „Pravidla pro předávání klíčů položek jsou velmi jednoduchá. Pokud klíč začíná číslicí, bude chápán jako číslo. V ostatních případech bude použit jako řetězec.“
- `sys.modules['humansize'].SUFFIXES` je slovník definovaný na začátku modulu humansize. Odkazuje se na něj oblast náhrad `{0.modules[humansize].SUFFIXES}`.
- `sys.modules['humansize'].SUFFIXES[1000]` je seznam přípon jednotek si: `['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']`. Takže oblast náhrad `{0.modules[humansize].SUFFIXES[1000]}` se odkazuje na zmíněný seznam.
- `sys.modules['humansize'].SUFFIXES[1000][0]` je první položkou seznamu přípon jednotek si: `'KB'`. Z toho plyne, že celá oblast náhrad `{0.modules[humansize].SUFFIXES[1000][0]}` je nahrazena dvojznakovým řetězcem `KB`.

4.4.2. Specifikátory formátu

Ale počkat! Ono je toho ještě víc! Podívejme se ještě jednou na tento divný řádek kódu ze souboru `humansize.py`:

```
if size < multiple:
    return '{0:.1f} {1}'.format(size, suffix)
```

`{1}` je nahrazena druhým argumentem předaným metodě `format()`, a tím je `suffix`. Ale co znamená `{0:.1f}`? Jde o dvě věci: význam `{0}` už znáte, ale význam `:.1f` ještě ne. Druhá část (dvojtečka a to, co následuje) definuje *specifikátor formátu* (format specifier), který upřesňuje, jak má být dosazovaná hodnota formátována.

> Specifikátory formátu vám dovolí upravit výsledný text do řady užitečných podob — podobně jako funkce `printf()` v jazyce C. Můžete přidat vycpávku z nul nebo z mezer, zarovnat řetězce, řídit počet desetinných míst a dokonce konvertovat čísla do šestnáctkové soustavy.

Dvojtečka (`:`) uvnitř oblasti náhrad označuje začátek specifikátoru formátu. Specifikátor „`.1`“ znamená „zaokrouhli na nejbližší desetiny“ (tj. zobraz jen jedno místo za desetinnou tečkou). Specifikátor „`f`“ znamená „číslo s pevnou řádovou čárkou“ (jako opak k exponenciálnímu zápisu nebo k jiným způsobům reprezentace čísla). Takže pokud má `size` hodnotu `698.24` a `suffix` hodnotu `'GB'`, pak naformátovaný řetězec bude mít podobu `'698.2 GB'`. Hodnota `698.24` bude zaokrouhlena na jedno desetinné místo a hodnota `suffix` bude připojena za číslo.

```
>>> '{0:.1f} {1}'.format(698.24, 'GB')
'698.2 GB'
```

Detaily kolem specifikátorů formátů naleznete v oficiální pythonovské dokumentaci, v části Format Specification Mini-Language.

4.5. Další běžné metody řetězců

S řetězci můžeme, kromě formátování, provádět řadu dalších užitečných kousků.

```
>>> s = '''Finished files are the re-    [1]
... sult of years of scientif-
... ic study combined with the
... experience of years.'''
>>> s.splitlines()                       [2]
['Finished files are the re-',
 'sult of years of scientif-',
 'ic study combined with the',
 'experience of years.']
>>> print(s.lower())                     [3]
finished files are the re-
sult of years of scientif-
ic study combined with the
experience of years.
>>> s.lower().count('f')                 [4]
6
```

[1] V interaktivním pythonovském shellu můžeme zadat i víceřádkové řetězce. Pokud zahájíme víceřádkový řetězec uvedením trojitého uvozovacího znaku, můžeme jednoduše stisknout `ENTER` a interaktivní shell nás vyzve k zadání pokračování řetězce. Zapsáním uzavírací trojice uvozovacího znaku označíme konec řetězce. Po následném stisku `ENTER` se příkaz provede.

(V tomto případě bude řetězec přiřazen do proměnné s).

[2] Metoda `splitlines()` přebírá jeden víceřádkový řetězec a vrací seznam řetězců, ve kterém každá položka reprezentuje jeden řádek z originálu. Všimněte si, že znaky konců řádků nejsou do jednotlivých řádků zahrnuty.

[3] Metoda `lower()` převádí celý řetězec na malá písmena. (Podobně zase metoda `upper()` převádí řetězec na velká písmena.)

[4] Metoda `count()` vrací počet výskytů zadaného podřetězce. Ano, v uvedené větě je opravdu šest „f“!

Vezměme si další běžný případ. Dejme tomu, že máme seznam dvojic klíč-hodnota ve tvaru `key1=value1&key2=value2` a my bychom je chtěli rozdělit a vytvořit z nich slovník v podobě `{key1: value1, key2: value2}`.

```
>>> query = 'user=pilgrim&database=master&password=PapayaWhip'
>>> a_list = query.split('&')                                    [1]
>>> a_list
['user=pilgrim', 'database=master', 'password=PapayaWhip']
>>> a_list_of_lists = [v.split('=', 1) for v in a_list if '=' in v]   [2]
>>> a_list_of_lists
[['user', 'pilgrim'], ['database', 'master'], ['password', 'PapayaWhip']]
>>> a_dict = dict(a_list_of_lists)                               [3]
>>> a_dict
{'password': 'PapayaWhip', 'user': 'pilgrim', 'database': 'master'}
```

[1] Řetězcové metodě `split()` jsme zadali jeden argument, hodnotu oddělovače. Metoda v místech zadaného oddělovače rozdělí řetězec na seznam řetězců. Zde je jako oddělovač použit znak ampersand, ale může to být cokoliv.

[2] Teď máme seznam řetězců, kde každý obsahuje klíč, následuje znak rovnítka a poté hodnota. K průchodu tímto seznamem a k rozdělení každého řetězce na dva v místě rovnítka můžeme použít generátorovou notaci seznamu (list comprehension). Druhý nepovinný argument metody `split()` říká, kolikrát chceme dělení řetězce provést. Hodnota `1` znamená „rozdělit jen jednou“, takže metoda split() vrátí dvouprvkový seznam. (Hodnota by teoreticky mohla také obsahovat znak rovnítka. Pokud bychom použili pouze `'key=value=foo'.split('=')`, dostali bychom seznam s třemi prvky `['key', 'value', 'foo']`.)

[3] A nakonec necháme Pythonu převést tento seznam seznamů na slovník jednoduše tím, že jej předáme funkci `dict()`.

> Předchozí příklad se hodně podobá získávání parametrů dotazu uvedeného v URL, ale rozklad opravdu používaných URL je ve skutečnosti složitější. Pokud se máte zabývat parametry dotazu v URL, pak pro vás bude mnohem lepší, když použijete funkci `urllib.parse.parse_qs()`. Ta je schopná zvládnout i některé ne příliš zřejmé hraniční případy.

4.5.1. Vykrajování podřetězců

Jakmile máme vytvořen řetězec, můžeme získat jakoukoliv jeho část v podobě nového řetězce. Anglicky se tomu říká „*slicing* the string“, což můžeme přeložit jako „vykrajování z řetězce“ nebo „výřez z řetězce“. Vykrajování podřetězců funguje naprosto stejně jako vykrajování podseznamů. Ono to dává smysl, protože řetězce jsou prosté posloupnosti znaků.

```
>>> a_string = 'My alphabet starts where your alphabet ends.'
>>> a_string[3:11]                          [1]
'alphabet'
>>> a_string[3:-3]                          [2]
'alphabet starts where your alphabet en'
>>> a_string[0:2]                           [3]
'My'
>>> a_string[:18]                           [4]
'My alphabet starts'
>>> a_string[18:]                           [5]
' where your alphabet ends.'
```

[1] Část řetězce, výřez (slice), můžeme získat zadáním dvou indexů. Návratovou hodnotou je nový řetězec, který obsahuje všechny znaky (při zachování pořadí) počínaje prvním indexem výřezu a konče znakem před druhým indexem.

[2] Při vykrajování z řetězců můžeme rovněž použít záporné indexy výřezu, stejně jako u seznamu.

[3] Řetězce se indexují od nuly, takže zápis `a_string[0:2]` vrací první dva znaky řetězce počínaje znakem `a_string[0]` až po `a_string[2]` vyjma (ten už ve výsledku nebude).

[4] Pokud je levý index výřezu roven nule, můžeme nulu vynechat. Bude dosazena implicitně. Takže zápis `a_string[:18]` je stejný jako `a_string[0:18]`. Počáteční nula se dosadí jako implicitní hodnota.

[5] Podobně, pokud by pravý index výřezu měl mít hodnotu rovnou délce řetězce, můžeme jej vynechat. Takže `a_string[18:]` je totéž jako `a_string[18:44]`, protože v tomto řetězci se nachází 44 znaků. A najdeme zde opět potěšitelnou symetrii. Pro tento 44znakový řetězec vrací zápis `a_string[:18]` prvních 18 znaků a `a_string[18:]` vrací vše kromě prvních 18 znaků. Obecně platí, že `a_string[:n]` vždy vrátí prvních n znaků a `a_string[n:]` vrátí zbytek — nezávisle na délce řetězce.

4.6. Řetězce vs. bajty

Bajty jsou bajty, znaky jsou abstrakce. Neměnitelná posloupnost Unicode znaků se nazývá *řetězec*. Neměnitelná posloupnost čísel z intervalu 0–255 se nazývá objekt typu *bytes*.

```
>>> by = b'abcd\x65'             [1]
>>> by
b'abcde'
>>> type(by)                     [2]
<class 'bytes'>
>>> len(by)                      [3]
5
>>> by += b'\xff'                [4]
>>> by
b'abcde\xff'
>>> len(by)                      [5]
6
>>> by[0]                        [6]
97
>>> by[0] = 102                  [7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

[1] Objekt typu `bytes` definujeme použitím `b''`, tedy syntaxe pro „bajtový literál“ . Každý bajt uvnitř bajtového literálu může být buď `ASCII` znak, nebo zakódované šestnáctkové číslo od `\x00` do `\xff` (0–255).

[2] Bajtový objekt je typu `bytes`.

[3] Délku obsahu objektu typu `bytes` můžeme získat zabudovanou funkcí `len()`, tedy stejně jako u seznamů a řetězců.

[4] A stejně jako u seznamů a řetězců, pro konkatenaci (zřetězení, spojení) objektů typu `bytes` můžeme použít operátor `+`. Výsledkem je nový objekt typu `bytes`.

[5] Zřetězením 5bajtového objektu a jednobajtového objektu typu `bytes` vznikne 6bajtový objekt typu `bytes`.

[6] Stejně jako u seznamů a řetězců můžeme jednotlivé bajty z objektu typu `bytes` zpřístupnit indexovou notací. Položkami řetězců jsou znaky, položkami objektu typu `bytes` jsou čísla. Konkrétně jsou to celá čísla z intervalu 0–255.

[7] Objekt typu `bytes` je neměnitelný (immutable). Jednotlivým bajtům nemůžeme nic přiřadit. Pokud potřebujete měnit jednotlivé bajty, můžete buď použít výřezy (slicing) a operátor konkatenace (fungují stejně jako u řetězců), nebo můžete objekt typu `bytes` konvertovat na objekt typu `bytearray`.

```
>>> by = b'abcd\x65'
>>> barr = bytearray(by)         [1]
>>> barr
bytearray(b'abcde')
>>> len(barr)                    [2]
5
>>> barr[0] = 102                [3]
>>> barr
bytearray(b'fbcde')
```

[1] Pro konverzi objektu typu `bytes` na objekt měnitelného typu `bytearray` použijte zabudovanou funkci `bytearray()`.

[2] Všechny metody a operace, které můžete provádět s objektem typu `bytes`, můžete provádět i s objektem typu `bytearray`.

[3] Jedním z rozdílů je to, že objektu typu `bytearray` můžete při využití indexové notace přiřazovat hodnoty jednotlivým bajtům. Přiřazovaná hodnota musí být celé číslo v intervalu 0–255.

Jednou z věcí, které *nikdy nemůžete udělat*, je míchání bajtů s řetězci.

```
>>> by = b'd'
>>> s = 'abcde'
>>> by + s                                [1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
>>> s.count(by)                           [2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> s.count(by.decode('ascii'))           [3]
1
```

[1] Bajty a řetězce nelze spojovat. Jsou různých datových typů.

[2] Nemůžete spočítat výskyt bajtů v řetězci, protože v řetězci žádné bajty nejsou. Řetězec je posloupností znaků. Možná jste měli na mysli „spočítej výskyty řetězce, který bychom získali po dekódování této posloupnosti bajtů při použití určitého znakového kódování“? V pořádku, ale budete to muset zapsat explicitně. Python 3 neprovádí implicitní konverzi bajtů na řetězce a řetězců na bajty.

[3] Překvapivou shodou okolností tento řádek kódu říká „spočítej výskyty řetězce, který bychom získali po dekódování této posloupnosti bajtů při určitém znakovém kódování“.

A tady máme spojení mezi řetězci a bajty: objekt typu `bytes` má metodu `decode()`, která přebírá znakové kódování a vrací řetězec. A řetězce zase mají metodu `encode()`, která přebírá znakové kódování a vrací objekt typu `bytes`. V předchozím případě bylo dekódování poměrně přímočaré — co se týká konverze posloupnosti bajtů v kódování `ASCII` na řetězec znaků. Ale stejný postup funguje pro libovolné kódování, které odpovídá znakům řetězce. Platí to dokonce i pro historická (ne Unicode) kódování.

```
>>> a_string = '深入 Python'          [1]
>>> len(a_string)
9
>>> by = a_string.encode('utf-8')     [2]
>>> by
b'\xe6\xb7\xb1\xe5\x85\xa5 Python'
```

```
>>> len(by)
13
>>> by = a_string.encode('gb18030')  [3]
>>> by
b'\xc9\xee\xc8\xeb Python'
>>> len(by)
11
>>> by = a_string.encode('big5')     [4]
>>> by
b'\xb2\xa4J Python'
>>> len(by)
11
>>> roundtrip = by.decode('big5')    [5]
>>> roundtrip
'深入 Python'
>>> a_string == roundtrip
True
```

[1] Toto je řetězec. Má devět znaků.

[2] Toto je objekt typu `bytes`. Obsahuje 13 bajtů. Posloupnost bajtů vznikla zakódováním řetězce `a_string` do `UTF-8`.

[3] Tento objekt typu `bytes` obsahuje 11 bajtů. Vznikl zakódováním řetězce `a_string` v kódování `GB18030`.

[4] Toto je objekt typu `bytes`. Má 11 bajtů. Jde o *zcela jinou posloupnost bajtů*, která vznikla zakódováním řetězce `a_string` v kódování `Big5`.

[5] Toto je řetězec. Má devět znaků. Jde o posloupnost znaků, kterou jsme získali, když jsme objekt by dekódovali algoritmem `Big5`. Shoduje se s původním řetězcem.

4.7. Závěrečná poznámka: Kódování znaků v pythonovském zdrojovém textu

Python 3 předpokládá, že váš zdrojový kód — tj. každý soubor s příponou `.py` — je uložen v kódování `UTF-8`.

> V Pythonu 2 bylo u souborů s příponou `.py` výchozím kódováním `ASCII`.
> V Pythonu 3 je výchozím kódováním `UTF-8`.

Pokud byste ve svých zdrojových textech chtěli používat jiné kódování, můžete na první řádek souboru vložit deklaraci použitého kódování. Tato deklarace říká, že soubor `.py` používá kódování windows-1252:

```
# -*- coding: windows-1252 -*-
```

Z technického pohledu můžete deklaraci použitého kódování umístit i na druhý řádek. Na prvním řádku se může vyskytovat UNIXovský magický příkazový komentář (hash-bang command).

```
#!/usr/bin/python3
# -*- coding: windows-1252 -*-
```

Více informací naleznete v **PEP 263: Defining Python Source Code Encodings**.
(www.python.org/dev/peps/pep-0263/)

4.8. Přečtěte si

O Unicode v jazyce Python:

- **Python Unicode HOWTO**
  (http://docs.python.org/py3k/howto/unicode.html)
- **What's New In Python 3: Text vs. Data Instead Of Unicode vs. 8-bit**
  (http://docs.python.org/release/3.0.1/whatsnew/3.0.html)
- **PEP 261** vysvětluje, jak Python zachází s astrálními znaky mimo Základní vícejazyčnou rovinu (Basic Multilingual Plane), tj. se znaky s ordinální hodnotou větší než 65535.
  (www.python.org/dev/peps/pep-0261/)

O Unicode obecně:

- **The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)**
  (www.joelonsoftware.com/articles/Unicode.html)
- **On the Goodness of Unicode**
  (www.tbray.org/ongoing/When/200x/2003/04/06/Unicode)
- **On Character Strings**
  (www.tbray.org/ongoing/When/200x/2003/04/13/Strings)
- **Characters vs. Bytes**
  (www.tbray.org/ongoing/When/200x/2003/04/26/UTF)

O znakovém kódování v jiných formátech:

- **Character encoding in XML**
  (http://feedparser.org/docs/character-encoding.html)
- **Character encoding in HTML**
  (http://blog.whatwg.org/the-road-to-html-5-character-encoding)

O řetězcích a jejich formátování:

- **`string` — Common string operations**
- **Format String Syntax**
- **Format Specification Mini-Language**
  (vše na http://docs.python.org/py3k/library/string.html)
- **`PEP` 3101: Advanced String Formatting**
  (www.python.org/dev/peps/pep-3101/)

> **“ Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems. ”**
> (Když se někteří lidé setkají s problémem, pomyslí si: „Já vím! Použiji regulární výrazy.“ V tom okamžiku mají problémy dva.)
>
> — Jamie Zawinski

5. Regulární výrazy

5. Regulární výrazy — 123
5.1. Ponořme se — **125**
5.2. Případová studie: Adresa ulice — **125**
5.3. Případová studie: Římská čísla — **128**
5.3.1. Kontrola tisícovek — **128**
5.3.2. Kontrola stovek — **129**
5.4. Využití syntaxe `{n,m}` — **131**
5.4.1. Kontrola desítek a jednotek — **132**
5.5. Víceslovné regulární výrazy — **134**
5.6. Případová studie:
Analýza telefonních čísel — **136**
5.7. Shrnutí — **141**

5.1. Ponořme se
5.2. Případová studie: Adresa ulice

5.1. Ponořme se

Získávání malých kousků textu z velkých bloků textu představuje výzvu. Pythonovské řetězcové objekty poskytují metody pro vyhledávání a náhrady: `index()`, `find()`, `split()`, `count()`, `replace()` atd. Ale použití těchto metod je omezeno na nejjednodušší případy. Tak například metoda `index()` hledá jediný, pevně zadaný řetězec a vyhledávání je vždy citlivé na velikost písmen. Pokud chceme řetězec s vyhledat bez ohledu na velikost písmen, musíme zavolat `s.lower()` (převod na malá písmena) nebo `s.upper()` (převod na velká písmena) a zajistit odpovídající převod prohledávaných řetězců. Metody `replace()` and `split()` mají stejná omezení.

Pokud svého cíle můžete dosáhnout metodami řetězcového objektu, měli byste je použít. Jsou rychlé, jednoduché a snadno čitelné. O rychlém, jednoduchém a čitelném kódu bychom se mohli bavit ještě dlouho. Ale pokud se přistihnete, že používáte velké množství různých řetězcových funkcí a příkazů `if`, abyste zvládli speciální případy, nebo pokud musíte kombinovat volání `split()` a `join()`, abyste řetězce rozsekávali na kousky a zase je slepovali, v takových případech může být vhodné přejít k regulárním výrazům.

Regulární výrazy představují mocný a (většinou) standardizovaný způsob vyhledávání, náhrad a rozkladu textu se složitými vzorci znaků. Syntaxe regulárních výrazů je sice obtížná a nepodobná normálnímu kódu, ale výsledek může být nakonec *čitelnější* než řešení používající mnoho řetězcových funkcí. Existují dokonce způsoby, jak lze do regulárních výrazů vkládat komentáře. To znamená, že jejich součástí může být podrobná dokumentace.

> Pokud už jste regulární výrazy používali v jiných jazycích (jako jsou Perl, JavaScript nebo PHP), bude vám pythonovská syntaxe připadat důvěrně známá. Abyste získali přehled o dostupných funkcích a jejich argumentech, přečtěte si shrnutí v dokumentaci modulu `re`.

5.2. Případová studie: Adresa ulice

Následující série příkladů byla inspirována problémem, který jsem před několika lety řešil v práci. Potřeboval jsem vyčistit a standardizovat adresy ulic, které byly vyexportované z původního systému, ještě před jejich importem do nového systému. (Vidíte? Já si ty věci jen tak nevymýšlím. Ony jsou ve skutečnosti užitečné.) Tento příklad ukazuje, jak jsem na to šel.

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')                [1]
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')                [2]
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.')  [3]
'100 NORTH BROAD RD.'
```

```
>>> import re                                    [4]
>>> re.sub('ROAD$', 'RD.', s)                   [5]
'100 NORTH BROAD RD.'
```

[1] Mým cílem bylo standardizovat adresu ulice tak, aby se `'ROAD'` vždycky zkrátilo na `'RD.'`. Na první pohled jsem si myslel, že je to dost jednoduché, takže prostě použiji řetězcovou metodu `replace()`. Koneckonců, všechna data už byla převedena na velká písmena, takže problém citlivosti na velikost písmen odpadl. A vyhledávaný řetězec `'ROAD'` je konstantní. A v tomto klamně jednoduchém případě `s.replace()` samozřejmě funguje.

[2] Život je ale, naneštěstí, plný protipříkladů a na jeden takový jsem hned narazil. Problém následující adresy spočívá v dvojím výskytu `'ROAD'`. Jednou jde o část jména ulice `'BROAD'` a jednou o samostatné slovo. Metoda `replace()` tyto dva výskyty najde a slepě je oba nahradí. A já jen pozoruji, jak se mé adresy kazí.

[3] Abychom problém adres s více než jedním výskytem podřetězce `'ROAD'` vyřešili, můžeme se uchýlit k něčemu takovému: hledání a náhradu `'ROAD'` budeme provádět jen v posledních čtyřech znacích adresy (`s[-4:]`) a zbytek řetězce ponecháme beze změny (`s[:-4]`). Ale už sami vidíte, že to začíná být těžkopádné. Například už jen to, že řešení závisí na délce řetězce, který nahrazujeme. (Pokud bychom chtěli nahradit `'STREET'` zkratkou `'ST.'`, museli bychom napsat `s[:-6] a s[-6:].replace(...)`.) Líbilo by se vám, kdybyste se k tomu museli za šest měsíců vrátit a hledat chybu? Jsem si jistý, že ne.

[4] Nastal čas, abychom přešli k regulárním výrazům. Veškerá funkčnost spojená s regulárními výrazy se v Pythonu nachází v modulu `re`.

[5] Podívejme se na první parametr: `'ROAD$'`. Jde o jednoduchý regulární výraz, ke kterému `'ROAD'` pasuje jen v případě, když se vyskytne na konci řetězce. Znak `$` vyjadřuje „konec řetězce“. (Existuje také odpovídající znak, stříška `^`, která znamená „začátek řetězce“.) Voláním funkce `re.sub()` hledáme v řetězci s regulární výraz `'ROAD$'` a nahradíme jej řetězcem `'RD.'`. Nalezne se tím `ROAD` na konci řetězce `s`, ale nenalezne se podřetězec `ROAD`, který je součástí slova `BROAD`. To se totiž nachází uprostřed řetězce `s`.

^ odpovídá začátku řetězce.
$ odpovídá konci řetězce.

Pokračujme v mém příběhu o čištění adres. Brzy jsem zjistil, že předchozí řešení, kdy `'ROAD'` lícuje s koncem adresy, není dost dobré. Ne všechny adresy totiž obsahují údaj, že se jedná o ulici. Některé adresy jednoduše končí jménem ulice. Většinou to vyšlo, ale pokud by se ulice jmenovala `'BROAD'`, pak by regulární výraz pasoval na `'ROAD'`, které se nachází na konci řetězce, ale je součástí slova `'BROAD'`. A to není to, co bych potřeboval.

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)          [1]
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)          [2]
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)          [3]
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s)         [4]
'100 BROAD RD. APT 3'
```

[1] To, co jsem *opravdu* chtěl, bylo vyhledání podřetězce `'ROAD'`, který se nacházel na konci řetězce *a navíc tvořil samostatné slovo* (a ne část nějakého delšího slova). V regulárním výrazu to vyjádříme zápisem `\b`, který má význam „hranice slova se musí vyskytnout právě tady“ (b jako boundary). V Pythonu je to komplikované skutečností, že znak `'\'` musíme v řetězci vyjádřit zvláštním způsobem. (Tento znak se anglicky nazývá též „escape character“ a používá se pro zápis zvláštních posloupností. Má tedy zvláštní význam. Pokud jej chceme použít v prostém významu, musíme jej také zapsat jako „escape“ sekvenci. Prakticky to znamená, že jej musíme zdvojit.) Někdy se to označuje jako mor zpětných lomítek. Je to jeden z důvodů, proč se psaní regulárních výrazů v Perlu jeví snadnější než v jazyce Python. Negativní stránkou Perlu je míchání vlastních regulárních výrazů a odlišností při jejich zápisu. Takže pokud se někde projevuje chyba, dá se někdy obtížně odhadnout, zda je to chyba syntaxe nebo chyba ve vašem regulárním výrazu.

[2] Mor zpětných lomítek můžeme obejít tím, že uvedením písmene `r` před uvozovacím znakem použijeme to, čemu se říká *surový řetězec* (ve smyslu přírodní, nezpracovaný; anglicky raw string). Tím Pythonu říkáme, že se v tomto řetězci nepoužívají speciální posloupnosti (escape sequence). Zápis `'\t'` vyjadřuje tabulační znak, ale `r'\t'` se opravdu chápe jako znak `\` následovaný písmenem t. Pokud budete pracovat s regulárními výrazy, doporučuji vám vždy používat surové řetězce. V opačném případě dospějete velmi rychle k velkým zmatkům. (Regulární výrazy jsou už i tak dost matoucí.)

[3] *Ach jo.* Naneštěstí jsem brzy našel případy, které odporovaly mému přístupu. V tomto případě obsahovala adresa slovo `'ROAD'` jako samostatné slovo, ale to se nenacházelo na konci. Za označením ulice se totiž nacházelo číslo bytu. A protože se `'ROAD'` nenacházelo na úplném konci řetězce, nepasovalo to s regulárním výrazem, takže celé volání `re.sub()` neprovedlo vůbec žádnou náhradu a vrátil se původní řetězec, což nebylo to, co jsem chtěl.

[4] Abych tento problém vyřešil, odstranil jsem znak `$` a přidal jsem další `\b`. Teď už regulární výraz můžeme číst „vyhledej samostatné slovo `'ROAD'` kdekoliv v řetězci“, ať už je to na konci, na začátku nebo někde uprostřed.

5.3. Případová studie: Římská čísla

Římská čísla už jste určitě viděli, i když jste je možná nerozpoznali. Mohli jste je vidět u starých filmů nebo televizních pořadů jako „Copyright `MCMXLVI`“ místo „Copyright `1946`“, nebo na stěnách knihoven a univerzit („založeno `MDCCCLXXXVIII`“ místo „založeno `1888`“). Mohli jste je vidět v různých číslováních a odkazech na literaturu. Jde o systém zápisu čísel, který se opravdu datuje do dob starého římského impéria (proto ten název).

U římských čísel se používá sedm znaků, které se opakují a kombinují různými způsoby, aby vyjádřily číselnou hodnotu.

```
I = 1
V = 5
X = 10
L = 50
C = 100
D = 500
M = 1000
```

Následují základní pravidla pro konstrukci římských čísel:

- V některých případech se znaky sčítají. `I` je `1`, `II` je rovno `2` a `III` znamená `3`. `VI` se rovná `6` (doslova „`5` a `1`“), `VII` je `7` a `VIII` je `8`.
- Desítkové znaky (`I`, `X`, `C` a `M`) se mohou opakovat nanejvýš třikrát. Hodnotu `4` musíme vyjádřit odečtením od dalšího vyššího pětkového znaku. Hodnotu `4` nemůžeme zapsat jako `IIII`. Místo toho ji musíme zapsat jako `IV` („o `1` méně než `5`“). `40` se zapisuje jako `XL` („o `10` méně než `50`“), `41` jako `XLI`, `42` jako `XLII`, `43` jako `XLIII` a následuje `44` jako `XLIV` („o `10` méně než `50` a k tomu o `1` méně než `5`“).
- Někdy znaky vyjadřují... opak sčítání. Když některé znaky umístíme před jiné, provádíme odčítání od konečné hodnoty. Například hodnotu `9` musíme vyjádřit odečtením od dalšího vyššího desítkového znaku: `8` zapíšeme jako `VIII`, ale `9` zapíšeme `IX` („o `1` méně než `10`“) a ne jako `VIIII` (protože znak I nemůžeme opakovat čtyřikrát). `90` je `XC`, `900` je `CM`.
- Pětkové znaky se nesmí opakovat. `10` se vždy zapisuje jako `X` a nikdy jako `VV`. `100` je vždy `C`, nikdy `LL`.
- Římská čísla se čtou zleva doprava, takže na pořadí znaků velmi záleží. `DC` znamená `600`, ale `CD` je úplně jiné číslo (`400`, „o `100` méně než `500`“). `CI` je `101`; `IC` není dokonce vůbec platné římské číslo (protože `1` nemůžeme přímo odčítat od `100`; musíme to napsat jako `XCIX`, „o `10` méně než `100` a k tomu o `1` méně než `10`“).

5.3.1. Kontrola tisícovek

Jak bychom vlastně mohli ověřit, zda je libovolný řetězec platným římským číslem? Podívejme se na to po jednotlivých číslicích. Římské číslice se vždycky píší od největších k nejmenším. Začněme tedy u nejvyšších, na místě tisícovek. U čísel `1000` a vyšších se tisícovky vyjadřují jako řada znaků `M`.

```
>>> import re
>>> pattern = '^M?M?M?$'                [1]
>>> re.search(pattern, 'M')             [2]
<_sre.SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')            [3]
<_sre.SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')           [4]
<_sre.SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')          [5]
>>> re.search(pattern, '')              [6]
<_sre.SRE_Match object at 0106F4A8>
```

[1] Tento vzorek má tři části. Znak `^` zajistí vazbu další části výrazu na začátek řetězce. Pokud bychom jej nepoužili, pak by vzorek pasoval nezávisle na tom, kde by se znaky `M` nacházely. A to bychom nechtěli. Chceme si být jistí ním, že pokud se nějaké znaky `M` najdou, musí se nacházet na začátku řetězce. Zápis `M?` odpovídá nepovinnému výskytu jednoho znaku `M`. A protože se opakuje třikrát, odpovídá výraz výskytu žádného až tří znaků `M` za sebou. Znak `$` odpovídá konci řetězce. Když to dáme dohromady se znakem `^` na začátku, znamená to, že vzorek musí odpovídat celému řetězci. Znakům `M` nemůže žádný jiný znak předcházet a ani za nimi nemůže následovat.

[2] Základem modulu re je funkce `search()`. Ta přebírá regulární výraz (pattern) a řetězec (`'M'`) a zkusí, jestli k sobě pasují. Pokud je shoda nalezena, vrátí funkce `search()` objekt, který nabízí různé metody k popisu výsledku. Pokud ke shodě nedojde, vrací funkce `search()` hodnotu `None`, což je pythonovská hodnota null (nil, nic). V tomto okamžiku nás zajímá jen to, zda vzorek pasuje. Abychom mohli odpovědět, stačí se podívat na návratovou hodnotu funkce `search()`. Řetězec `'M'` odpovídá regulárnímu výrazu, protože první nepovinný znak `M` sedí a druhý a třetí nepovinný znak `M` se ignoruje.

[3] Řetězec `'MM'` vyhovuje, protože první a druhý nepovinný znak `M` pasují a třetí `M` se ignoruje.

[4] Řetězec `'MMM'` vyhovuje, protože všechny tři znaky `M` pasují.

[5] Řetězec `'MMMM'` nevyhovuje. Všechny tři znaky `M` pasují, ale pak regulární výraz trvá na tom, že řetězec musí skončit (protože je to předepsáno znakem `$`). Jenže řetězec ještě nekončí (protože následuje čtvrté `M`). Takže `search()` vrací `None`.

[6] Zajímavé je, že prázdný řetězec tomuto regulárnímu výrazu vyhovuje, protože všechny znaky `M` jsou nepovinné.

5.3.2. Kontrola stovek

? říká, že vzorek je nepovinný.

Kontrola stovek je obtížnější než kontrola tisícovek. Je to tím, že v závislosti na hodnotě existuje několik vzájemně se vylučujících způsobů, kterými mohou být stovky vyjádřeny.

```
100 = C
200 = CC
300 = CCC
400 = CD
500 = D
600 = DC
700 = DCC
800 = DCCC
900 = CM
```

Takže tu máme čtyři možné vzory:

- `CM`
- `CD`
- Žádný až tři znaky `C` (nula v případě, kdy má být na místě stovek `0`).
- `D` následované žádným až třemi znaky `C`.

Poslední dva vzory můžeme zkombinovat:

- Nepovinné `D` následované žádným až třemi znaky `C`.
- Následující příklad ukazuje, jak můžeme u římských čísel ověřit zápis stovek.

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$'  [1]
>>> re.search(pattern, 'MCM')             [2]
<_sre.SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')              [3]
<_sre.SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')          [4]
<_sre.SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')            [5]
>>> re.search(pattern, '')                [6]
<_sre.SRE_Match object at 01071D98>
```

[1] Tento vzorek začíná stejně jako u předchozího příkladu. Kontrolujeme hranici začátku řetězce (`^`) a potom místo pro tisícovky (`M?M?M?`). V závorkách je poté uvedena nová část, která definuje sadu tří vzájemně výlučných vzorků oddělených svislými čarami: `CM`, `CD` a `D?C?C?C?` (což vyjadřuje nepovinné `D` následované žádným nebo třemi znaky `C`). Analyzátor (parser) regulárního výrazu kontroluje každý z těchto vzorků v daném pořadí (zleva doprava), zvolí první, který situaci odpovídá, a ostatní ignoruje.

[2] Řetězec `'MCM'` vyhovuje, protože pasuje první `M`, druhý a třetí znak `M` vzorku se ignorují. Následující podřetězec `CM` odpovídá prvnímu vzorku v závorce (takže části vzorku `CD` a `D?C?C?C?` se neuvažují). `MCM` je římské číslo vyjadřující hodnotu `1900`.

[3] Řetězec 'MD' vyhovuje, protože pasuje první M, druhé a třetí M se ignorují. Vzorek D?C?C?C? pasuje k D (každý z následujících tří znaků C je nepovinný, takže se ignorují). MD je římské číslo vyjadřující 1500.

[4] Řetězec 'MMMCCC' testem prošel. Všechny tři znaky M pasují. Následující vzorek D?C?C?C? pasuje k podřetězci CCC (znak D je nepovinný a ignoruje se). MMMCCC je římské číslo vyjadřující hodnotu 3300.

[5] Řetězec 'MCMC' nevyhovuje. První znak M pasuje, druhé a třetí M se ignorují. Následující CM vyhovuje, ale poté vzorek předepisuje znak $, který nesedí, protože ještě nejsme na konci řetězce. (Pořád nám zbývá nezpracovaný znak C.) Poslední znak C *nelze napasovat* ani na část vzorku D?C?C?C?, protože ta se vzájemně vylučuje s částí vzorku CM, která se již použila.

[6] Zajímavé je, že tomuto vzorku vyhovuje prázdný řetězec, protože všechny znaky M jsou nepovinné a ignorují se. Prázdný řetězec dále vyhovuje i části vzorku D?C?C?C?, protože všechny znaky jsou nepovinné a ignorují se.

Uffff! Vidíte, jak se mohou regulární výrazy rychle stát nechutnými? A to jsme zatím vyřešili části římských čísel jen pro tisíce a stovky. Ale pokud jste zatím vše sledovali, budou pro vás desítky a jednotky jednoduché, protože u nich použijeme naprosto stejný přístup. Ale podívejme se ještě na další možnost vyjádření vzorku.

5.4. Využití syntaxe {n,m}

Zápis {1,4} vyjadřuje 1 až 4 výskyty vzorku.

V předcházející podkapitole jsme pracovali se vzorkem, ve kterém se mohly stejné znaky opakovat až třikrát. V regulárních výrazech existuje ještě jiný způsob, jak to vyjádřit. Někteří lidé jej považují za čitelnější. Podívejme se nejdříve na způsoby, které jsme použili v předcházejícím příkladu.

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')              [1]
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MM')             [2]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMM')            [3]
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM')           [4]
>>>
```

[1] Zde dochází ke shodě se začátkem řetězce a s prvním nepovinným M, ale ne s druhým a s třetím M (což je v pořádku, protože jsou nepovinná). Potom následuje konec řetězce.

[2] Zde dochází ke shodě se začátkem řetězce a s prvním a druhým nepovinným M, ale ne s třetím M (ale to je v pořádku, protože je nepovinné). Poté pasuje i konec řetězce.

[3] Zde dochází ke shodě se začátkem řetězce, se všemi třemi nepovinnými M a s koncem řetězce.

[4] Zde dochází ke shodě se začátkem řetězce a se všemi třemi nepovinnými M, ale poté nenásleduje předepsaný konec řetězce (protože tu máme ještě jedno nepasující M). To znamená, že vzorek nesedí a vrací se None.

```
>>> pattern = '^M{0,3}$'                [1]
>>> re.search(pattern, 'M')             [2]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')            [3]
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')           [4]
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM')          [5]
>>>
```

[1] Tento vzorek říká: „Zde musí být začátek řetězce, potom následují nula až tři znaky M a pak musí být konec řetězce.“ Na místě 0 a 3 mohou být uvedena libovolná čísla. Pokud chceme předepsat „nejméně jeden, ale ne víc než tři znaky M“, můžeme napsat M{1,3}.

[2] Zde dochází ke shodě se začátkem řetězce a pak s jedním ze tří možných M a s koncem řetězce.

[3] Zde dochází ke shodě se začátkem řetězce a pak s dvěma ze tří možných M a s koncem řetězce.

[4] Zde dochází ke shodě se začátkem řetězce a pak s třemi ze tří možných M a s koncem řetězce.

[5] Zde dochází ke shodě se začátkem řetězce a pak s třemi ze tří možných M, ale poté nedochází ke shodě s předpisem pro konec řetězce. Tento regulární výraz předepisuje maximálně tři znaky M následované koncem řetězce, ale řetězec obsahuje čtyři, takže vzorek nepasuje a vrací se None.

5.4.1. Kontrola desítek a jednotek

Rozšiřme tedy regulární výraz pro kontrolu římských čísel o kontrolu na místě desítek a jednotek. Následující příklad ukazuje, jak můžeme kontrolovat desítky.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')         [1]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')          [2]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')         [3]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')       [4]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX')      [5]
>>>
```

[1] Tento řetězec pasuje k předepsanému začátku řetězce, pak k prvnímu nepovinnému M, následuje shoda s CM, poté s XL a s předpisem pro konec řetězce. Připomeňme si, že syntaxe (A|B|C) vyjadřuje „odpovídá právě jednomu z A, B nebo C“. Došlo ke shodě s XL, takže se ignorují možnosti XC a L?X?X?X?. Poté byl nalezen konec řetězce. MCMXL je římské číslo vyjadřující hodnotu 1940.

[2] Tento řetězec vyhovuje předepsanému začátku řetězce, pak prvnímu nepovinnému M, následuje shoda s CM a pak s L?X?X?X?. Co se týká části L?X?X?X?, vyhovuje jí L a přeskakují se všechny tři nepovinné znaky X. Poté se dostáváme ke konci řetězce. MCML je římské číslo vyjadřující hodnotu 1950.

[3] Tento řetězec pasuje k předepsanému začátku řetězce, pak k prvnímu nepovinnému M, následuje shoda s CM, poté s nepovinným L, s prvním nepovinným X, pak se přeskočí druhé a třetí nepovinné X a následuje očekávaný konec řetězce. MCMLX je římské číslo vyjadřující hodnotu 1960.

[4] Tento řetězec vyhovuje předepsanému začátku řetězce, pak prvnímu nepovinnému M, potom CM, pak následuje nepovinné L a všechna tři nepovinná X a vyžadovaný konec řetězce. MCMLXXX je římské číslo vyjadřující hodnotu 1980.

[5] Tento případ vyhovuje předepsanému začátku řetězce, pak prvnímu nepovinnému M, potom CM, pak tu máme nepovinné L a všechna tři nepovinná X, ale poté dochází k selhání předpokladu konce řetězce, protože nám zbývá ještě jedno X, se kterým jsme nepočítali. Takže celý regulární výraz selhává (nepasuje) a vrací se None. MCMLXXXX není platné římské číslo.

(A|B) předepisuje buď shodu se vzorkem A nebo se vzorkem B, ale ne s oběma najednou.

Výraz pro test jednotek vytvoříme stejným způsobem. Ušetřím vás detailů a ukážu vám jen konečný výsledek.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

Takže jak by to vypadalo, kdybychom použili alternativní syntaxi {n,m}? To nám ukáže následující příklad.

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')              [1]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI')          [2]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII')   [3]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I')                 [4]
<_sre.SRE_Match object at 0x008EEB48>
```

[1] Zde dochází ke shodě se začátkem řetězce, pak s jedním ze tří možných znaků `M` a následně s předpisem `D?C{0,3}`. U posledního podvýrazu dochází ke shodě s nepovinným `D` a s nulou ze tří možných znaků `C`. Posuňme se dál. Zde pasuje podvýraz `L?X{0,3}`, protože vyhoví nepovinné `L` a nula ze tří možných znaků `X`. Další kousek řetězce vyhovuje podvýrazu `V?I{0,3}`, protože je nalezeno nepovinné V a nula ze tří možných znaků `I`. A na závěr nastává očekávaný konec řetězce. `MDLV` je římské číslo vyjadřující hodnotu `1555`.

[2] Zde dochází ke shodě se začátkem řetězce a pak s dvěma ze tří možných znaků `M`, pak s `D?C{0,3}` s jedním `D` a s jedním ze tří možných znaků `C`. Pokračujeme `L?X{0,3}` s jedním `L` a jedním ze tří možných znaků `X`. A dále tu máme `V?I{0,3}` s jedním `V` a jedním ze tří možných znaků `I`. Pasuje i očekávaný konec řetězce. `MMDCLXVI` je římské číslo vyjadřující hodnotu `2666`.

[3] Zde dochází ke shodě se začátkem řetězce a pak s třemi ze tří možných znaků `M`, pak je tu `D?C{0,3}` s jedním `D` a s třemi ze tří možných znaků `C`. Pokračujeme `L?X{0,3}` s jedním `L` a s třemi ze tří možných znaků `X`. A dále se uplatní `V?I{0,3}` s jedním `V` a s třemi ze tří možných znaků `I`. A očekávaný konec řetězce. `MMMDCCCLXXXVIII` je římské číslo vyjadřující hodnotu `3888`. Současně je to největší římské číslo, které můžete napsat bez použití rozšířené syntaxe.

[4] A teď se pozorně dívejte. (Připadám si jako kouzelník. „Děti, pozorně se dívejte. Teď ze svého klobouku vytáhnu králíka.“) Tady nám pasuje začátek řetězce, pak následuje nula ze tří možných znaků `M`, pak pasuje `D?C{0,3}` — přeskočení nepovinného `D` a absence znaku `C` (nula až tři možné výskyty). Pokračujeme shodou s podvýrazem `L?X{0,3}` přeskočením nepovinného `L` a přípustnou absencí znaku `X` (nula až tři možné výskyty). A dále se uplatní `V?I{0,3}` přeskočením nepovinného `V` a shodou jednoho ze tří možných znaků `I`. A pak je tu konec řetězce. No páni.

Pokud jste to všechno stihli sledovat a rozuměli jste tomu napoprvé, jde vám to líp, než to šlo mně. Teď si představte, že se snažíte porozumět regulárnímu výrazu, který napsal někdo jiný a který se nachází uprostřed kritické funkce rozsáhlého programu. Nebo si představte, že se po několika měsících vracíte ke svému vlastnímu regulárnímu výrazu. Už se mi to stalo a není to pěkný pohled.

Podívejme se na alternativní syntaxi, která nám pomůže zapsat regulární výraz tak, aby se dal udržovat.

5.5. Víceslovné regulární výrazy

Zatím jsme se zabývali tím, čemu budu říkat „kompaktní“ regulární výrazy. Jak jste sami viděli, obtížně se čtou. Dokonce i když přijdete na to, co nějaký z nich dělá, není tu žádná záruka, že mu budete rozumět o šest měsíců později. To, co opravdu potřebujeme, je dokumentace připisovaná k danému místu.

V Pythonu toho lze dosáhnout u takzvaných *víceslovných regulárních výrazů* (verbose regular expressions). Víceslovný regulární výraz se od kompaktního regulárního výrazu liší ve dvou směrech:

- Bílé znaky se ignorují. Mezery, tabulátory a přechody na nový řádek se nesnaží napasovat na mezery, tabulátory a přechody na nový řádek. Nepasují vůbec k ničemu. (Pokud chcete ve více-

slovném regulárním výrazu předepsat shodu s mezerou, musíte před ni napsat zpětné lomítko — speciální znak (escape) uvozující sekvenci.)

- Komentáře se ignorují. Komentáře uvnitř víceslovných regulárních výrazů mají podobu běžných pythonovských komentářů: začínají znakem # a pokračují do konce řádku. V tomto případě jde o komentář uvnitř víceřádkového řetězce a ne uvnitř zdrojového souboru. Ale funguje stejně.

Z dalšího příkladu to bude jasnější. Revidujme kompaktní regulární výraz, s kterým jsme pracovali před chvílí, a převeďme jej na víceslovný regulární výraz. Příklad nám ukáže, jak na to.

```
>>> pattern = '''
    ^                   # začátek řetězce
    M{0,3}              # tisíce - 0 až 3 M
    (CM|CD|D?C{0,3})    # stovky - 900 (CM), 400 (CD), 0-300 (0 až 3 C),
                        #        nebo 500-800 (D následované 0 až 3 C)
    (XC|XL|L?X{0,3})    # desítky - 90 (XC), 40 (XL), 0-30 (0 až 3 X),
                        #        nebo 50-80 (L následované 0 až 3 X)
    (IX|IV|V?I{0,3})    # jednotky - 9 (IX), 4 (IV), 0-3 (0 až 3 I),
                        #        nebo 5-8 (V následované 0 až 3 I)
    $                   # konec řetězce
    '''
>>> re.search(pattern, 'M', re.VERBOSE)                [1]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)        [2]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII', re.VERBOSE)  [3]
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')                            [4]
```

[1] Nejdůležitější věcí při práci s víceslovnými regulárními výrazy je to, abychom nezapomněli předat jeden argument navíc: v modulu `re` je definována konstanta `re.VERBOSE`, kterou dáváme najevo, že vzorek se má brát jako víceslovný regulární výraz. Jak vidíte, v tomto vzorku se nachází docela hodně bílých znaků (všechny se ignorují) a několik komentářů (opět se všechny ignorují). Pokud budete ignorovat bílé znaky a komentáře, dostanete naprosto stejný regulární výraz, jaký jsme si ukázali v minulé podkapitole. Ale je mnohem čitelnější.

[2] Zde dochází ke shodě se začátkem řetězce a pak s třemi `M`, pak s `CM`, následuje `L` a tři ze tří možných `X`, pak `IX` a konec řetězce.

[3] Tady pasuje začátek řetězce, pak tři z možných tří `M`, následuje `D` a tři ze tří možných `C`, pak `L` a tři ze tří možných `X`, pak `V` a tři ze tří možných `I` a konec řetězce.

[4] Shoda nebyla nalezena. Proč? Protože jsme neuvedli příznak `re.VERBOSE`. Takže funkce `re.search` považuje vzorek za kompaktní regulární výraz, ve kterém hrají roli všechny bílé znaky i znaky `#`. Python nemůže rozpoznávat automaticky, zda je regulární výraz víceslovný nebo ne. Python považuje každý regulární výraz za kompaktní — pokud explicitně neřekneme, že je víceslovný.

5.6. Případová studie: Analýza telefonních čísel

Prozatím jsme se soustředili na shodu celých vzorků. Vzorek buď pasuje, nebo ne. Ale regulární výrazy jsou ještě mnohem mocnější. Pokud regulární výraz *pasuje*, můžeme z řetězce vybrat specifické úseky. Můžeme zjistit, jaká část a kde pasovala.

Následující příklad přinesl opět reálný život. Setkal jsem se s ním o jeden pracovní den dříve než s tím předchozím. Problém: rozklad amerického telefonního čísla. Klient požadoval, aby se číslo dalo zadávat ve volném tvaru (v jednom poli formuláře), ale pak je chtěl mít ve firemní databázi rozdělené na kód oblasti, hlavní linku, číslo a případně klapku. Proštrachal jsem web a našel jsem spoustu příkladů regulárních výrazů, které byly pro tento účel vytvořeny. Ale žádný z nich nebyl dost benevolentní.

\d vyjadřuje libovolnou číslici (0–9). \D vyjadřuje vše kromě číslice.

Tady máme pár telefonních čísel, která měla být přijata:

```
800-555-1212
800 555 1212
800.555.1212
(800) 555-1212
1-800-555-1212
800-555-1212-1234
800-555-1212x1234
800-555-1212 ext. 1234
work 1-(800) 555.1212 #1234
```

Docela široký záběr, že? V každém z těchto případů jsem potřeboval zjistit, že číslo oblasti bylo `800`, číslo hlavní linky bylo `555` a zbytek telefonního čísla byl `1212`. U čísel s klapkou (extension, ext.) jsem potřeboval zjistit, že klapka byla `1234`.

Takže si projděme vývoj řešení pro analýzu telefonního čísla. Následující příklad ukazuje první krok.

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$')  [1]
>>> phonePattern.search('800-555-1212').groups()             [2]
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')                 [3]
>>> phonePattern.search('800-555-1212-1234').groups()        [4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

[1] Regulární výraz čteme vždy zleva doprava. Tento odpovídá začátku řetězce a pak následuje (`\d{3}`). Co to je `\d{3}`? No, `\d` vyjadřuje „libovolnou číslici (`0` až `9`). Společně s `{3}` znamená

„přesně tři číslice“. Jde o variaci na syntaxi `{n,m}`, kterou jsme si ukazovali dříve. Když to vše obalíme do závorek, znamená to „napasuj se přesně na tři číslice *a potom si je zapamatuj jako skupinu, kterou si můžeme vyžádat později*“. Pak musí následovat pomlčka. Pak má následovat skupina zase přesně tří číslic. A pak další pomlčka. A další skupina tentokrát čtyř číslic. A poté se očekává konec řetězce.

[2] Ke skupinám, které se zapamatovaly během analýzy předepsané regulárním výrazem, můžeme přistupovat metodou `groups()` objektu, který vrátila metoda `search()`. Vrací tolikačlennou n-tici, kolik skupin bylo v regulárním výrazu definováno. V našem případě jsme definovali tři skupiny: jednu s třemi číslicemi, další s třemi číslicemi a poslední se čtyřmi číslicemi.

[3] Tento regulární výraz ale není hotový, protože nezvládne telefonní čísla s klapkou na konci. Pro tento účel musíme regulární výraz rozšířit.

[4] Tento případ ilustruje, proč bychom ve skutečně používaném kódu neměli nikdy „řetězit“ použití metod `search()` a `groups()`. Pokud metoda `search()` nevrátí žádnou shodu, vrací None a nikoliv objekt vyjadřující shodu s regulárním výrazem (MatchObject). Volání `None.groups()` vyvolá naprosto zřejmou výjimku. None totiž žádnou metodu `groups()` nemá. (Je to samozřejmě méně zjevné v situaci, kdy se taková výjimka vynoří někde z hloubky našeho kódu. Ano, tady mluvím z vlastní zkušenosti.)

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$')  [1]
>>> phonePattern.search('800-555-1212-1234').groups()              [2]
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                       [3]
>>>
>>> phonePattern.search('800-555-1212')                            [4]
>>>
```

[1] Tento regulární výraz se s předchozím téměř shoduje. Také nejdříve předepisuje začátek řetězce, pak se pamatuje skupina tří číslic, pomlčka, pak se pamatuje skupina tří číslic, pomlčka a nakonec se pamatuje skupina čtyř číslic. Nové je tady to, že se očekává další pomlčka, pak se pamatuje skupina jedné nebo více číslic a teprve potom má nastat konec řetězce.

[2] Metoda `groups()` teď vrací n-tici se čtyřmi prvky, protože regulární výraz nyní definuje čtyři pamatované skupiny.

[3] Tento regulární výraz ale, bohužel, také není konečnou odpovědí, protože předpokládá, že jednotlivé části telefonního čísla jsou odděleny pomlčkou. Co kdyby je někdo oddělil mezerami, čárkami nebo tečkami? Potřebujeme obecnější řešení, které by akceptovalo více typů oddělovačů.

[4] Ouha! Tenhle regulární výraz nejen že nedělá vše, co si přejeme. Je to ve skutečnosti krok zpět, protože teď nejsme schopni analyzovat číslo *bez klapky*. To vůbec není to, co jsme chtěli. Pokud tam klapka je, pak chceme vědět jaká. Pokud tam klapka není, pak chceme znát, jaké byly části hlavního čísla.

Následující příklad ukazuje regulární výraz, který si poradí s různými oddělovači mezi částmi telefonního čísla.

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')  [1]
>>> phonePattern.search('800 555 1212 1234').groups()                    [2]
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()                    [3]
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234')                                [4]
>>>
>>> phonePattern.search('800-555-1212')                                  [5]
>>>
```

[1] Držte si klobouky, jedeme z kopce! Očekáváme začátek řetězce, potom skupinu tří číslic, pak `\D+`. A co je zase tohle? Zápis `\D` vyjadřuje libovolný znak *s výjimkou* číslice a `+` znamená „1 nebo víckrát“. Takže `\D+` pasuje na jeden nebo více znaků, které nejsou číslicemi. A to je právě to, co použijeme místo přímo zapsané pomlčky a co nám bude pasovat s různými oddělovači.

[2] Protože používáme `\D+` místo `-`, bude nám regulární výraz pasovat i na telefonní čísla, kde jsou jednotlivé části odděleny mezerami.

[3] Ale čísla oddělená pomlčkami budou fungovat také.

[4] Stále to ale ještě, bohužel, není konečná odpověď, protože tam nějaký oddělovač je. Co když někdo zadá telefonní číslo úplně bez mezer nebo jiných oddělovačů?

[5] Jejda! Pořád ještě není vyřešeno to, že se požaduje zadání klapky. Takže teď máme dva problémy, ale můžeme je oba vyřešit stejnou technikou.

Následující příklad ukazuje regulární výraz pro telefonní čísla bez oddělovačů.

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  [1]
>>> phonePattern.search('80055512121234').groups()                       [2]
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()                   [3]
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                         [4]
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')                            [5]
>>>
```

[1] Jediná věc, kterou jsme od minulého kroku udělali, byla záměna `+` za `*`. Mezi částmi telefonního čísla nyní místo `\D+` předepisujeme `\D*`. Pamatujete si ještě, že `+` znamená „jednou nebo víckrát“? Fajn. Takže `*` znamená „nula nebo více výskytů“. Takže teď bychom měli být schopni zpracovat čísla, která neobsahují vůbec žádný oddělovací znak.

[2] No podívejme, ono to opravdu funguje! Jak to? Napasovali jsme se na začátek řetězce, pak jsme si zapamatovali skupinu tří číslic (`800`), potom nula nenumerických znaků, pak následuje zapamatovaná skupina tří číslic (`555`), pak nula nenumerických znaků, pak zapamatovaná skupina čtyř číslic (`1212`), pak nula nenumerických znaků, pak zapamatovaná skupina libovolného počtu číslic (`1234`) a konec řetězce.

[3] Ostatní obměny teď fungují také: tečky místo pomlček i kombinace mezer a `x` před klapkou.

[4] Nakonec se nám podařilo vyřešit i dlouho odolávající problém: klapka už je opět nepovinná. Metoda `groups()` vrací n-tici se čtyřmi prvky i tehdy, když nebyla nalezena klapka. V takovém případě se ale na místě čtvrtého prvku vrací prázdný řetězec.

[4] Nechci být poslem špatných zpráv, ale pořád ještě nejsme hotovi. Co je tady špatně? Před kódem oblasti máme znak navíc, ale regulární výraz předpokládá, že na začátku řetězce se má jako první nacházet kód oblasti. Žádný problém. Úvodní znaky před kódem oblasti můžeme přeskočit již dříve představenou technikou „nula nebo více nečíselných znaků".

Další příklad ukazuje, jak bychom si měli počínat.

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  [1]
>>> phonePattern.search('(800)5551212 ext. 1234').groups()                  [2]
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                            [3]
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')                      [4]
>>>
```

[1] Tady je to stejné jako v předchozím příkladu — s tou výjimkou, že před první pamatovanou skupinou znaků (před číslem oblasti) předepisuje `\D*` nula nebo více nenumerických znaků. Všimněte si, že si tyto nenumerické znaky nepamatujeme (předpis není uzavřen v závorkách). Pokud jsou nějaké nalezeny, jednoduše je přeskočíme a teprve pak si zapamatujeme nalezené číslo oblasti.

[2] Telefonní číslo se nám podaří úspěšně rozložit i v případě, kdy je před číslem oblasti uvedena levá závorka. (Pravá závorka za číslem oblasti se už zpracovává. Bere se jako nenumerický oddělovač a napasuje se na předpis `\D*` nacházející se za první pamatovanou skupinou.)

[3] Proveďme ještě test funkčnosti (sanity check), abychom se ujistili, že se nepokazilo nic, co dříve fungovalo. Úvodní znaky jsou zcela nepovinné, takže po začátku řetězce se našlo nula nenumerických znaků, pak pamatovaná skupina tří číslic (`800`), pak jeden nenumerický znak (pomlčka), zapamatovaná skupina tří číslic (`555`), pak jeden nenumerický znak (pomlčka), poté zapamatovaná skupina čtyř číslic (`1212`), pak nula nenumerických znaků, pak zapamatovaná skupina nula číslic a na závěr konec řetězce.

[4] Tak toto je případ, kdy mám v souvislosti s regulárními výrazy chuť vydloubnout si oči tupým předmětem. Proč tohle telefonní číslo nepasuje? Protože se před kódem oblasti vyskytuje `1`, ale my jsme předpokládali, že všechny znaky před kódem oblasti budou nenumerické (`\D*`). Grrrrr.

Podívejme se na to znovu. Zatím se všechny regulární výrazy chytaly na začátek řetězce. Ale teď vidíme, že se na začátku řetězce může vyskytnout obsah neurčité délky, který bychom chtěli ignorovat. Mohli bychom se sice pokusit o vytvoření předpisu, kterým bychom ten začátek přeskočili, ale zkusme k tomu přistoupit jinak. Nebudeme se vůbec snažit o to, abychom se napasovali na začátek řetězce. Zmíněný přístup je použit v následujícím příkladu.

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  [1]
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()         [2]
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                        [3]
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234').groups()                      [4]
('800', '555', '1212', '1234')
```

[1] Všimněte si, že v regulárním výrazu chybí ^. Už se nesnažíme ukotvit na začátek řetězce. Nikde není řečeno, že by se náš regulární výraz měl napasovat na celý vstupní řetězec. Mechanismus, který regulární výraz vyhodnocuje, už si dá tu práci, aby zjistil, od jakého místa vstupního řetězce dochází ke shodě s předpisem, a bude pokračovat odtud.

[2] Teď už jsme úspěšně rozložili telefonní číslo, které obsahuje úvodní znaky i s nechtěnými čísly a které odděluje skupiny chtěných čísel libovolným počtem libovolných oddělovačů.

[3] Test funkčnosti (sanity check). Funguje to správně.

[4] A tohle taky funguje.

Vidíte, jak se může regulární výraz rychle vymknout kontrole? Letmo mrkněte na libovolný z předchozích pokusů. Poznáte snadno rozdíl mezi ním a po něm následujícím?

Takže dokud ještě rozumíme konečnému řešení (a tohle opravdu je konečné řešení; pokud jste objevili případ, který by to nezvládlo, nechci o něm vědět), zapišme ho jako víceslovný regulární výraz. Mohli bychom brzy zapomenout, proč jsme něco zapsali právě takto.

```
>>> phonePattern = re.compile(r'''
                # nevázat se na začátek řetězce, číslo může začít kdekoliv
    (\d{3})     # číslo oblasti má 3 číslice (např. '800')
    \D*         # nepovinný oddělovač - libovolný počet nenumerických znaků
    (\d{3})     # číslo hlavní linky má 3 číslice (např. '555')
    \D*         # nepovinný oddělovač
    (\d{4})     # zbytek čísla má 4 číslice (např. '1212')
    \D*         # nepovinný oddělovač
    (\d*)       # nepovinná klapka - libovolný počet číslic
    $           # konec řetězce
    ''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()          [1]
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                                  [2]
('800', '555', '1212', '')
```

[1] Jediným rozdílem proti regulárnímu výrazu z minulého kroku je to, že je vše rozepsáno na více řádcích. Proto není žádným překvapením, že zpracovává vstupy stejným způsobem.

[2] Konečný test funkčnosti (sanity check). Ano, tohle pořád funguje. Jsme hotovi.

5.7. Shrnutí

Zatím jsme viděli pouhou špičku ledovce z toho, co regulární výrazy zvládnou. Jinými slovy, ačkoliv jimi můžete být momentálně zcela ohromeni, zatím jste neviděli nic. To mi věřte.

Následující věci už by vám neměly být cizí:

`^`	odpovídá začátku řetězce.
`$`	vyjadřuje konec řetězce.
`\b`	odpovídá hranici slova (word boundary).
`\d`	odpovídá číslici.
`\D`	odpovídá znaku jinému než číslice.
`x?`	odpovídá nepovinnému znaku `x` (jinými slovy vyjadřuje žádný nebo jeden výskyt `x`).
`x*`	vyjadřuje nula nebo více výskytů `x`.
`x+`	odpovídá x jedenkrát nebo víckrát.
`x{n,m}`	vyjadřuje znak `x` opakovaný nejméně n-krát, ale ne více než m-krát.
`(a|b|c)`	odpovídá přesně jedné z možností `a`, `b` nebo `c`.
`(x)`	vyjadřuje obecně *zapamatovanou skupinu*. Hodnotu zapamatované skupiny můžeme získat voláním metody `groups()` objektu, který byl vrácen voláním `re.search`.

Regulární výrazy jsou velmi mocné, ale jejich použití není správným řešením pro každý problém. Měli byste se o nich naučit tolik, abyste věděli, kdy je jejich použití vhodné, kdy vám pomohou problém vyřešit a kdy naopak způsobí víc problémů, než vyřeší.

> **“My spelling is Wobbly. It’s good spelling but it Wobbles, and the letters get in the wrong places.”**
> (Mé jméno je Houpavý. Hláskuji to správně, ale Houpe se to a písmenka se dostávají na špatná místa.)
>
> — Medvídek Pú

6. Uzávěry a generátory

6. Uzávěry a generátory — 143
6.1. Ponořme se — **145**
6.2. Já vím jak na to! Použijeme regulární výrazy! — **146**
6.3. Seznam funkcí — **148**
6.4. Seznam vzorků — **150**
6.5. Soubor vzorků — **152**
6.6. Generátory — **154**
6.6.1. Generátor Fibonacciho posloupnosti — **155**
6.6.2. Generátor pravidel pro množné číslo — **156**
6.7. Přečtěte si — **158**

6.1 Ponořme se

Vyrůstal jsem jako syn knihovnice, která vystudovala angličtinu, a vždycky mě fascinovaly jazyky. Nemyslím programovací jazyky. Tedy ano, i programovací jazyky, ale také přirozené jazyky. Dejme tomu angličtina. Angličtina je schizofrenní jazyk, který si slova půjčuje z němčiny, francouzštiny, španělštiny a latiny (když už mám pár vyjmenovat). Slova „půjčuje si“ ve skutečnosti nejsou ta pravá, „vykrádá“ je přiléhavější. Nebo si je možná „asimiluje“ — jako Borg. Jo, to se mi líbí.

```
My jsme Borg. Zvláštnosti vašeho jazyka a původu slov budou přidány do našeho
vlastního. Odpor je marný.
```

V této kapitole se naučíte něco o anglických podstatných jménech v množném čísle. A také o funkcích, které vracejí jiné funkce, o regulárních výrazech pro pokročilé a o generátorech. Ale nejdříve si řekněme
Kap. něco o tom, jak se tvoří podstatná jména v množném čísle. (Pokud jste nečetli kapitolu o regulárních výrazech, tak je na to vhodná doba právě teď. V této kapitole se předpokládá, že základům regulárních výrazů už rozumíte, protože se rychle dostaneme k látce pro pokročilé.)

Pokud jste vyrostli v anglicky mluvící zemi nebo pokud jste se angličtinu učili ve školních lavicích, pak pravděpodobně základní pravidla znáte:

- Pokud slovo končí na S, X nebo Z, přidáme ES. Z *bass* se stává *basses*, z *fax* se stává *faxes* a *waltz* se mění na waltzes.
- Pokud slovo končí hlasitým H, přidáme ES. Pokud končí tichým H, přidáme jen S. Co to je hlasité H? Když H zkombinujeme s jinými písmeny, vydá zvuk, který slyšíme. Takže *coach* [kouč] se změní na coaches a z *rash* [reš] se stane rashes, protože při vyslovování slyšíme zvuky pro CH [č] a SH [š]. Ale z *cheetah* [číta] se stane *cheetahs*, protože H je zde tiché.
- Pokud slovo končí písmenem Y, které zní jako I, změníme Y na IES. Pokud se Y kombinuje se samohláskou tak, že zní jako něco jiného, pak pouze přidáme S. Vacancy se proto změní na *vacancies*, ale z *day* se stane *days*.
- Pokud všechno selhalo, přidáme S a doufáme, že to projde.

(No ano, existuje spousta výjimek. Z *man* se stává *men* a z *woman* zase *women*, ale *human* se mění na *humans*. Mouse přechází v mice a z *louse* je zase *lice*, ale *house* se mění v *houses*. *Knife* přechází v *knives* a z *wife* se stávají *wives*, ale *lowlife* se mění v *lowlifes*. A nechtějte, abych začal o slovech, která jsou sama svým množným číslem (tj. pomnožná), jako jsou *sheep*, *deer* a *haiku*.)

V jiných jazycích je to, samozřejmě, úplně jiné.

Pojďme si navrhnout pythonovskou knihovnu, která automaticky převádí anglická podstatná jména do množného čísla. Začneme s uvedenými čtyřmi pravidly. Ale myslete na to, že budeme nevyhnutelně muset přidávat další.

6.2. Já vím jak na to! Použijeme regulární výrazy!

Takže se díváme na slova, což znamená (přinejmenším v angličtině), že se díváme na řetězce znaků. Pak tady máme pravidla, která nám říkají, že potřebujeme najít různé kombinace znaků a podle nich něco udělat. Vypadá to jako práce pro regulární výrazy!

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):                [1]
        return re.sub('$', 'es', noun)           [2]
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

Kap. [1] Jde o regulární výraz, ale používá syntaxi, se kterou jste se v kapitole Regulární výrazy nesetkali. Hranaté závorky znamenají „napasuj se přesně na jeden z těchto znaků“. Takže `[sxz]` znamená „`s` nebo `x` nebo `z`“, ale jenom jeden z nich. Znak `$` by vám měl být povědomý. Vyjadřuje shodu s koncem řetězce. Když to dáme dohromady, pak tento regulární výraz testuje, zda `noun` (podstatné jméno) končí znakem `s`, `x` nebo `z`.

[2] Funkce `re.sub()` provádí náhrady v řetězci, které jsou založeny na použití regulárního výrazu.

Podívejme se na náhrady předepsané regulárním výrazem podrobněji.

```
>>> import re
>>> re.search('[abc]', 'Mark')                   [1]
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark')                 [2]
'Mork'
>>> re.sub('[abc]', 'o', 'rock')                 [3]
'rook'
>>> re.sub('[abc]', 'o', 'caps')                 [4]
'oops'
```

[1] Obsahuje řetězec `Mark` znak `a`, `b` nebo `c`? Ano, obsahuje `a`.

[2] Fajn. Teď najdi `a`, `b` nebo `c` a nahraď ho znakem `o`. Z `Mark` se stane `Mork`.

[3] Stejná funkce změní `rock` na `rook`.

[4] Mohli byste si myslet, že stejná funkce změní `caps` na `oaps`, ale není tomu tak. Funkce `re.sub` nahrazuje všechny shody s regulárním výrazem, nejenom první z nich. Takže tento regulární výraz změní `caps` na `oops`, protože jak `c`, tak `a` se změní na `o`.

A teď zpět k funkci `plural()` (množné číslo)…

```
def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)          [1]
    elif re.search('[^aeioudgkprt]h$', noun):   [2]
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):         [3]
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

[1] Zde nahrazujeme konec řetězce (shoda s předpisem `$`) řetězcem `es`. Jinými slovy, přidáváme `es` na konec řetězce. Stejného efektu byste mohli dosáhnout konkatenací řetězců (spojením), například použitím `noun + 'es'`. Ale z důvodu, které budou jasnější později, jsem se rozhodl každé pravidlo realizovat pomocí regulárního výrazu.

[2] Teď se pořádně podívejte na následující novinku. Znak `^` uvedený v hranatých závorkách na začátku má speciální význam — negaci. Zápis `[^abc]` znamená „libovolný znak *s výjimkou* `a`, `b` nebo `c`“. Takže `[^aeioudgkprt]` znamená libovolný znak s výjimkou `a`, `e`, `i`, `o`, `u`, `d`, `g`, `k`, `p`, `r` nebo `t`. Tento znak musí být následován znakem `h` a koncem řetězce. Hledáme slova, která končí písmenem `H` a ve kterých je `H` slyšet.

[3] Stejně postupujeme v tomto případě: napasuj se na slova, která končí písmenem `Y`, kde předcházejícím znakem není `a`, `e`, `i`, `o` nebo `u`. Hledáme slova, která končí písmenem `Y`, které zní jako `I`.

Podívejme se na regulární výrazy s negací podrobněji.

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy')          [1]
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy')              [2]
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita')             [3]
>>>
```

[1] `vacancy` tomuto regulárnímu výrazu vyhovuje, protože končí na `cy` a `c` nepatří mezi `a`, `e`, `i`, `o` nebo `u`.

[2] `boy` k regulárnímu výrazu nepasuje, protože končí `oy` a regulárním výrazem jsme přímo řekli, že před znakem `y` nemůže být `o`. Nepasuje ani `day`, protože končí na `ay`.

[3] `pita` nevyhovuje také, protože nekončí na `y`.

```
>>> re.sub('y$', 'ies', 'vacancy')                [1]
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy')   [2]
'vacancies'
```

[1] Tento regulární výraz mění `vacancy` na `vacancies` a `agency` na `agencies`, což jsme chtěli. Všimněte si, že by změnil také `boy` na `boies`, ale k tomu uvnitř funkce nikdy nedojde, protože provedení `re.sub` je podmíněno výsledkem předchozího `re.search`.

[2] Když už jsme u toho, chtěl bych upozornit, že uvedené dva regulární výrazy (jeden, který rozhoduje o uplatnění pravidla, a druhý, který ho realizuje) můžeme zkombinovat do jednoho. Vypadalo by to nějak takto. S většinou výrazu už byste neměli mít problém. Používáme zapamatovanou skupinu, o které jsme si povídali v případové studii zabývající se analýzou telefonních čísel. Skupina se používá k zapamatování si znaku, který se nachází před písmenem `y`. V řetězci s náhradou se pak používá nový syntaktický prvek `\1`, který znamená: „Máš tu první zapamatovanou skupinu? Vlož ji sem.“ V tomto případě se před `y` zapamatovalo `c`. V okamžiku substituce se na místo `c` vloží `c` a `y` se nahradí `ies`. (Pokud pracujete s více než jednou zapamatovanou skupinou, můžete použít `\2` a `\3` a tak dále.)

Náhrady pomocí regulárních výrazů jsou velmi mocné a syntaxe `\1` je činí ještě mocnějšími. Ale zkombinování celé operace do jednoho regulárního výrazu snižuje čitelnost a navíc toto řešení nevyjadřuje přímočaře způsob popisu pravidla pro vytváření množného čísla. Původně jsme pravidlo vyjádřili ve stylu „pokud slovo končí `S`, `X` nebo `Z`, pak přidáme `ES`“. Když se podíváte na zápis funkce, vidíte dva řádky kódu, které říkají „jestliže slovo končí `S`, `X` nebo `Z`, pak přidej `ES`“. Přímočařeji už to snad ani vyjádřit nejde.

6.3. Seznam funkcí

Teď přidáme úroveň abstrakce. Začali jsme definicí seznamu pravidel: Jestliže platí tohle, udělej tamto, v opačném případě přejdi k dalšímu pravidlu. Dočasně zkomplikujeme jednu část programu, abychom mohli zjednodušit jinou.

```
import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)
```

```
def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):                                  [1]
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):                                  [2]
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return True

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),                    [3]
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )

def plural(noun):
    for matches_rule, apply_rule in rules:          [4]
        if matches_rule(noun):
            return apply_rule(noun)
```

[1] V tomto okamžiku má každé rozhodovací (match) pravidlo svou vlastní funkci, která vrací výsledek volání funkce `re.search()`.

[2] Každé aplikační pravidlo má také svou vlastní funkci, která volá funkci `re.sub()` realizující příslušný způsob vytvoření množného čísla.

[3] Místo jedné funkce (`plural()`) s mnoha pravidly teď máme datovou strukturu `rules` (pravidla), která je posloupností dvojic funkcí.

[4] A protože pravidla byla rozbita do podoby oddělené datové struktury, může být nová funkce `plural()` zredukována na pár řádků kódu. V cyklu `for` můžeme z datové struktury `rules` po dvojicích vybírat rozhodovací a aplikační pravidla (jedno rozhodovací a jedno aplikační). Při prvním průchodu cyklem for nabude `matches_rule` hodnoty `match_sxz` a `apply_rule` hodnoty `apply_sxz`. Při druhém průchodu (za předpokladu, že se tak daleko dostaneme) bude proměnné `matches_rule` přiřazena `match_h` a proměnné `apply_rule` bude přiřazena `apply_h`. Je zaručeno, že funkce nakonec něco vrátí, protože poslední rozhodovací funkce (`match_default`) vrací prostě `True`. To znamená, že se provede odpovídající aplikační pravidlo (`apply_default`).

Proměnná „rules“ je posloupností dvojic funkcí.

Funkčnost této techniky je zaručena tím, že v Pythonu je objektem všechno, včetně funkcí. Datová struktura `rules` obsahuje funkce — nikoliv jména funkcí, ale skutečné objekty funkcí. Když v cyklu `for` dojde k jejich přiřazení, stanou se z proměnných `matches_rule` a `apply_rule` skutečné funkce, které můžeme volat. Při prvním průchodu cyklu `for` je to stejné, jako kdyby se volala funkce `matches_sxz(noun)`. A pokud by vrátila objekt odpovídající shodě, zavolala by se funkce `apply_sxz(noun)`.

Pokud se vám přidaná úroveň abstrakce jeví jako matoucí, zkuste si cyklus uvnitř funkce rozepsat a shodu rozpoznáte snadněji. Celý cyklus for je ekvivalentní následujícímu zápisu:

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

Výhodou je, že funkce `plural()` se zjednodušila. Přebírá sadu pravidel, která mohla být definována kdekoliv, a prochází jimi zobecněným způsobem.

1. Získej rozhodovací pravidlo (`match rule`).
2. Došlo ke shodě? Tak volej aplikační pravidlo a vrať výsledek.
3. Nedošlo ke shodě? Přejdi ke kroku 1.

Pravidla mohou být definována kdekoliv, jakýmkoliv způsobem. Funkci `plural()` je to jedno.

Dobrá, ale bylo vůbec přidání úrovně abstrakce k něčemu dobré? No, zatím ne. Zvažme, co to znamená, když k funkci chceme přidat nové pravidlo. V prvním příkladu by to znamenalo přidat do funkce `plural()` příkaz `if`. V tomto druhém příkladu by to vyžadovalo přidání dalších dvou funkcí `match_foo()` a `apply_foo()`. Pak bychom museli určit, do kterého místa posloupnosti `rules` má být dvojice s rozhodovací a aplikační funkcí zařazena (poloha vůči ostatním pravidlům).

Ale to jsme již jen krůček od následující podkapitoly. Pojďme na to...

6.4. Seznam vzorků

Ono ve skutečnosti není nezbytné, abychom pro každé rozhodovací a aplikační pravidlo definovali samostatné pojmenované funkce. Nikdy je nevoláme přímo. Přidáváme je do posloupnosti `rules` a voláme je

přes tuto strukturu. Každá z těchto funkcí navíc odpovídá jednomu ze dvou vzorů. Všechny rozhodovací funkce volají `re.search()` a všechny aplikační funkce volají `re.sub()`. Rozložme tyto vzory tak, abychom si usnadnili budování nových pravidel.

```
import re

def build_match_and_apply_functions(pattern, search, replace):
    def matches_rule(word):                                          [1]
        return re.search(pattern, word)
    def apply_rule(word):                                            [2]
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)                                [3]
```

[1] `build_match_and_apply_functions()` je funkce, která vytváří další funkce dynamicky. Přebírá argumenty `pattern`, `search` a `replace`. Pak definuje rozhodovací funkci `matches_rule()`, která volá `re.search()` s vzorkem `pattern`, který byl předán funkci `build_match_and_apply_functions()`, a se slovem `word`, které se předává právě budované funkci `matches_rule()`. Ty jo!

[2] Aplikační funkce se vytváří stejným způsobem. Aplikační funkce přebírá jeden parametr a volá `re.sub()` s argumenty `search` a `replace`, které byly předány funkci `build_match_and_apply_functions()`, a s parametrem `word`, který se předává právě budované funkci `apply_rule()`. Této technice, kdy se uvnitř dynamicky budované funkce použijí vnější hodnoty, se říká *uzávěr* (closure). Uvnitř budované aplikační funkce v podstatě definujeme konstanty. Funkce přebírá jeden parametr (`word`), potom se chová podle něj, ale také podle dalších dvou hodnot (`search` a `replace`), které platily v době definice aplikační funkce.

[3] Nakonec funkce `build_match_and_apply_functions()` vrátila dvojici hodnot — dvě funkce, které jsme právě vytvořili. Konstanty, které jsme uvnitř těchto funkcí definovali (`pattern` uvnitř funkce `matches_rule()` a `search` a `replace` uvnitř funkce `apply_rule()`), v nich zůstávají uzavřené dokonce i po návratu z funkce `build_match_and_apply_functions()`. To je prostě špica!

Pokud se vám to zdá neuvěřitelně matoucí (a to by mělo, protože to je fakt ujeté), může se to vyjasnit, když uvidíte, jak se to používá.

```
patterns = \                                                         [1]
  (
    ('[sxz]$',           '$',  'es'),
    ('[^aeioudgkprt]h$', '$',  'es'),
    ('(qu|[^aeiou])y$',  'y$', 'ies'),
    ('$',                '$',  's')                                  [2]
  )
rules = [build_match_and_apply_functions(pattern, search, replace)   [3]
         for (pattern, search, replace) in patterns]
```

[1] Naše pravidla (rules) pro tvorbu množného čísla jsou nyní definována jako n-tice trojic *řetězců* (ne funkcí). Prvním řetězcem v každé skupině je regulární výraz, který se bude používat v `re.search()` pro rozhodování, zda se toto pravidlo uplatňuje. Druhý a třetí řetězec ve skupině jsou výrazy pro vyhledání a náhradu, které se použijí v `re.sub()` pro aplikaci pravidla, které sloveso převede do množného čísla.

[2] U záložního pravidla došlo k drobné změně. Pokud v předchozím příkladu nebylo nalezeno žádné ze specifičtějších pravidel, vracela funkce `match_default()` hodnotu `True`, což znamenalo, že se na konec slova jednoduše přidá `s`. Tento dosahuje stejné funkčnosti trochu jinak. Poslední regulární výraz zjišťuje, jestli slovo končí (`$` odpovídá konci řetězce). A samozřejmě, každý řetězec končí (dokonce i prázdný řetězec), takže shoda s tímto výrazem je nalezena vždy. Tento přístup tedy plní stejný účel jako funkce `match_default()`, která vždycky vracela `True`. Pokud nepasuje žádné specifičtější pravidlo, zajistí přidání s na konec daného slova.

[3] Tento řádek je magický. Přebírá řetězce z posloupnosti patterns a mění je na posloupnost funkcí. Jak to dělá? „Zobrazením“ řetězců prostřednictvím funkce `build_match_and_apply_functions()`. To znamená, že se vezme každá trojice řetězců a ty se předají jako argumenty funkci `build_match_and_apply_functions()`. Funkce `build_match_and_apply_functions()` vrátí dvojici funkcí. To znamená, že struktura `rules` získá funkčně shodnou podobu jako v předchozím příkladu — seznam dvojic, kde každá obsahuje dvě funkce. První funkce je rozhodovací (match; pasovat) a volá `re.search()`, druhá funkce je aplikační a volá `re.sub()`.

Skript zakončíme hlavním vstupním bodem, funkcí `plural()`.

```
def plural(noun):
    for matches_rule, apply_rule in rules:       [1]
        if matches_rule(noun):
            return apply_rule(noun)
```

[1] A protože je seznam `rules` stejný jako v předchozím příkladu (a to opravdu je), nemělo by být žádným překvapením, že se funkce `plural()` vůbec nezměnila. Je zcela obecná. Přebírá seznam funkcí realizujících pravidla a volá je v uvedeném pořadí. Nestará se o to, jak jsou pravidla definována. V předcházejícím příkladu byla definována jako pojmenované funkce. Teď jsou funkce pravidel budovány dynamicky zobrazením řetězců ze vstupního seznamu voláním funkce `build_match_and_apply_functions()`. Na tom ale vůbec nezáleží. Funkce `plural()` pracuje stále stejným způsobem.

6.5. Soubor vzorků

Jsme v situaci, kdy už jsme rozpoznali veškeré duplicity v kódu a přešli jsme na dostatečnou úroveň abstrakce. To nám umožnilo definovat pravidla pro vytváření množného čísla v podobě seznamu řetězců. Další logický krok spočívá v uložení těchto řetězců v odděleném souboru. Pravidla (v podobě řetězců) pak mohou být udržována odděleně od kódu, který je používá.

Nejdříve vytvořme textový soubor, který obsahuje požadovaná pravidla. Nebudeme používat žádné efektní datové struktury. Stačí nám tři sloupce řetězců oddělené bílými znaky (whitespace; zde mezery nebo tabulátory). Soubor nazveme `plural4-rules.txt`.

```
[sxz]$               $    es
[^aeioudgkprt]h$     $    es
[^aeiou]y$          y$    ies
$                    $    s
```

Teď se podívejme na to, jak můžeme soubor s pravidly použít.

```
import re

def build_match_and_apply_functions(pattern, search, replace):           [1]
    def matches_rule(word):
        return re.search(pattern, word)
    def apply_rule(word):
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)

rules = []
with open('plural4-rules.txt', encoding='utf-8') as pattern_file:        [2]
    for line in pattern_file:                                            [3]
        pattern, search, replace = line.split(None, 3)                   [4]
        rules.append(build_match_and_apply_functions(                    [5]
                pattern, search, replace))
```

[1] Funkce `build_match_and_apply_functions()` se nezměnila. Pro dynamické vytvoření funkcí, které používají proměnné definované vnější funkcí, pořád používáme uzávěry.

[2] Globální funkce `open()` otvírá soubor a vrací souborový objekt. V tomto případě otvíráme soubor, který obsahuje vzorky řetězců pro převádění podstatných jmen do množného čísla. Příkaz `with` vytváří takzvaný *kontext*. Jakmile blok příkazu `with` skončí, Python soubor automaticky uzavře, a to i v případě, kdyby byla uvnitř bloku with vyvolána výjimka. O blocích `with` a o souborových objektech se dozvíte více v kapitole Soubory.

[3] Obrat `for line in <souborový_objekt>` čte data z otevřeného souborového objektu řádek po řádku a přiřazuje text do proměnné `line` (řádek). O čtení ze souboru se dozvíte více v kapitole Soubory.

[4] Každý řádek souboru obsahuje tři hodnoty, ale jsou oddělené bílými znaky (tabulátory nebo mezerami, na tom nezáleží). Rozdělíme je použitím řetězcové metody `split()`. Prvním argumentem metody `split()` je `None`, což vyjadřuje požadavek „rozdělit v místech posloupností bílých znaků (tabulátorů nebo mezer, na tom nezáleží)“. Druhým argumentem je hodnota `3`, což znamená „rozdělit na místě bílých znaků maximálně 3krát a zbytek řádku ponechat beze změny“. Například řádek `[sxz]$ $ es` bude rozložen na seznam `['[sxz]$', '$', 'es']`. To znamená,

že proměnná `pattern` získá hodnotu `'[sxz]$'`, proměnná search hodnotu `'$'` a proměnná `replace` hodnotu `'es'`. V tak krátkém řádku kódu se skrývá docela hodně síly.

[5] Nakonec předáme `pattern`, `search` a `replace` funkci `build_match_and_apply_functions()`, která vrátí dvojici funkcí. Tuto dvojici připojíme na konec seznamu pravidel, takže nakonec bude `rules` uchovávat seznam rozhodovacích a aplikačních funkcí, které potřebuje funkce `plural()`.

Zdokonalení spočívá v tom, že jsme pravidla pro vytváření množného čísla podstatných jmen oddělili do vnějšího souboru, který může být udržován odděleně od kódu, který pravidla využívá. Kód se stal kódem, z dat jsou data a život je krásnější.

6.6. Generátory

Nebylo by skvělé, kdybychom měli obecnou funkci `plural()`, která si umí sama zpracovat soubor s pravidly? Získala by pravidla, zkontrolovala by, které se má uplatnit, provedla by příslušné transformace, přešla by k dalšímu pravidlu. To je to, co bychom po funkci `plural()` chtěli. A to je to, co by funkce `plural()` měla dělat.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)
            yield build_match_and_apply_functions(pattern, search, replace)

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

Jak sakra funguje *tohle?* Podívejme se nejdříve na interaktivní příklad.

```
>>> def make_counter(x):
...     print('entering make_counter')
...     while True:
...         yield x                     [1]
...         print('incrementing x')
...         x = x + 1
...
>>> counter = make_counter(2)           [2]
>>> counter                             [3]
<generator object at 0x001C9C10>
>>> next(counter)                       [4]
```

```
entering make_counter
2
>>> next(counter)                          [5]
incrementing x
3
>>> next(counter)                          [6]
incrementing x
4
```

[1] Přítomnost klíčového slova `yield` v `make_counter` znamená, že nejde o obyčejnou funkci. Jde o speciální druh funkce, která generuje hodnoty jednu po druhé. Můžeme si ji představit jako funkci, která umí při dalším volání pokračovat v činnosti. Když ji zavoláme, vrátí nám generátor, který můžeme použít pro generování posloupnosti hodnot `x`.

[2] Instanci generátoru `make_counter` vytvoříme tím, že ji zavoláme jako každou jinou funkci. Poznamenejme, že tím ve skutečnosti nedojde k provedení kódu funkce. Jde to poznat i podle toho, že se na prvním řádku funkce `make_counter()` volá `print()`, ale nic se zatím nevytisklo.

[3] Funkce `make_counter()` vrátila objekt generátoru.

[4] Funkce `next()` přebírá objekt generátoru a vrací jeho další hodnotu. Při prvním volání funkce `next()` pro generátor `counter` se provede kód z `make_counter()` až do prvního příkazu `yield` a vrátí se vyprodukovaná hodnota. V našem případě to bude `2`, protože jsme generátor vytvořili voláním `make_counter(2)`.

[5] Při opakovaném volání funkce `next()` pro stejný generátorový objekt se dostáváme přesně do místa, kde jsme minule skončili, a pokračujeme až do místa, kdy znovu narazíme na příkaz `yield`. Při provedení `yield` jsou všechny proměnné, lokální stav a další věci uloženy a při dalším volání `next()` jsou obnoveny. Další řádek kódu, který čeká na provedení, volá funkci `print()`, která vytiskne `incrementing x` (zvyšuji hodnotu `x`). Poté je proveden příkaz `x = x + 1`. Pak se provede další obrátka cyklu `while` a hned se narazí na příkaz `yield x`. Ten uloží stav všeho možného a vrátí aktuální hodnotu proměnné `x` (v tomto okamžiku `3`).

[6] Při druhém volání `next(counter)` se vše opakuje, ale tentokrát má `x` hodnotu `4`.

Protože `make_counter` definuje nekonečný cyklus, mohli bychom pokračovat teoreticky do nekonečna a docházelo by k neustálému zvyšování proměnné `x` a vracení její hodnoty. Místo toho se ale podívejme na užitečnější použití generátorů.

6.6.1. Generátor Fibonacciho posloupnosti

```
def fib(max):
    a, b = 0, 1                 [1]
    while a < max:
        yield a                 [2]
        a, b = b, a + b         [3]
```

[1] Fibonacciho posloupnost je řada čísel, kde každé další číslo je součtem dvou předchozích. Začíná hodnotami `0` a `1`, zpočátku roste pomalu a pak rychleji a rychleji. Na začátku potřebujeme dvě proměnné: `a` s počáteční hodnotou `0` a `b` s počáteční hodnotou `1`.

[2] Proměnná `a` obsahuje aktuální číslo posloupnosti, takže hodnotu vyprodukujeme (yield).

[3] Proměnná `b` představuje další číslo v posloupnosti, takže je přiřadíme do `a`, ale současně vypočteme další hodnotu (`a + b`) a přiřadíme ji do `b` pro pozdější použití. Poznamenejme, že se to děje paralelně. Pokud má `a` hodnotu `3` a `b` hodnotu `5`, pak `a, b = b, a + b` nastaví `a` na `5` (předchozí hodnota `b`) a `b` na `8` (součet předchozí hodnoty `a` a `b`).

„yield“ funkci zastaví. „next()“ pokračuje od místa zastavení.

Dostali jsme funkci, která postupně chrlí Fibonacciho čísla. Mohli byste to popsat i rekurzivním řešením, ale tento způsob je čitelnější. A navíc dobře funguje při použití v cyklech `for`.

```
>>> from fibonacci import fib
>>> for n in fib(1000):                                    [1]
...     print(n, end=' ')                                  [2]
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> list(fib(1000))                                       [3]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

[1] Generátor jako `fib()` můžete v cyklu `for` použít přímo. Cyklus for automaticky získává hodnoty generátoru `fib()` voláním funkce `next()` a přiřazuje je do proměnné cyklu `n`.

[2] Při každé obrátce cyklu `for` získává proměnná n novou hodnotu, která je uvnitř `fib()` produkována příkazem `yield`. Stačí ji jen vytisknout. Jakmile `fib()` dojdou čísla (`a` nabude hodnoty větší než `max`, což je v našem případě `1000`), cyklus `for` elegantně skončí.

[3] Toto je užitečný obrat. Funkci `list()` předáme generátor. Funkce projde (iteruje přes) všechny jeho hodnoty (stejně jako tomu bylo v předchozím příkladu u cyklu `for`) a vrátí seznam všech generovaných hodnot.

6.6.2. Generátor pravidel pro množné číslo

Vraťme se k `plural5.py` a podívejme se, jak tato verze funkce `plural()` pracuje.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)                    [1]
            yield build_match_and_apply_functions(pattern, search, replace)   [2]
```

```
def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):                    [3]
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

[1] Není v tom žádná magie. Vzpomeňte si, že řádky souboru s pravidly obsahují vždy tři hodnoty oddělené bílými znaky. Takže použijeme `line.split(None, 3)` k získání tří „sloupců“ a jejich hodnoty přiřadíme do tří lokálních proměnných.

[2] *A pak vyprodukujeme výsledek (yield).* Jaký výsledek? Dvojici funkcí, které byly dynamicky vytvořeny naší starou známou funkcí `build_match_and_apply_functions()` (je stejná jako v předchozích příkladech). Řečeno jinak, `rules()` je generátor, který na požádání produkuje rozhodovací a aplikační funkce.

[3] Protože `rules()` je generátor, můžeme jej přímo použít v cyklu `for`. Při první obrátce cyklu `for` zavoláme funkci `rules()`, která otevře soubor se vzorky, načte první řádek, na základě vzorků uvedených na řádku dynamicky vybuduje rozhodovací funkci a aplikační funkci a tyto funkce vrátí (yield). Ale během druhé obrátky cyklu `for` se dostáváme přesně do místa, kde jsme kód `rules()` opustili (což je uprostřed cyklu `for line in pattern_file`). První věcí, která se provede, bude načtení řádku souboru (který je pořád otevřen). Na základě vzorků z tohoto řádku souboru se dynamicky vytvoří další rozhodovací a aplikační funkce a tato dvojice se vrátí (yield).

Co jsme vlastně proti verzi 4 získali navíc? Startovací čas. Ve verzi 4 se při importu modulu `plural4` — než jsme mohli vůbec uvažovat o volání funkce `plural()` — načítal celý soubor vzorků a budoval se seznam všech možných pravidel. Při použití generátorů můžeme vše dělat na poslední chvíli. Přečteme si první pravidlo, vytvoříme funkce a vyzkoušíme je. Pokud to funguje, nemusíme číst zbytek souboru nebo vytvářet další funkce.

A co jsme ztratili? Výkonnost! Generátor `rules()` startuje znovu od začátku pokaždé, když voláme funkci `plural()`. To znamená, že soubor se vzorky musí být znovu otevřen a musíme číst od začátku, jeden řádek po druhém.

Chtělo by to nějak získat to nejlepší z obou řešení: minimální čas při startu (žádné provádění kódu při import) a maximální výkonnost (žádné opakované vytváření funkcí). Ale pokud nebudeme muset číst stejné řádky dvakrát, bylo by dobré, aby pravidla mohla zůstat v odděleném souboru (protože kód je kód a data jsou data).

Abychom toho dosáhli, budeme muset vytvořit svůj vlastní iterátor. Ale předtím se musíme naučit něco o pythonovských třídách.

6.7. Přečtěte si

- **PEP 255: Simple Generators**
  (www.python.org/dev/peps/pep-0255/)
- **Understanding Python's "with" statement**
  (http://effbot.org/zone/python-with-statement.htm)
- **Closures in Python**
  (http://ynniv.com/blog/2007/08/closures-in-python.html)
- **Fibonacci numbers**
  (http://en.wikipedia.org/wiki/Fibonacci_number)
- **English Irregular Plural Nouns**
  (www2.gsu.edu/~wwwesl/egw/crump.htm)

> **“East is East, and West is West, and never the twain shall meet.”**
> (Východ je východ, západ je západ a ta dvojice se nikdy nesetká.)
>
> — Rudyard Kipling

7. Třídy a iterátory

7. Třídy a iterátory — 159
7.1. Ponořme se — **161**
7.2. Definice tříd — **161**
7.2.1. Metoda `__init__()` — **162**
7.3. Vytváření instancí tříd — **163**
7.4. Členské proměnné — **163**
7.5. Fibonacciho iterátor — **164**
7.6. Iterátor pro pravidla množného čísla — **166**
7.7. Přečtěte si — **172**

7.1 Ponořme se

Iterátory jsou „tajnou omáčkou“ Pythonu 3. Jsou všude, vše je na nich založeno, vždy zůstávají v pozadí, neviditelné. Generátorové notace jsou jednoduchou formou *iterátorů*. Generátory jsou jednoduchou formou *iterátorů*. Funkce, která produkuje hodnoty příkazem `yield`, je ukázkou pěkného a kompaktního způsobu vytvoření iterátoru, aniž bychom museli iterátor tvořit. Ukážu vám, co tím míním.

Kap.

Vzpomínáte si na Fibonacciho generátor? Tady ho máme v podobě iterátoru vytvořeného od základu:

```
class Fib:
    '''iterator that yields numbers in the Fibonacci sequence'''

    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

Proberme si jeho kód řádek po řádku.

```
class Fib:
```

`class`? Česky se tomu říká *třída*. Ale co to je?

7.2. Definice tříd

Python je plně objektově orientovaný. Můžete definovat své vlastní třídy, dědit ze svých vlastních nebo ze zabudovaných tříd a z definovaných tříd můžete vytvářet instance.

Třídu definujeme v Pythonu jednoduše. Nepoužívá se zde oddělená definice rozhraní — je to jako u funkcí. Prostě definujeme třídu a začneme psát její kód. Pythonovská třída začíná vyhrazeným slovem `class`, za kterým následuje jméno třídy. Z technického pohledu je to vše, co se vyžaduje, protože třída nemusí dědit z žádné jiné třídy.

```
class PapayaWhip:          [1]
    pass                   [2]
```

[1] Jméno této třídy je `PapayaWhip`. Není odvozena od žádné jiné třídy. Jména tříd se obvykle zapisují s velkými písmeny u slov názvu, `KazdeSlovoNazvuTakto`. Ale je to jen konvence, není to závazné.

[2] Asi už jste odhadli, že vše uvnitř třídy je odsazené — podobně jako kód uvnitř funkce, v příkazu `if`, u cyklu `for` nebo v případě jakéhokoliv jiného bloku kódu. Řádek, který není odsazen, už do třídy nepatří.

Třída `PapayaWhip` nedefinuje žádnou metodu ani atributy, ale ze syntaktických důvodů v definici něco být musí. Proto jsme zde použili příkaz `pass`. V Pythonu je toto slovo vyhrazeno a znamená „pokračuj dál, tady není nic k vidění“. Je to příkaz, který nic nedělá. Hodí se nám právě v případech, kdy potřebujeme napsat funkci nebo třídu, která existuje, ale nic nedělá.

> Příkaz `pass` znamená v Pythonu totéž co prázdné složené závorky (`{}`) v jazycích Java nebo C.

Mnohé třídy dědí z jiných tříd, ale to není náš případ. Mnohé třídy definují metody, ale tato ne. Pythonovská třída nemusí mít nic, jen jméno. Obzvláště programátorům v C++ může přijít divné, že pythonovské třídy nemají explicitní konstruktory a destruktory. Ačkoliv se to nevyžaduje, pythonovské třídy *mohou* mít něco, co se konstruktoru podobá. Je to metoda `__init__()`.

7.2.1. Metoda `__init__()`

Následující příklad ukazuje inicializaci třídy `Fib` s využitím metody `__init__`.

```
class Fib:
    '''iterator that yields numbers in the Fibonacci sequence'''         [1]

    def __init__(self, max):                                             [2]
```

[1] Třídy mohou (a měly by) mít své dokumentační řetězce — stejně jako moduly a funkce.

[2] Metoda `__init__()` je zavolána bezprostředně po vytvoření instance třídy. Svádí nás to, abychom ji nazývali „konstruktorem“ třídy, ale z technického hlediska to není pravda. Svádí nás to, protože vypadá jako C++ konstruktor (konvence říká, že by metoda `__init__()` měla být v definici třídy uvedena jako první), chová se jako konstruktor (je to první kousek kódu, který se v nově vytvořené instanci třídy provádí) a vůbec. Chyba! V době volání metody `__init__()` už byl objekt zkonstruován (už existoval) a na novou instanci třídy už máme platný odkaz.

Prvním argumentem metody třídy je vždy odkaz na aktuální instanci třídy a platí to i pro metodu `__init__()`. Podle konvence je tento argument pojmenován `self`. Plní roli vyhrazeného slova, jakým je this v jazycích c++ nebo Java, ale v Pythonu není `self` vyhrazeným slovem. Je to jen konvenční pojmenování. Přesto jej, prosím vás, nenazývejte nikdy jinak než `self`. Jde o velmi silnou konvenci.

Uvnitř metody `__init__()` odkazuje self na nově vytvořený objekt. U ostatních metod třídy odkazuje na instanci třídy, jejíž metoda byla zavolána. V okamžiku definice metody musíme uvést `self` explicitně. Ale v okamžiku volání metody už tento argument *neuvádíme*. Python ho přidá za nás automaticky.

7.3. Vytváření instancí tříd

Vytváření instancí tříd je v Pythonu přímočaré. Jednoduše zavoláme třídu, jako kdyby to byla funkce, a předáme jí argumenty, které vyžaduje metoda `__init__()`. Vrátí se nám nově vytvořený objekt.

```
>>> import fibonacci2
>>> fib = fibonacci2.Fib(100)            [1]
>>> fib                                  [2]
<fibonacci2.Fib object at 0x00DB8810>
>>> fib.__class__                        [3]
<class 'fibonacci2.Fib'>
>>> fib.__doc__                          [4]
'iterator that yields numbers in the Fibonacci sequence'
```

[1] Vytváříme instanci třídy `Fib` (definované v modulu `fibonacci2`) a nově vytvořenou instanci přiřazujeme do proměnné `fib`. Předáváme jeden parametr (`100`), který se při volání metody `__init__()` třídy `Fib` stane jejím argumentem `max`.

[2] `fib` je nyní instancí třídy `Fib`.

[3] Každá instance třídy má zabudovaný atribut `__class__`, který odkazuje na třídu objektu. Programátoři v Javě možná znají třídu `Class`. Ta poskytuje metody jako `getName()` a `getSuperclass()`, které nám zpřístupňují metainformace o objektu. V Pythonu je tento druh metadat přístupný prostřednictvím atributů, ale základní myšlenka je stejná.

[4] Dokumentační řetězec instance můžeme zpřístupnit stejně jako u funkce nebo u modulu. Všechny instance třídy sdílejí stejný `docstring`.

> Novou instanci třídy v Pythonu vytvoříme jednoduše zavoláním třídy, jako kdyby to byla funkce. Nenajdeme zde žádný explicitní operátor `new`, jako je tomu u jazyků `c++` nebo Java.

7.4. Členské proměnné

Pokračujeme k dalšímu řádku:

```
class Fib:
    def __init__(self, max):
        self.max = max        [1]
```

[1] Co to je `self.max`? Jde o členskou proměnnou (nebo také instanční proměnnou nebo proměnnou instance). Je to něco zcela jiného než argument max, který byl předán metodě `__init__()`. `self.max` je „globální“ v rámci instance. To znamená, že k této proměnné můžeme přistupovat z jiných metod.

```
class Fib:
    def __init__(self, max):
        self.max = max          [1]
    .
    .
    .
    def __next__(self):
        fib = self.a
        if fib > self.max:      [2]
```

[1] `self.max` je definována metodou `__init__()`…
[2] …a odkazujeme se na ni v metodě `__next__()`.

Členské proměnné jsou pro každou instanci třídy specifické. Pokud například vytvoříme dvě instance třídy `Fib` s různými hodnotami maxima, bude si každá z nich pamatovat svou vlastní hodnotu.

```
>>> import fibonacci2
>>> fib1 = fibonacci2.Fib(100)
>>> fib2 = fibonacci2.Fib(200)
>>> fib1.max
100
>>> fib2.max
200
```

7.5. Fibonacciho iterátor

Až teď jsme připraveni se naučit, jak se vytváří interátor. Iterátor je jednoduše třída, která definuje metodu `__iter__()`.

Všechny tři z uvedených metod třídy, `__init__`, `__iter__` a `__next__`, začínají a končí dvojicí znaků podtržení (_). Proč zrovna takhle? Není v tom nic magického, ale obvykle to naznačuje, že jde o „speciální metody“. Jedinou „speciální“ věcí je na těchto speciálních metodách to, že se nevolají přímo. Python je volá, když použijete nějaký jiný syntaktický obrat pro třídu nebo pro instanci třídy. Více o speciálních metodách v kapitole Jména speciálních metod.

Kap.

```
class Fib:                                          [1]
    def __init__(self, max):                        [2]
        self.max = max

    def __iter__(self):                             [3]
        self.a = 0
        self.b = 1
        return self

    def __next__(self):                             [4]
        fib = self.a
        if fib > self.max:
            raise StopIteration                     [5]
        self.a, self.b = self.b, self.a + self.b
        return fib                                  [6]
```

[1] Abychom vybudovali iterátor od základů, musíme z `Fib` udělat třídu, a ne funkci.

[2] „Volání“ `Fib(max)` ve skutečnosti znamená vytvoření instance této třídy a zavolání její metody `__init__()` s argumentem `max`. Metoda `__init__()` uloží maximální hodnotu do členské proměnné, takže se na ni mohou později odkazovat ostatní metody.

[3] Metoda `__iter__()` se volá, kdykoliv někdo zavolá `iter(fib)`. (Jak uvidíme za minutku, cyklus `for` ji volá automaticky. Ale vy sami ji můžete volat také, ručně.) Po provedení inicializace na začátku iterace (v tomto případě jde o nastavení počátečního stavu dvou počítadel `self.a` a `self.b`) může metoda `__iter__()` vrátit libovolný objekt, který implementuje metodu `__next__()`. V našem případě (a ve většině případů) metoda `__iter__()` vrátí jednoduše `self`, protože tato třída implementuje svou vlastní metodu `__next__()`.

[4] Metoda `__next__()` se volá vždy, když někdo zavolá funkci `next()` s iterátorem instance třídy. Za minutku to bude dávat větší smysl.

[5] Když metoda `__next__()` vyvolá výjimku `StopIteration`, signalizuje tím volajícímu, že iterace skončila. Na rozdíl od většiny jiných výjimek se zde nesignalizuje chyba. Jde o běžnou situaci, která prostě znamená, že iterátor už nemá žádná data, která by generoval. Pokud je volajícím cyklus `for`, bude výjimka StopIteration zachycena a cyklus bude bezproblémově ukončen. (Jinými slovy, cyklus výjimku spolkne.) Toto malé kouzlo je ve skutečnosti klíčem k použití iterátorů v cyklech `for`.

[6] Vyprodukování další hodnoty provede iterátor tak, že metoda `__next__()` hodnotu jednoduše vrátí příkazem `return`. Nepoužívejte zde příkaz `yield`. Ten je pouze syntaktickým cukrátkem a má význam pouze v souvislosti s generátory. Zde vytváříme od základů svůj vlastní iterátor, proto budeme používat `return`.

Už jste úplně zmatení? Výborně. Podívejme se, jak budeme iterátor volat:

```
>>> from fibonacci2 import Fib
>>> for n in Fib(1000):
...     print(n, end=' ')
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Cože? Vždyť je to úplně stejné! V každém bajtu se to shoduje s voláním generátoru Fibonacciho posloupnosti (až na rozdíl jednoho velkého písmene). Ale jak je to možné?

Cykly `for` v sobě skrývají trochu magie. Odehrává se v nich následující:

- Cyklus `for` volá `Fib(1000)`, jak je vidět z kódu. Vrací se instance třídy `Fib`. Říkejme jí třeba `fib_inst`.
- Cyklus for potají a docela chytře volá funkci `iter(fib_inst)`, která vrátí objekt iterátoru. Říkejme mu třeba `fib_iter`. V našem případě platí `fib_iter == fib_inst`, protože metoda `__iter__()` vrací `self`. Ale o tom cyklus `for` neví (a je mu to jedno).
- Za účelem „průchodu hodnotami“ iterátoru volá cyklus for funkci `next(fib_iter)`, která zase volá metodu `__next__()` objektu `fib_iter`. Ta provede výpočet dalšího Fibonacciho čísla a vrací hodnotu. Cyklus `for` hodnotu převezme, přiřadí ji do proměnné `n` a s touto hodnotou v `n` provede tělo cyklu.
- Jak cyklus `for` ví, kdy má skončit? To jsem rád, že jste se zeptali! Když `next(fib_iter)` vyvolá výjimku `StopIteration`, cyklus `for` ji spolkne a spořádaně se ukončí. (Jakákoliv jiná výjimka se propustí a projeví se obvyklým způsobem.) A kde jsme zahlédli výjimku `StopIteration`? No přece v metodě `__next__()`!

7.6. Iterátor pro pravidla množného čísla

Přišel čas na finále. Přepišme generátor pravidel pro množné číslo do podoby iterátoru.

iter(f) volá f.__iter__
next(f) volá f.__next__

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')
        self.cache = []

    def __iter__(self):
        self.cache_index = 0
        return self
```

```
    def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
            return self.cache[self.cache_index - 1]

        if self.pattern_file.closed:
            raise StopIteration

        line = self.pattern_file.readline()
        if not line:
            self.pattern_file.close()
            raise StopIteration

        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(
            pattern, search, replace)
        self.cache.append(funcs)
        return funcs

rules = LazyRules()
```

Tohle je tedy třída, která implementuje metody `__iter__()` a `__next__()`, takže ji můžeme použít jako iterátor. Za koncem její definice se vytvoří instance třídy a přiřadí se do `rules`. To se stane jen jednou, při importu.

Proberme si zmíněnou třídu po kouscích.

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')          [1]
        self.cache = []                                                          [2]
```

[1] Když vytvoříme instanci třídy `LazyRules` (líná pravidla), otevře se soubor s definicemi vzorků, ale nic se z něj nečte. (K tomu dojde později.)

[2] Po otevření souboru se inicializuje vyrovnávací paměť (cache). Budeme ji používat později, během čtení řádků ze souboru vzorků (v metodě `__next__()`).

Než budeme pokračovat, podívejme se podrobněji na `rules_filename`. Tato proměnná není definována uvnitř metody `__init__()`. Ve skutečnosti není definována uvnitř *žádné* metody. Je definována na úrovni třídy. Jde o *proměnnou třídy*. Ačkoliv k ní můžeme přistupovat stejným způsobem jako k nějaké členské proměnné (`self.rules_filename`), sdílí ji všechny instance třídy `LazyRules`.

```
>>> import plural6
>>> r1 = plural6.LazyRules()
>>> r2 = plural6.LazyRules()
>>> r1.rules_filename                                    [1]
'plural6-rules.txt'
>>> r2.rules_filename
'plural6-rules.txt'
>>> r2.rules_filename = 'r2-override.txt'                [2]
>>> r2.rules_filename
'r2-override.txt'
>>> r1.rules_filename
'plural6-rules.txt'
>>> r2.__class__.rules_filename                          [3]
'plural6-rules.txt'
>>> r2.__class__.rules_filename = 'papayawhip.txt'       [4]
>>> r1.rules_filename
'papayawhip.txt'
>>> r2.rules_filename                                    [5]
'r2-overridetxt'
```

[1] Každá instance třídy dědí atribut `rules_filename` s hodnotou definovanou na úrovni třídy.

[2] Když změníme hodnotu tohoto atributu v jedné instanci, neovlivníme tím ostatní instance…

[3] …a ani neovlivníme atribut třídy. K atributu třídy (v protikladu k atributu jednotlivých instancí) můžeme přistupovat prostřednictvím speciálního atributu `__class__`, který zpřístupňuje třídu jako takovou.

[4] Pokud změníte hodnotu atributu třídy, pak to ovlivní všechny instance, které tuto hodnotu dosud dědí (zde `r1`).

[5] Instance, které tento atribut přepsaly (zde `r2`), ovlivněny nebudou.

Ale zpět k naší ukázce.

```
    def __iter__(self):                 [1]
        self.cache_index = 0
        return self                     [2]
```

[1] Metoda `__iter__()` bude volána pokaždé, když někdo (dejme tomu cyklus `for`) zavolá `iter(rules)`.

[2] Jednou z věcí, kterou musí každá metoda `__iter__()` udělat, je vrácení iterátoru. V tomto případě se vrací `self`, čímž dáváme najevo, že tato třída definuje nějakou metodu `__next__()`, která se postará o vracení hodnot během iterace.

```
    def __next__(self):                                    [1]
        .
        .
        .
        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(           [2]
            pattern, search, replace)
        self.cache.append(funcs)                           [3]
        return funcs
```

[1] Metoda `__next__()` bude volána pokaždé, když někdo (dejme tomu cyklus `for`) zavolá `next(rules)`. Smysl této metody pochopíme, když začneme od jejího konce a půjdeme pozpátku. Takže pojďme na to.

[2] Poslední část této funkce by vám měla být přinejmenším povědomá. Funkce `build_match_and_apply_functions()` se nezměnila. Je pořád stejná, jako vždycky byla.

[3] Jediný rozdíl spočívá v tom, že před vrácením rozhodovací a aplikační funkce (jsou uloženy v dvojici `funcs`) je nejdříve uložíme do `self.cache`.

Posuňme se zpět…

```
    def __next__(self):
        .
        .
        .
        line = self.pattern_file.readline()      [1]
        if not line:                             [2]
            self.pattern_file.close()
            raise StopIteration                  [3]
        .
        .
        .
```

[1] Tady použijeme fintu se souborem pro trošku pokročilejší. Metoda `readline()` (poznámka: jednotné číslo, nikoliv množné `readlines()`) přečte z otevřeného souboru přesně jeden řádek. Přesněji řečeno, přečte další řádek. (Souborové objekty jsou také iterátory! Iterátory jsou všude, až po základy...)

[2] Pokud mohla `readline()` přečíst řádek do proměnné `line`, bude to neprázdný řetězec. Dokonce i kdyby soubor obsahoval prázdný řádek, skončí line jako jednoznakový řetězec `'\n'` (znak konce řádku). Pokud se v proměnné `line` opravdu nachází prázdný řetězec, znamená to, že soubor už neobsahuje žádné další řádky ke čtení.

[3] Když dosáhneme konce souboru, měli bychom soubor zavřít a vyvolat magickou výjimku `StopIteration`. Připomeňme si, že do tohoto bodu jsme se dostali, protože jsme potřebovali rozhodovací a aplikační funkci pro další pravidlo. Další pravidlo je definované dalším řádkem

souboru… Ale další řádek už nemáme! Takže už nemáme co vrátit. Iterace skončila. (The iteration is over. ♫ The party's over… ♫)

A jdeme pozpátku až k začátku metody `__next__()`…

```
    def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
            return self.cache[self.cache_index - 1]      [1]

        if self.pattern_file.closed:
            raise StopIteration                          [2]
        .
        .
        .
```

[1] `self.cache` bude mít podobu seznamu funkcí, které potřebujeme pro rozhodování a aplikaci jednotlivých pravidel. (Přinejmenším tohle by vám mělo být povědomé!) V `self.cache_index` se pamatuje, která další (už zapamatovaná) položka se má vrátit příště. Pokud jsme dosud nevyčerpali prostor se zapamatovanými položkami (tj. pokud je délka `self.cache` větší než `self.cache_index`), pak jsme ji našli (`cache hit`)! Hurá! Rozhodovací a aplikační funkci můžeme vrátit z vyrovnávací paměti a nemusíme je budovat znovu.

[2] Na druhou stranu, pokud jsme na položku ve vyrovnávací paměti nenarazili a zároveň je souborový objekt už uzavřen (což se níže v kódu metody může stát — jak jsme viděli v předcházející ukázce), pak už nemůžeme nic víc dělat. Pokud je soubor uzavřen, znamená to, že jsme jeho obsah vyčerpali. Už jsme přečetli každý jeho řádek a vybudovali jsme funkce pro rozhodování a pro aplikaci pro každý vzorek a uložili jsme je do vyrovnávací paměti. Soubor je vyčerpaný, vyrovnávací paměť je vyčerpaná, já jsem vyčerpaný. Počkat! Co? „Выдержай пионер“ [vyděržaj pijaněr], už je to skoro hotové.

Když to dáme všechno dohromady, provádí se následující:

- V okamžiku importu modulu se vytvoří jediná instance třídy `LazyRules`, která je nazvaná `rules` (pravidla). Tato instance otevřela soubor se vzorky, ale nečetla z něj.
- V okamžiku, kdy se požaduje první dvojice funkcí pro rozhodování a pro aplikaci, dojde ke kontrole vyrovnávací paměti, ale zjistí se, že je prázdná. Takže se ze souboru přečte jeden řádek se vzorky, vybudují se podle něj funkce pro rozhodování a pro aplikaci a uloží se do vyrovnávací paměti.
- Dejme tomu, že vyhovělo úplně první pravidlo. Pokud tomu tak je, žádné další funkce pro rozhodování a aplikaci se nevytvářejí a ze souboru se nečtou žádné další řádky.
- Dále dejme tomu, že volající zavolá funkci `plural()` *znovu*, protože chce převést do množného čísla jiné slovo. Cyklus `for` ve funkci `plural()` zavolá `iter(rules)`, což vede k nastavení inde-

xu vyrovnávací paměti na začátek, ale nedojde k resetování otevřeného souborového objektu.

- Při prvním průchodu požádá cyklus `for` o hodnotu ze struktury `rules`, což vede k zavolání jeho metody `__next__()`. Ale v tomto okamžiku už vyrovnávací paměť obsahuje jediný pár funkcí pro rozhodování a pro aplikaci — odpovídají vzorkům z prvního řádku souboru. Protože už byly vytvořeny a uloženy do vyrovnávací paměti při zpracování minulého slova, jsou z ní vybrány. Index do vyrovnávací paměti se zvýší a otevřený soubor zůstane nedotčen.
- Dejme tomu, že první pravidlo tentokrát *nevyhovělo*. Cyklus for udělá další obrátku a zeptá se na další hodnotu ze seznamu `rules`. Tím se podruhé aktivuje metoda `__next__()`. Tentokrát je ale vyrovnávací paměť vyčerpána, protože obsahovala jen jednu položku a my jsme požádali o druhou. Takže metoda `__next__()` pokračuje v činnosti. Z otevřeného souboru přečte další řádek, vybuduje podle něj rozhodovací a aplikační funkci a dvojici uloží do vyrovnávací paměti.
- Pokud pravidla budovaná z načítaných řádků souboru pro zadané slovo nevyhovují, pokračuje proces fázemi „přečti, vybuduj, ulož“ dál. Jakmile se nalezne vhodné pravidlo před koncem souboru, jednoduše se použije a další načítání se zastaví. Soubor zůstane otevřený. Ukazatel do souboru zůstane tam, kde jsme přestali číst, a bude se čekat na další příkaz `readline()`. Ve vyrovnávací paměti se teď nachází více položek. Pokud znovu zahájíme vytváření množného čísla pro nové slovo, vyzkoušíme před případným čtením dalšího řádku souboru nejdříve všechny položky z vyrovnávací paměti.

Dosáhli jsme „množnočíselné“ nirvány.

1. **Minimální startovací čas**. Jediné činnosti, které se při příkazu `import` provedou, jsou vytvoření jediné instance třídy a otevření souboru (ale nečte se z něj).
2. **Maximální výkonnost**. U předcházejícího příkladu bychom četli ze souboru a dynamicky budovali funkce pokaždé, když bychom chtěli vytvořit množné číslo zadaného slova. V této verzi dochází hned po vybudování funkcí k jejich uložení do vyrovnávací paměti a v nejhorším případě dojde k přečtení celého souboru jednou — nezávisle na tom, z kolika slov tvoříme množné číslo.
3. **Oddělení kódu a dat**. Všechny vzorky jsou uložené v odděleném souboru. Kód je kód, data jsou data a ta dvojice se nikdy nesetká.

> Je to opravdu nirvána? Inu, ano i ne. U příkladu s `LazyRules` musíme počítat s následujícím: soubor se vzorky se otevře (během `__init__()`) a zůstane otevřen, dokud nebude dosaženo posledního pravidla. Soubor se nakonec uzavře při ukončení Pythonu nebo po zrušení poslední instance třídy `LazyRules`, ale může to trvat *velmi dlouho*. Pokud je tato třída součástí dlouho běžícího procesu, nemusí interpret Pythonu skončit nikdy a také objekt třídy `LazyRules` nemusí být nikdy zrušen.
>
> Dá se to obejít různými způsoby. Místo toho, aby byl soubor otevřen během `__init__()` a ponechán v otevřeném stavu pro čtení po jednom řádku, můžeme soubor otevřít, přečíst všechny řádky a soubor hned zavřít. Nebo můžeme soubor otevřít, přečíst jeden řádek s pravidlem, uložit pozici v souboru zjištěnou metodou `tell()` a soubor uzavřít. Později jej znovu otevřeme,

použijeme metodu `seek()` a pokračujeme ve čtení tam, kde jsme skončili. A nebo si s tím nebudeme dělat těžkou hlavu a prostě necháme soubor otevřený, jako to dělá tento příklad. Programování úzce souvisí s návrhem a návrh je založen na kompromisech a omezeních. Pokud bude soubor ponechán v otevřeném stavu příliš dlouho, může to vést k problému. Pokud místo toho vytvoříte komplikovanější kód, může to také vést k problému. Který z těchto problémů je větší, záleží na vašem vývojovém týmu, na vaší aplikaci a na provozním prostředí.

7.7. Přečtěte si

- **Iterator types**
  (http://docs.python.org/py3k/library/stdtypes.html)
- **PEP 234: Iterators**
  (www.python.org/dev/peps/pep-0234/)
- **PEP 255: Simple Generators**
  (www.python.org/dev/peps/pep-0255/)
- **Generator Tricks for Systems Programmers**
  (www.dabeaz.com/generators/)

**“Great fleas have little fleas upon their backs to bite ’em,
And little fleas have lesser fleas, and so ad infinitum.”**
(Veliké blechy maj malé své blechy,
aby je kousaly do jejich zad,
Hle, malé si nesou své o něco menší;
konce to nemá — podivný řád.)

— Augustus De Morgan

8. Iterátory pro pokročilé

8. Iterátory pro pokročilé — 173
8.1. Ponořme se — **175**
8.2. Nalezení všech výskytů vzorku — **176**
8.3. Nalezení jedinečných prvků posloupnosti — **177**
8.4. Činíme předpoklady — **178**
8.5. Generátorové výrazy — **179**
8.6. Výpočet permutací (pro lenochy) — **180**
8.7. Další legrácky v modulu `itertools` — **182**
8.8. Nový způsob úpravy řetězce — **185**
8.9. Vyhodnocování libovolných řetězců zachycujících pythonovské výrazy — **187**
8.10. Spojme to všechno dohromady — **190**
8.11. Přečtěte si — **191**

8.1 Ponořme se

Jestliže přirovnáme regulární výrazy ke steroidům pro řetězce, pak modul `itertools` představuje steroidy pro iterátory. Ale nejdříve si ukážeme jednu klasickou hádanku.

```
HAWAII + IDAHO + IOWA + OHIO == STATES
510199 + 98153 + 9301 + 3593 == 621246

H = 5
A = 1
W = 0
I = 9
D = 8
O = 3
S = 6
T = 2
E = 4
```

Pro hádanky tohoto typu se používají anglické názvy *cryptarithms* nebo *alphametics*. Písmena jsou složena do skutečných slov, ale pokud každé z nich nahradíte číslicí `0-9`, pak tvoří aritmetickou rovnici. Úkol spočívá v nalezení dvojic písmeno/číslice. Všechny výskyty stejného písmene se musí dát nahradit stejnou číslicí. Žádná číslice se nesmí opakovat a žádné „slovo“ nesmí začínat číslicí `0`.

Nejznámější alfametickou hádankou je SEND + MORE = MONEY.

V této kapitole se ponoříme do neuvěřitelného pythonovského programu, který původně napsal Raymond Hettinger. Program řeší alfametické hádanky *na pouhých 14 řádcích kódu*.

```
import re
import itertools

def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
    assert len(unique_characters) <= 10, 'Too many letters'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + \
        ''.join(unique_characters - first_letters)
    characters = tuple(ord(c) for c in sorted_characters)
```

```
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):
        if zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters, guess)))
            if eval(equation):
                return equation

if __name__ == '__main__':
    import sys
    for puzzle in sys.argv[1:]:
        print(puzzle)
        solution = solve(puzzle)
        if solution:
            print(solution)
```

Program můžeme spustit z příkazového řádku. Pod Linuxem to bude vypadat nějak takto. (V závislosti na rychlosti vašeho počítače to může zabrat nějaký čas a není zde žádný indikátor průběhu výpočtu. Buďte trpěliví.)

```
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "HAWAII + IDAHO + IOWA + OHIO == STATES"
HAWAII + IDAHO + IOWA + OHIO = STATES
510199 + 98153 + 9301 + 3593 == 621246
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "I + LOVE + YOU == DORA"
I + LOVE + YOU == DORA
1 + 2784 + 975 == 3760
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "SEND + MORE == MONEY"
SEND + MORE == MONEY
9567 + 1085 == 10652
```

8.2. Nalezení všech výskytů vzorku

Program pro řešení alfametiky ze všeho nejdřív hledá v hádance písmena (A–Z).

```
>>> import re
>>> re.findall('[0-9]+', '16 2-by-4s in rows of 8')      [1]
['16', '2', '4', '8']
>>> re.findall('[A-Z]+', 'SEND + MORE == MONEY')         [2]
['SEND', 'MORE', 'MONEY']
```

[1] Modul `re` implementuje v Pythonu regulární výrazy. Najdeme v něm i šikovnou funkci nazvanou `findall()`, které zadáváme vzorek pro regulární výraz a řetězec. Funkce v zadaném řetězci nalezne všechny výskyty vzorku. V tomto případě vzorek pasuje na posloupnosti číslic. Funkce `findall()` vrací seznam všech podřetězců, které vzorku vyhovují.

[2] Zde regulární výraz popisuje posloupnosti písmen. Návratovou hodnotou je opět seznam, jehož prvky jsou řetězce, které pasovaly k regulárnímu výrazu.

Následuje další příklad, který vám trochu procvičí mozek.

```
>>> re.findall(' s.*? s', "The sixth sick sheikh's sixth sheep's sick.")
[' sixth s', " sheikh's s", " sheep's s"]
```

Tohle je nejtěžší jazykolam, jaký v anglickém jazyce najdete.

Překvapeni? Regulární výraz hledá mezeru, znak `s`, pak nejkratší možnou posloupnost libovolných znaků (`.*?`), pak mezeru a další `s`. Když se tak dívám na vstupní řetězec, vidím pět pasujících podřetězců:

```
The sixth sick sheikh's sixth sheep's sick.
The sixth sick sheikh's sixth sheep's sick.
The sixth sick sheikh's sixth sheep's sick.
The sixth sick sheikh's sixth sheep's sick.
The sixth sick sheikh's sixth sheep's sick.
```

Ale funkce `re.findall()` vrátila jen tři shody. Konkrétně vrátila jen první, třetí a pátou. Proč jen tři? *Protože nevrací překrývající se shody se vzorkem*. První shoda se překrývá s druhou, takže první se vrací a druhá se přeskakuje. Pak se třetí shoda překrývá se čtvrtou, takže třetí se vrací a čtvrtá se přeskakuje. A nakonec je tu pátá shoda, která se vrací. Najdou se tedy tři výskyty a ne pět.

Tahle poznámka neměla s řešením alfametiky nic společného. Prostě mi to připadlo zajímavé.

8.3. Nalezení jedinečných prvků posloupnosti

Jedinečné hodnoty z posloupnosti můžeme snadno najít pomocí množin (set).

```
>>> a_list = ['The', 'sixth', 'sick', "sheik's", 'sixth', "sheep's", 'sick']
>>> set(a_list)                                    [1]
{'sixth', 'The', "sheep's", 'sick', "sheik's"}
>>> a_string = 'EAST IS EAST'
>>> set(a_string)                                  [2]
{'A', ' ', 'E', 'I', 'S', 'T'}
>>> words = ['SEND', 'MORE', 'MONEY']
>>> ''.join(words)                                 [3]
'SENDMOREMONEY'
>>> set(''.join(words))                            [4]
{'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

[1] Pokud máme seznam s několika řetězci, pak nám z něj funkce `set()` vytvoří množinu jedinečných řetězců. Dá se to snadno pochopit, když si to představíte jako cyklus `for`. Vezmeme první položku ze seznamu a vložíme ji do množiny. Pak druhou. A třetí. Čtvrtou. Pátou... Počkat! Ta už v množině je, takže se bude vypisovat jen jednou, protože množiny v Pythonu neumožňují existenci duplicit. A šestou. Sedmou — a znovu duplicita, takže se pak objeví jen jednou. A jaký je konečný výsledek? Z původního seznamu zbyly jen jedinečné položky bez duplicit. Původní seznam ani nemusíme předem seřadit.

[2] Stejná technika funguje i pro řetězce, protože řetězce jsou posloupnostmi znaků.

[3] Pokud máme seznam řetězců, pak `''.join(a_list)` spojí všechny řetězce do jednoho.

[4] Takže pokud máme seznam řetězců, tento řádek kódu vrátí jedinečné znaky nacházející se ve všech řetězcích. Bez duplicit.

Program pro řešení alfametik tuto techniku používá pro vytvoření množiny všech jedinečných znaků v zadání.

```
unique_characters = set(''.join(words))
```

Program postupně prochází všemi možnými řešeními a tuto množinu používá pro přiřazení číslic jednotlivým znakům.

8.4. Činíme předpoklady

V Pythonu, stejně jako v mnoha jiných programovacích jazycích, najdeme příkaz `assert`. Funguje následovně.

```
>>> assert 1 + 1 == 2                                          [1]
>>> assert 1 + 1 == 3                                          [2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 2 + 2 == 5, "Only for very large values of 2"       [3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Only for very large values of 2
```

[1] Za příkaz `assert` uvedeme libovolný platný pythonovský výraz. V tomto případě se výraz `1 + 1 == 2` vyhodnotí jako `True`, takže příkaz `assert` nedělá nic.

[2] Pokud se ale pythonovský výraz vyhodnotí jako `False`, vyvolá příkaz `assert` výjimku `AssertionError`.

[3] Za výraz můžeme uvést také lidsky čitelnou zprávu, která se v případě vyvolání výjimky `AssertionError` zobrazí.

Takže následující řádek kódu:

```
assert len(unique_characters) <= 10, 'Too many letters'
```

…je ekvivalentem zápisu:

```
if len(unique_characters) > 10:
    raise AssertionError('Too many letters')
```

Program řešící alfametiku používá přesně takový příkaz `assert` k předčasnému ukončení činnosti v případě, kdy hádanka obsahuje víc než deset jedinečných znaků. Protože každému písmenu přiřazujeme jedinečnou číslici a číslic máme jen deset, hádanka s více než deseti jedinečnými znaky nemůže mít řešení.

8.5. Generátorové výrazy

Generátorový výraz se podobá generátorové funkci, ale funkce to není.

```
>>> unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
>>> gen = (ord(c) for c in unique_characters)    [1]
>>> gen                                          [2]
<generator object <genexpr> at 0x00BADC10>
>>> next(gen)                                    [3]
69
>>> next(gen)
68
>>> tuple(ord(c) for c in unique_characters)     [4]
(69, 68, 77, 79, 78, 83, 82, 89)
```

[1] Generátorový výraz se chová jako anonymní funkce, která produkuje hodnoty. Výraz samotný se podobá generátorové notaci seznamu (list comprehension), ale místo do hranatých závorek je uzavřen v kulatých závorkách.

[2] Generátorový výraz vrací… iterátor.

[3] Při volání `next(gen)` se nám vrací další hodnota iterátoru.

[4] Pokud chcete, můžete iterovat přes všechny hodnoty a vrátit n-tici, seznam nebo množinu tím, že generátorový výraz použijete v roli argumentu `tuple()`, `list()` nebo `set()`. V takovém případě nemusíte používat sadu kulatých závorek navíc. Funkci `tuple()` stačí předat „holý“ výraz `ord(c) for c in unique_characters` a Python už pozná, že jde o generátorový výraz.

> Když místo generátorové notace seznamu použijete generátorový výraz, ušetříte jak CPU, tak RAM. Pokud konstruujete seznam jen proto, abyste ho zase zahodili (tj. když ho například chcete předat do `tuple()` nebo `set()`), použijte raději generátorový výraz!

Následující ukázka dosahuje stejného efektu s použitím generátorové funkce:

```
def ord_map(a_string):
    for c in a_string:
        yield ord(c)

gen = ord_map(unique_characters)
```

Generátorový výraz je kompaktnější, ale funguje stejně.

8.6. Výpočet permutací (pro lenochy)

Ze všeho nejdříve se podívejme, co to vlastně jsou permutace? Permutace jsou matematický koncept. (Ve skutečnosti existuje několik definicí v závislosti na tom, jakým druhem matematiky se zabýváte. Zde se dotkneme kombinatoriky. Ale pokud vám to nic neříká, nedělejte si s tím starosti. Tak jako vždy, vaším kamarádem je Wikipedie.)

Základní myšlenka spočívá v tom, že vezmeme seznam věcí (mohou to být čísla, písmenka nebo tancující medvídci) a najdeme všechny možné způsoby, jak z něj udělat menší seznamy. (Poznámka překladatele: V našich školách se pro označení tohoto úkonu používá pojem variace k-té třídy z n prvků bez opakování. Pojem permutace bez opakování se u nás používá jen pro speciální případ, kdy k je rovno n. V dalším textu zůstanu u chápání pojmu z originální publikace také z důvodu pojmenování příslušné funkce.) Všechny menší seznamy mají mít stejnou velikost, která může být od 1 až po celkový počet prvků. A nic se nesmí opakovat. Matematici by řekli „najděme permutace dvojic z tří různých prvků“ (u nás „najděte variace druhé třídy z tří prvků bez opakování“). To znamená, že máme posloupnost tří prvků a chceme nalézt všechny možné uspořádané dvojice.

```
>>> import itertools                                    [1]
>>> perms = itertools.permutations([1, 2, 3], 2)       [2]
>>> next(perms)                                         [3]
(1, 2)
>>> next(perms)
(1, 3)
>>> next(perms)
(2, 1)                                                  [4]
>>> next(perms)
(2, 3)
>>> next(perms)
(3, 1)
>>> next(perms)
(3, 2)
>>> next(perms)                                         [5]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

[1] Modul `itertools` obsahuje celou řadu zábavných věcí, včetně funkce `permutations()`, která nás při hledání permutací zbaví veškeré námahy.

[2] Funkce `permutations()` přebírá posloupnost (zde jde o seznam tří čísel) a požadovaný počet prvků v menších skupinách. Funkce vrací iterátor, který můžeme použít v cyklu `for` nebo na jakémkoliv starém známém místě, ve kterém se iteruje (tj. prochází všemi prvky). Zde budeme provádět kroky iterátoru ručně, abychom si všechny hodnoty ukázali.

[3] První permutací ze seznamu `[1, 2, 3]` je dvojice `(1, 2)`.

[4] Poznamenejme, že permutace jsou uspořádané: `(2, 1)` je něco jiného než `(1, 2)`.

[5] Tak to jsou ony! Tohle jsou permutace všech dvojic z `[1, 2, 3]`. Dvojice jako `(1, 1)` nebo `(2, 2)` zde nikdy neuvidíte, protože obsahují opakující se prvky. Takže nejde o platné permutace. Pokud už více permutací neexistuje, iterátor vyvolá výjimku `StopIteration`.

Modul itertools obsahuje všemožné zábavné věci.

Funkci `permutations()` nemusíme předávat jen seznam. Může přebírat jakoukoliv posloupnost, dokonce i řetězec.

```
>>> import itertools
>>> perms = itertools.permutations('ABC', 3)     [1]
>>> next(perms)
('A', 'B', 'C')                                  [2]
>>> next(perms)
('A', 'C', 'B')
>>> next(perms)
('B', 'A', 'C')
>>> next(perms)
('B', 'C', 'A')
>>> next(perms)
('C', 'A', 'B')
>>> next(perms)
('C', 'B', 'A')
>>> next(perms)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> list(itertools.permutations('ABC', 3))       [3]
[('A', 'B', 'C'), ('A', 'C', 'B'),
 ('B', 'A', 'C'), ('B', 'C', 'A'),
 ('C', 'A', 'B'), ('C', 'B', 'A')]
```

[1] Řetězec je jen posloupností znaků. Takže pro účely hledání permutací je řetězec `'ABC'` ekvivalentem k seznamu `['A', 'B', 'C']`.

[2] První permutací trojic z tří prvků `['A', 'B', 'C']` je `('A', 'B', 'C')`. Pro stejné znaky existuje pět dalších myslitelných uspořádání, tedy permutací.

[3] Funkce `permutations()` vrací vždy iterátor. Snadný způsob zviditelnění všech permutací při ladění spočívá ve vytvoření jejich seznamu předáním iterátoru do zabudované funkce `list()`.

8.7. Další legrácky v modulu `itertools`

```
>>> import itertools
>>> list(itertools.product('ABC', '123'))        [1]
[('A', '1'), ('A', '2'), ('A', '3'),
 ('B', '1'), ('B', '2'), ('B', '3'),
 ('C', '1'), ('C', '2'), ('C', '3')]
>>> list(itertools.combinations('ABC', 2))       [2]
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

[1] Funkce `itertools.product()` vrací iterátor, který vytváří kartézský součin dvou posloupností.

[2] Funkce `itertools.combinations()` vrací iterátor, který vytváří všechny možné kombinace dané délky z dané posloupnosti. Podobá se funkci `itertools.permutations()` s tou výjimkou, že kombinace nezahrnují výsledky, které vzniknou pouhou změnou uspořádání položek jiného výsledku. Takže `itertools.permutations('ABC', 2)` vrátí jak `('A', 'B')`, tak `('B', 'A')` (mimo jiné), ale `itertools.combinations('ABC', 2)` nevrátí `('B', 'A')`, protože jde o duplicitu vytvořenou změnou pořadí položek `('A', 'B')`.

```
>>> names = list(open('examples/favorite-people.txt', encoding='utf-8'))          [1]
>>> names
['Dora\n', 'Ethan\n', 'Wesley\n', 'John\n', 'Anne\n',
'Mike\n', 'Chris\n', 'Sarah\n', 'Alex\n', 'Lizzie\n']
>>> names = [name.rstrip() for name in names]                                      [2]
>>> names
['Dora', 'Ethan', 'Wesley', 'John', 'Anne',
'Mike', 'Chris', 'Sarah', 'Alex', 'Lizzie']
>>> names = sorted(names)                                                          [3]
>>> names
['Alex', 'Anne', 'Chris', 'Dora', 'Ethan',
'John', 'Lizzie', 'Mike', 'Sarah', 'Wesley']
>>> names = sorted(names, key=len)                                                 [4]
>>> names
['Alex', 'Anne', 'Dora', 'John', 'Mike',
'Chris', 'Ethan', 'Sarah', 'Lizzie', 'Wesley']
```

[1] Tento obrat vrací seznam všech řádků v textovém souboru.

[2] Naneštěstí (pro tento příklad) obrat `list(open(filename))` vrací na konci každého řádku i znak konce řádku. V této generátorové notaci seznamu použijeme metodu řetězce `rstrip()`, která z konce každého řádku odstraní koncové bílé znaky. (Řetězce definují též metodu `lstrip()`, která odstraňuje úvodní bílé znaky, a metodu `strip()`, která odstraňuje bílé znaky z obou konců.)

[3] Funkce `sorted()` přebírá seznam a vrací nový, seřazený. Neřekneme-li jinak, řadí se podle abecedy.

[4] Ale funkci `sorted()` můžeme parametrem `key` předat funkci a pak se provede řazení podle jejích výsledků. V tomto případě byla předána funkce `len()`, takže řazení probíhá podle výsledků funkce `len(položka)`. Nejkratší jména se dostanou na začátek, pak budou následovat delší a delší.

A co to má společného s modulem `itertools`? To jsem rád, že se ptáte.

```
…pokračování v předchozí práci s interaktivním shellem…
>>> import itertools
>>> groups = itertools.groupby(names, len)              [1]
>>> groups
<itertools.groupby object at 0x00BB20C0>
>>> list(groups)
[(4, <itertools._grouper object at 0x00BA8BF0>),
 (5, <itertools._grouper object at 0x00BB4050>),
 (6, <itertools._grouper object at 0x00BB4030>)]
>>> groups = itertools.groupby(names, len)              [2]
>>> for name_length, name_iter in groups:               [3]
...     print('Names with {0:d} letters:'.format(name_length))
...     for name in name_iter:
...         print(name)
...
Names with 4 letters:
Alex
Anne
Dora
John
Mike
Names with 5 letters:
Chris
Ethan
Sarah
Names with 6 letters:
Lizzie
Wesley
```

[1] Fukce `itertools.groupby()` přebírá posloupnost a funkci klíče. Vrací iterátor, který vytváří dvojice. Každá dvojice obsahuje jednak výsledek `funkce_klic(každá položka)` a jednak další iterátor, který prochází všemi položkami se stejným výsledkem funkce klíče.

[2] Voláním funkce `list()` jsme iterátor „vyčerpali“. To znamená, že jsme při vytváření seznamu vygenerovali každou položku iterátoru. Iterátor nemá žádné tlačítko „reset“. Jakmile jsme posloupnost jednou vyčerpali, nemůžeme začít znovu. Pokud chceme hodnoty projít znovu (dejme tomu v dalším cyklu `for`), musíme znovu zavolat `itertools.groupby()` a vytvořit nový iterátor.

[3] Za předpokladu, že už máme seznam jmen *seřazený podle jejich délek*, přidělí `itertools.groupby(names, len)` všem jménům délky `4` jeden iterátor, všem jménům délky `5` druhý iterátor atd. Funkce `groupby()` je zcela obecná. Řetězce můžeme seskupit podle prvního písmene, čísla podle počtu jejich prvočinitelů nebo podle jakékoliv myslitelné funkce klíče.

> Funkce `itertools.groupby()` funguje jen v případě, kdy je vstupní posloupnost již seřazená podle sdružovací funkce. Ve výše uvedeném příkladu jsme seznam jmen seskupili podle funkce `len()`. Fungovalo to jen díky tomu, že byl vstupní seznam již seřazen podle délky položek.

Díváte se pozorně?

```
>>> list(range(0, 3))
[0, 1, 2]
>>> list(range(10, 13))
[10, 11, 12]
>>> list(itertools.chain(range(0, 3), range(10, 13)))            [1]
[0, 1, 2, 10, 11, 12]
>>> list(zip(range(0, 3), range(10, 13)))                        [2]
[(0, 10), (1, 11), (2, 12)]
>>> list(zip(range(0, 3), range(10, 14)))                        [3]
[(0, 10), (1, 11), (2, 12)]
>>> list(itertools.zip_longest(range(0, 3), range(10, 14)))      [4]
[(0, 10), (1, 11), (2, 12), (None, 13)]
```

[1] Funkce `itertools.chain()` přebírá dva iterátory a vrací iterátor, který vytváří posloupnost všech položek nejdříve z prvního iterátoru a pak všech položek z druhého iterátoru. (Ve skutečnosti můžeme předat libovolný počet iterátorů a tato funkce zřetězí všechny jejich hodnoty v pořadí, v jakém jsme je funkci předali.)

[2] Funkce `zip()` dělá něco docela obyčejného, ale ukazuje se, že je velmi užitečná. Přebírá libovolný počet posloupností a vrací iterátor, který vytváří n-tice z prvních položek každé posloupnosti, pak z druhých položek, pak z třetích atd.

[3] Funkce `zip()` zastaví na konci nejkratší posloupnosti. Funkce `range(10, 14)` produkuje 4 položky (10, 11, 12 a 13), ale `range(0, 3)` jen 3. Takže funkce `zip()` vrátí iterátor produkující 3 položky.

[4] Naopak funkce `itertools.zip_longest()` zastaví až na konci *nejdelší* posloupnosti. Místo chybějících položek kratších posloupností doplní hodnoty `None`.

No dobrá, tohle všechno je sice velmi zajímavé, ale jak se to vztahuje k programu na řešení alfametik? Takto:

```
>>> characters = ('S', 'M', 'E', 'D', 'O', 'N', 'R', 'Y')
>>> guess = ('1', '2', '0', '3', '4', '5', '6', '7')
>>> tuple(zip(characters, guess))                          [1]
(('S', '1'), ('M', '2'), ('E', '0'), ('D', '3'),
 ('O', '4'), ('N', '5'), ('R', '6'), ('Y', '7'))
>>> dict(zip(characters, guess))                           [2]
{'E': '0', 'D': '3', 'M': '2', 'O': '4',
 'N': '5', 'S': '1', 'R': '6', 'Y': '7'}
```

[1] Máme-li dán seznam písmen a seznam číslic (každá z nich je v něm reprezentována jako jednoznakový řetězec), pak nám funkce `zip` páruje písmena a číslice v uvedeném pořadí.

[2] A proč by to mělo být nějak zvlášť výhodné? Protože shodou okolností je taková datová struktura přesně tou správnou datovou strukturou, kterou můžeme předat funkci `dict()`, aby vytvořila slovník, který používá písmena jako klíče a k nim přidružené číslice jako hodnoty. (Není to, samozřejmě, jediný způsob, jak toho můžeme dosáhnout. Slovník bychom mohli vytvořit přímo, pomocí generátorové notace.) Ačkoliv textová reprezentace obsahu slovníku zobrazuje dvojice v jiném pořadí (slovníky samy o sobě nedefinují „pořadí“), vidíme, že každé písmeno má k sobě číslici přidruženou na základě původních posloupností `characters` a `guess`.

Program pro řešení alfametik tuto techniku používá pro vytvoření slovníku, který převádí písmena z hádanky na čísla v řešení — pro každé možné řešení.

```
characters = tuple(ord(c) for c in sorted_characters)
digits = tuple(ord(c) for c in '0123456789')
...
for guess in itertools.permutations(digits, len(characters)):
    ...
    equation = puzzle.translate(dict(zip(characters, guess)))
```

Ale co je za metodu ta `translate()`? Teď se dostáváme k *opravdu* zábavné části.

8.8. Nový způsob úpravy řetězce

Teď se dostáváme k *opravdu* zábavné části.

Pythonovské řetězce definují mnoho metod. O některých z nich jsme se učili v kapitole Řetězce: `lower()`, `count()` a `format()`. Teď si představíme mocnou, ale málo známou techniku pro manipulaci s řetězcem. Jde o metodu `translate()`.

```
>>> translation_table = {ord('A'): ord('O')}              [1]
>>> translation_table                                     [2]
{65: 79}
>>> 'MARK'.translate(translation_table)                   [3]
'MORK'
```

[1] Překlad řetězce začíná naplněním překladové tabulky, což je prostě slovník, který zobrazuje jeden znak na jiný. Pojem „znak“ je zde vlastně uveden chybně. Překladová tabulka ve skutečnosti zobrazuje bajty na jiné bajty.

[2] Připomeňme si, že bajty jsou v Pythonu 3 celá čísla. Funkce `ord()` vrací `ASCII` hodnotu daného znaku. V případě znaků A–Z to budou vždy bajty od 65 do 90.

[3] Metoda řetězcového objektu `translate()` přebírá překladovou tabulku a obsah řetězce přes ni propasíruje. To znamená, že nahradí všechny výskyty klíčů z překladové tabulky odpovídajícími hodnotami. V tomto případě se `MARK` „přeloží“ na `MORK`.

Ale co to má společného s řešením alfametických hádanek? Jak se ukáže za chvílli, všechno.

```
>>> characters = tuple(ord(c) for c in 'SMEDONRY')       [1]
>>> characters
(83, 77, 69, 68, 79, 78, 82, 89)
>>> guess = tuple(ord(c) for c in '91570682')             [2]
>>> guess
(57, 49, 53, 55, 48, 54, 56, 50)
>>> translation_table = dict(zip(characters, guess))      [3]
>>> translation_table
{68: 55, 69: 53, 77: 49, 78: 54, 79: 48, 82: 56, 83: 57, 89: 50}
>>> 'SEND + MORE == MONEY'.translate(translation_table)   [4]
'9567 + 1085 == 10652'
```

[1] Prostřednictvím generátorového výrazu pro každý znak řetězce rychle vypočteme hodnotu odpovídajícího bajtu. Obsah proměnné `characters` je příkladem obsahu proměnné `sorted_characters` z funkce `alphametics.solve()`.

[2] Pomocí dalšího generátorového výrazu rychle vypočítáme hodnoty bajtů reprezentujících každou číslici řetězce. Výsledek v proměnné `guess` (tj. odhad) má podobu vrácenou funkcí `itertools.permutations()` — viz funkce `alphametics.solve()`

[3] Překladová tabulka se generuje zipováním posloupností `characters` a `guess` dohromady a použitím výsledné posloupnosti dvojic pro vybudování slovníku. Přesně tohle dělá funkce `alphametics.solve()` uvnitř cyklu `for`.

[4] Nakonec překladovou tabulku předáme metodě `translate()` původního řetězce hádanky. Tím se každý znak řetězce přeloží na odpovídající číslici (podle písmen v `characters` a číslic v `guess`). Výsledkem je platný pythonovský výraz v řetězcové podobě.

To je docela efektní. Ale co můžeme dělat s řetězcem, který shodou okolností zachycuje platný pythonovský výraz?

8.9. Vyhodnocování libovolných řetězců zachycujících pythonovské výrazy

Tohle je poslední kousek skládanky (nebo spíše poslední kousek programu pro řešení hádanky). Po všech těch efektních manipulacích s řetězci jsme skončili u řetězce, jako je `'9567 + 1085 == 10652'`. Ale je to jen řetězec. A k čemu je nám řetězec dobrý? Seznamte se s `eval()`, s univerzálním pythonovským vyhodnocovacím nástrojem.

```
>>> eval('1 + 1 == 2')
True
>>> eval('1 + 1 == 3')
False
>>> eval('9567 + 1085 == 10652')
True
```

Ale počkejte! Je toho ještě víc! Funkce `eval()` se neomezuje jen na booleovské výrazy. Zvládne *libovolný* pythonovský výraz a vrací *libovolný* datový typ.

```
>>> eval('"A" + "B"')
'AB'
>>> eval('"MARK".translate({65: 79})')
'MORK'
>>> eval('"AAAAA".count("A")')
5
>>> eval('["*"] * 5')
['*', '*', '*', '*', '*']
```

Ale počkejte, to ještě není vše!

```
>>> x = 5
>>> eval("x * 5")                [1]
25
>>> eval("pow(x, 2)")            [2]
25
>>> import math
>>> eval("math.sqrt(x)")         [3]
2.2360679774997898
```

[1] Výraz předávaný funkci `eval()` se může odkazovat na globální proměnné definované vně `eval()`. A pokud se volá uvnitř funkce, může se odkazovat i na lokální proměnné.

[2] A funkce.

[3] A moduly.

Hej, zastav na minutku…

```
>>> import subprocess
>>> eval("subprocess.getoutput('ls ~')")                                    [1]
'Desktop         Library         Pictures \
 Documents       Movies          Public   \
 Music           Sites'
>>> eval("subprocess.getoutput('rm /some/random/file')")                    [2]
```

[1] Modul `subprocess` vám dovolí spustit libovolný shellovský příkaz a získat výsledek v podobě pythonovského řetězce.

[2] Jenže libovolný shellovský příkaz může vést k trvalým následkům.

A je to dokonce ještě horší, protože existuje globální funkce `__import__()`, která přebírá jméno modulu v řetězcové podobě, importuje ho a vrací na něj odkaz. Když to zkombinujeme se silou funkce `eval()`, můžeme vytvořit výraz, který smaže všechny vaše soubory:

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')")    [1]
```

[1] A teď si představte výstup příkazu `'rm -rf ~'`. Ve skutečnosti žádný výstup neuvidíte. Ale neuvidíte už ani své soubory.

eval() is EVIL

(tj. `eval()` je zlý, špatný, zlověstný). Tou zlou stránkou je vyhodnocování libovolných výrazů pocházejících z nedůvěryhodných zdrojů. Funkci `eval()` byste měli používat výhradně pro vstup z důvěryhodných zdrojů. Problém je v tom, jak určit, co je „důvěryhodný“ zdroj. Ale něco vím určitě. Určitě byste **NEMĚLI** vzít tento program pro řešení alfametik a zveřejnit jej na internetu v podobě malé webovské služby. A nemyslete si: „Vždyť ta funkce dělá tolik řetězcových operací, než se vůbec dostane k vyhodnocení. Nedovedu si představit, jak by toho někdo mohl zneužít.“ Někdo **přijde** na to, jak propašovat nějaký nebezpečný kód všemi těmi řetězcovými manipulacemi (už se staly divnější věci). A pak už můžete svému serveru poslat jen polibek na rozloučenou.

Ale existuje vůbec nějaký způsob, jak výrazy vyhodnotit bezpečně? Lze nějak `eval()` umístit na pískoviště, odkud nemá přístup k okolnímu světu a nemůže mu škodit? Hmm, ano i ne.

```
>>> x = 5
>>> eval("x * 5", {}, {})                          [1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
```

```
NameError: name 'x' is not defined
>>> eval("x * 5", {"x": x}, {})                    [2]
25
>>> import math
>>> eval("math.sqrt(x)", {"x": x}, {})             [3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'math' is not defined
```

[1] Druhý a třetí parametr předávaný funkci `eval()` se chovají jako globální a lokální prostor jmen. Tyto prostory se používají při vyhodnocování výrazu. V tomto případě jsou oba prázdné. To znamená, že při vyhodnocování řetězce `"x * 5"` neexistuje žádný odkaz na `x` ani v globálním ani v lokálním prostoru jmen. Takže `eval()` vyvolá výjimku.

[2] Do globálního prostoru jmen můžeme vložit výběr určitých hodnot tím, že je jednotlivě vyjmenujeme. Během vyhodnocování pak budou k dispozici tyto a jen tyto proměnné.

[3] Ačkoliv jste zrovna importovali modul math, nevložili jsme jej do prostoru jmen, který předáváme funkci `eval()`. V důsledku toho vyhodnocení selhalo.

Tý jo. Tak to bylo jednoduché. Teď si udělám webovskou službu pro řešení alfametik!

```
>>> eval("pow(5, 2)", {}, {})                      [1]
25
>>> eval("__import__('math').sqrt(5)", {}, {})     [2]
2.2360679774997898
```

[1] Ačkoliv jste v roli globálního a lokálního prostoru jmen předali prázdné slovníky, během vyhodnocování jsou stále dostupné všechny zabudované pythonovské funkce. Takže `pow(5, 2)` funguje, protože `5` a `2` jsou literály a `pow()` je zabudovaná funkce.

[2] Naneštěstí (a pokud netušíte, proč naneštěstí, čtěte dál) je funkce `__import__()` také zabudovanou funkcí, takže také funguje.

Ano, to znamená, že můžete pořád dělat odporné věci, i když jste při volání `eval()` pro globální a lokální prostor jmen explicitně nastavili prázdné slovníky:

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')", {}, {})
```

A do prčic! Jsem rád, že jsem z modulu alphametics neudělal webovou službu. Je zde vůbec *nějaký* způsob, kterým bychom mohli `eval()` používat bezpečně? Ano i ne.

```
>>> eval("__import__('math').sqrt(5)",
...     {"__builtins__":None}, {})              [1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
>>> eval("__import__('subprocess').getoutput('rm -rf /')",
...     {"__builtins__":None}, {})              [2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

[1] Abyste mohli výrazy z nedůvěryhodných zdrojů vyhodnocovat bezpečně, musíte definovat slovník pro globální prostor jmen, který mapuje `"__builtins__"` na `None`, tedy na pythonovskou hodnotu null (nic, nil). „Zabudované“ funkce jsou totiž vnitřně uzavřeny do pseudomodulu nazvaného `"__builtins__"`. Tento pseudomodul (tj. množina zabudovaných funkcí) je vyhodnocovaným výrazům zpřístupněn — pokud jej explicitně nepotlačíte.

[2] Ujistěte se, že předefinováváte `__builtins__`. Žádné `__builtin__`, `__built-ins__` nebo nějakou podobnou variantu. Ono by to fungovalo bez problémů, ale vystavilo by vás to riziku katastrofy.

Takhle už je `eval()` bezpečný? Nu, ano i ne.

```
>>> eval("2 ** 2147483647",
...     {"__builtins__":None}, {})              [1]
```

[1] I bez přístupu k `__builtins__` můžete stále spustit útok typu odmítnutí služby. Pokud se například pokusíte o umocnění `2` na `2147483647`, využití procesoru vašeho serveru stoupne na 100 % na pěkně dlouhou dobu. (Pokud to zkoušíte v interaktivním shellu, můžete ho přerušit, když několikrát stisknete `Ctrl-C`.) Technicky vzato, tento výraz *nakonec vrátí* nějakou hodnotu, ale do té doby bude server dělat spoustu zbytečné práce.

Takže nakonec *je možné* bezpečně vyhodnocovat pythonovské výrazy z nedůvěryhodných zdrojů. Vyžaduje to ale určitou definici pojmu „bezpečně“, která v reálném životě není zas tak užitečná. Dobré je, když si hrajete někde poblíž. Dobré taky je, když připustíte jen důvěryhodný vstup. Cokoliv jiného znamená, že si koledujete o malér.

8.10. Spojme to všechno dohromady

Rekapitulace: Tento program řeší alfametické hádanky hrubou silou, tj. vyčerpávajícím hledáním všech možných řešení. Program za tím účelem...

1. **Nalezne v zadání všechna písmena** voláním funkce `re.findall()`.
2. **Nalezne všechna *jedinečná* písmena hádanky s využitím množiny a funkce** `set()`.
3. **Zkontroluje příkazem** `assert`**, zda se v zadání nevyskytuje více než 10 jedinečných znaků** (což by znamenalo, že hádanka je neřešitelná).
4. **Převede znaky na jejich** `ASCII` **hodnoty použitím objektu generátoru**.
5. **Počítá všechna možná řešení** pomocí funkce `itertools.permutations()`.
6. **Převádí každé možné řešení na pythonovský výraz** pomocí metody řetězcového objektu `translate()`.
7. **Testuje každé možné řešení vyhodnocením pythonovského výrazu** voláním funkce `eval()`.
8. Vrací první řešení, které se vyhodnotí jako `True`.

…to vše na pouhých 14 řádcích kódu.

8.11. Přečtěte si

- **`itertools` module**
  (http://docs.python.org/py3k/library/itertools.html)
- **`itertools` — Iterator functions for efficient looping**
  (www.doughellmann.com/PyMOTW/itertools/)
- **Podívejte se na přednášku Raymonda Hettingera „Easy AI with Python“** na PyCon 2009
  (www.doughellmann.com/PyMOTW/itertools/)
- **Recipe 576615: Alphametics solver**, původní program Raymonda Hettingera pro Python 2.
  (http://code.activestate.com/recipes/576615/)
- **Další recepty od Raymonda Hettingera** v ActiveState Code repository (archiv kódu).
  (http://code.activestate.com/recipes/users/178123/)
- **Alphametics on Wikipedia**
  (http://en.wikipedia.org/wiki/Verbal_arithmetic)
- **Alphametics Index**, včetně **mnoha zadání** a **generátoru vašich vlastních zadání.**
  (www.tkcs-collins.com/truman/alphamet/index.shtml)

Mnohokrát děkuji Raymondu Hettingerovi za souhlas s úpravou licence jeho kódu, abych ho mohl přepsat pro Python 3 a použít jako základ této kapitoly.

> **“Certitude is not the test of certainty. We have been cocksure of many things that were not so.”**
> (Pocit jistoty není měřítkem jistoty. Byli jsme si skálopevně jisti mnoha věcmi, které takové nebyly.)
>
> — Oliver Wendell Holmes, Jr.

9. Unit Testing

9. Unit Testing — 193
9.1. (Ne)ponořme se — **195**
9.2. Jediná otázka — **196**
9.3. „Zastav a začni hořet“ — **202**
9.4. Více zastávek, více ohně — **206**
9.5. A ještě jedna věc… — **209**
9.6. Symetrie, která potěší — **211**
9.7. Více špatných vstupů — **215**

9.1. (Ne)ponořme se

Ta dnešní mládež. Jsou tak zkažení těmi rychlými počítači a módními „dynamickými“ jazyky. Rychle napsat, pak dodat a ladit až nakonec (jestli vůbec). Za mých časů jsme dodržovali disciplínu. Říkám disciplínu! Museli jsme psát programy *ručně*, na *papír* a cpát je do počítače na *děrných štítcích*. A ono se nám to *líbilo*! A cože? Že je ten nadpis anglicky? Buďte rádi, že není v ruštině. Mnozí z vás ani neví, jak přečíst jednotlivá písmenka azbuky. No dobrá, trochu zvážním. Dá se to přeložit jako „testování jednotek“ nebo „jednotkové testování“. Ještě se k tomu dostaneme.

V této kapitole si napíšeme a odladíme pár pomocných funkcí pro konverzi na a z římských čísel. Způ-
Kap. sob tvorby a ověřování římských čísel jsme si ukázali v podkapitole Případová studie: Římská čísla. Teď si poodstoupíme a zvážíme, kolik by dalo práce rozšířit původní kód na obousměrné pomocné funkce.

Pravidla pro římská čísla vedla k řadě zajímavých postřehů:

1. Existuje jen jeden správný způsob vyjádření konkrétního čísla římskými číslicemi.
2. Platí také opak. Pokud je řetězec znaků platným římským číslem, reprezentuje jen jedno možné číslo (to znamená, že řetězec může být interpretován jen jedním způsobem).
3. Římskými čísly lze vyjádřit jen omezený rozsah čísel, konkrétně od `1` do `3999`. Římané používali několik způsobů vyjádření větších čísel. Tak například pruhem nad římským číslem vyjadřovali, že jeho číselná hodnota musí být vynásobená tisícem. Pro účely této kapitoly budeme uvažovat jen římská čísla od `1` do `3999`.
4. Neexistuje způsob, jak římskými číslicemi vyjádřit nulu.
5. Neexistuje způsob, jak římskými číslicemi vyjádřit záporná čísla.
6. Neexistuje způsob, jak římskými číslicemi vyjádřit zlomky nebo neceločíselné hodnoty.

Začněme mapovat, co by takový modul roman.py měl dělat. Bude obsahovat dvě hlavní funkce, `to_roman()` (na římské číslo) a `from_roman()` (z římského čísla). Funkce `to_roman()` by měla převzít celé číslo v intervalu od `1` do `3999` a vrátit jeho reprezentaci římskými číslicemi jako řetězec...

Hned tady se zastavíme. Teď uděláme něco trošku neočekávaného. Napíšeme si testovací příklad, který kontroluje, zda funkce `to_roman()` dělá to, co po ní chceme. Čtete dobře. Jdeme psát kód, který testuje jiný kód, který jsme ještě nenapsali.

Říká se tomu *vývoj řízený testy (test-driven development)* nebo `TDD`. (V anglické literatuře si potrpí na zavádění a používání zkratek.) Dvojice převodních funkcí — `to_roman()` a později `from_roman()` — může být napsána a testována jako *jednotka* (unit), odděleně od jakéhokoliv většího programu, který funkce importuje. V Pythonu najdeme rámec (framework) pro unit testing (tedy testování jednotek), který má podobu příhodně nazvaného modulu `unittest`.

Unit testing (testování jednotek) představuje důležitou součást celkové vývojové strategie založené na testování. Pokud testy jednotek píšete, je důležité, abyste je napsali brzy a abyste je udržovali v závislosti na změnách kódu a požadavků. Mnozí lidé se přimlouvají za to, aby se testy psaly dříve

než kód, který mají testovat. V této kapitole si takový přístup předvedeme. Ale testy jednotek mají své výhody nezávisle na tom, kdy je napíšete.

- Napsání jednotkových testů (i takto se to dá překládat) ještě před napsáním kódu vás účelným způsobem donutí upřesnit své požadavky
- Při vlastním psaní kódu vás pak jednotkové testy brzdí před psaním nadbytečných věcí. Jakmile všechny testy projdou, dosáhli jste úplné funkčnosti.
- Při provádění refaktorizace kódu vám testy jednotek pomohou prokázat, že se nová verze chová stejným způsobem jako ta stará.
- Při údržbě kódu vám existence testů pomůže krýt záda (v originále se mluví o té části těla, kde záda ztrácejí své slušné jméno) v situaci, kdy na vás někdo přiletí a řve, že vaše poslední změny pokazily jejich původní kód. („Ale *pane*, ale když jsem změny zveřejňoval, všechny unit testy prošly...“)
- Pokud píšeme kód v týmu, pak existence společné sady testů dramaticky snižuje možnost, že by váš kód způsobil nefunkčnost kódu někoho jiného. Jejich testy jednotek totiž můžete spustit jako první. (Tenhle druh závodů v psaní kódu už jsem zažil. Tým si zadání rozdělí, každý si převezme specifikace svého úkolu, napíše pro něj jednotkové testy a pak je dá k dispozici ostatním členům týmu. Při takovém postupu nikdo nezabloudí tak daleko, že by jím vyvíjený kód nespolupracoval s výsledky ostatních.)

9.2. Jediná otázka

Každý test je ostrov.

Testovací případ (test case) odpovídá na jedinou otázku, která se testovaného kódu týká. Testovací případ by měl být schopen...

- ...běžet zcela samostatně, bez jakéhokoliv lidského zásahu. Unit testing (testování jednotek) souvisí s automatizací.
- ...sám rozhodnout o tom, zda testovaná funkce prošla nebo selhala — bez nutnosti posuzování výsledků člověkem.
- ...běžet izolovaně, odděleně od jakýchkoliv jiných testovacích případů (dokonce i když testují stejnou funkci). Každý testovací případ je ostrov.

S ohledem na uvedené předpoklady začněme budovat testovací případ pro první požadavek:

> Funkce `to_roman()` by měla vracet reprezentaci římského čísla pro všechna celá čísla v intervalu `1` až `3999`.

V prvním okamžiku není zřejmé, jak následující kód dělá... no vlastně *cokoliv*. Definuje třídu, která nemá žádnou metodu `__init__()`. Třída sice *má* nějakou metodu, ale ta se nikdy nevolá. Celý skript obsahuje blok `__main__`, ale nenajdeme v něm odkaz ani na třídu, ani na její metodu. Ale on opravdu něco dělá. Za to ručím.

```
import roman1
import unittest

class KnownValues(unittest.TestCase):                                    [1]
    known_values = ( (1, 'I'),
                     (2, 'II'),
                     (3, 'III'),
                     (4, 'IV'),
                     (5, 'V'),
                     (6, 'VI'),
                     (7, 'VII'),
                     (8, 'VIII'),
                     (9, 'IX'),
                     (10, 'X'),
                     (50, 'L'),
                     (100, 'C'),
                     (500, 'D'),
                     (1000, 'M'),
                     (31, 'XXXI'),
                     (148, 'CXLVIII'),
                     (294, 'CCXCIV'),
                     (312, 'CCCXII'),
                     (421, 'CDXXI'),
                     (528, 'DXXVIII'),
                     (621, 'DCXXI'),
                     (782, 'DCCLXXXII'),
                     (870, 'DCCCLXX'),
                     (941, 'CMXLI'),
                     (1043, 'MXLIII'),
                     (1110, 'MCX'),
                     (1226, 'MCCXXVI'),
                     (1301, 'MCCCI'),
                     (1485, 'MCDLXXXV'),
                     (1509, 'MDIX'),
                     (1607, 'MDCVII'),
                     (1754, 'MDCCLIV'),
                     (1832, 'MDCCCXXXII'),
                     (1993, 'MCMXCIII'),
                     (2074, 'MMLXXIV'),
                     (2152, 'MMCLII'),
                     (2212, 'MMCCXII'),
                     (2343, 'MMCCCXLIII'),
                     (2499, 'MMCDXCIX'),
```

```
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))                              [2]

    def test_to_roman_known_values(self):                             [3]
        '''to_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman1.to_roman(integer)                         [4]
            self.assertEqual(numeral, result)                         [5]

if __name__ == '__main__':
    unittest.main()
```

[1] Když chceme napsat nějaký testovací případ (test case), musíme nejdříve vytvořit třídu odvozenou od třídy `TestCase` z modulu `unittest`. Uvedená třída nám poskytuje řadu užitečných metod, které můžeme v našem testovacím případě využít pro testování specifických podmínek.

[2] Tohle je seznam dvojic s celým číslem a s římským číslem, které jsem ověřil ručně. Obsahuje deset nejmenších čísel, největší číslo, každé číslo, které se vyjadřuje jednoznakovým římským číslem, a náhodnou sadu dalších platných čísel. Nemusíme testovat každý možný vstup, ale měli bychom se pokusit otestovat všechny zřejmé hraniční případy.

[3] Pro každý jednotlivý test je vytvořena jeho vlastní metoda. Metoda testu nemá žádné parametry, nevrací žádnou hodnotu a její jméno musí začínat čtyřmi písmeny `test`. Pokud testovací metoda skončí normálně, bez vyvolání výjimky, pokládáme test za úspěšný. Pokud metoda vyvolá výjimku, považujeme to za selhání testu.

[4] Tady voláme skutečnou funkci `to_roman()`. (Tu funkci jsme zatím nenapsali, ale jakmile ji jednou napíšeme, tento řádek ji zavolá.) Všimněte si, že jsme v tomto okamžiku pro funkci `to_roman()` definovali aplikační programové rozhraní (`API`). Musí přebírat celé číslo (převáděné číslo) a vrací řetězec (reprezentaci římského čísla). Pokud by rozhraní funkce bylo jiné, test by selhal. Všimněte si také, že při volání `to_roman()` žádnou výjimku neodchytáváme. Je to

záměrné. Funkce `to_roman()` by při volání s platným vstupem žádnou výjimku vyvolat neměla a uvedené vstupní hodnoty jsou všechny platné. Pokud `to_roman()` vyvolá výjimku, bude se to považovat za selhání tohoto testu.

[5] Dejme tomu, že funkce `to_roman()` byla korektně definována, korektně volána, úspěšně skončila a vrátila výsledek. Pak nám jako poslední krok zbývá zkontrolovat, zda vrátila *správnou* hodnotu. Jde o obecně používaný dotaz. Ke kontrole, zda se dvě hodnoty shodují, poskytuje třída `TestCase` metodu `assertEqual`. Pokud výsledek (result) vrácený funkcí `to_roman()` neodpovídá očekávané známé hodnotě (numeral), vyvolá `assertEqual` výjimku a test selže. Pokud se ty dvě hodnoty shodují, neudělá `assertEqual` nic. Pokud všechny hodnoty vrácené funkcí `to_roman()` odpovídají očekávaným hodnotám, assertEqual nikdy výjimku nevyvolá, takže metoda `test_to_roman_known_values` nakonec normálně skončí. To znamená, že funkce `to_roman()` testem prošla.

Napište test, který selže, a pak programujte, dokud neprojde.

Jakmile máme vytvořen testovací případ, začneme psát funkci `to_roman()`. Nejdříve ji nahradíme prázdnou funkcí a ověříme si, že test selhává. Pokud by test prošel, aniž jsme napsali nějaký kód, pak by testy náš kód vůbec netestovaly! Unit testing je jako tanec: testy vedou, kód následuje. Napište test, který selže, a pak programujte, dokud neprojde.

```
# roman1.py

def to_roman(n):
    '''convert integer to Roman numeral'''
    pass                                          [1]
```

[1] V této fázi bychom rádi definovali rozhraní funkce `to_roman()`, ale nechceme zatím psát žádný kód. (Náš test musí nejdříve selhat.) Prázdné funkčnosti dosáhneme použitím pythonovského vyhrazeného slova `pass`, které dělá doslova nic.

Spuštění testu zajistíme provedením `romantest1.py` z příkazového řádku. Pokud jej zavoláme s volbou `-v`, dosáhneme podrobnějšího výstupu, takže přesně uvidíme, co se při běhu každého testovacího případu děje. S trochou štěstí by váš výstup měl vypadat nějak takto:

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)                          [1]
to_roman should give known result with known input ... FAIL                [2]

======================================================================
FAIL: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest1.py", line 73, in test_to_roman_known_values
    self.assertEqual(numeral, result)
```

```
AssertionError: 'I' != None                                                    [3]

----------------------------------------------------------------------
Ran 1 test in 0.016s                                                           [4]

FAILED (failures=1)                                                            [5]
```

[1] Když skript spustíme, spustí se funkce `unittest.main()`, která zajistí provedení každého testovacího případu. Každý testovací případ je metodou třídy z `romantest1.py`. U testovacích tříd se nevyžaduje nějaká zvláštní organizace. Každá z nich může obsahovat jedinou metodu, nebo můžeme mít jednu třídu, která obsahuje množství testovacích metod. Jediným požadavkem je to, že každá testovací třída musí dědit z třídy `unittest.TestCase`.

[2] Pro každý testovací případ modul `unittest` vytiskne `docstring` metody a to, zda test prošel (pass) nebo selhal (fail). Tento test podle očekávání selhal.

[3] Pro každý testovací případ, který selhal, zobrazí `unittest` trasovací informaci, která přesně ukazuje, co se stalo. V tomto případě vyvolala metoda `assertEqual()` výjimku `AssertionError`, protože se očekávalo, že funkce `to_roman(1)` vrátí `'I'`, ale nevrátila. (Protože jsme v ní explicitně neuvedli příkaz `return`, vrátila funkce hodnotu `None`, což je pythonovský ekvivalent hodnoty null.)

[4] Po detailních výpisech každého testu zobrazí unittest souhrnně, kolik testů se provádělo a jak dlouho to trvalo.

[5] Testovací běh celkově selhal, protože minimálně jeden test neprošel. Pokud testovací případ neprojde, rozlišuje `unittest` mezi selháním (failure) a chybou (error). Selhání (failure) je důsledkem volání metody `assertXYZ`, jako je například `assertEqual` nebo `assertRaises`, která selhala, protože neplatí předepsaná podmínka nebo nebyla vyvolána očekávaná výjimka. Za chybu (error) se považuje jakýkoliv jiný druh výjimky, která vznikla uvnitř testované kódu nebo v kódu testovacího případu.

A teď už můžeme konečně napsat funkci `to_roman()`.

```
roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
                     ('L',  50),
                     ('XL', 40),
                     ('X',  10),
                     ('IX', 9),
                     ('V',  5),
                     ('IV', 4),
                     ('I',  1) )                                               [1]
```

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:                     [2]
            result += numeral
            n -= integer
    return result
```

[1] `roman_numeral_map` je n-tice n-tic, které definují tři věci: znakovou reprezentaci nejzákladnějších římských čísel, pořadí římských čísel (sestupně od `M` až po `I`), hodnotu každého římského čísla. Každá vnitřní n-tice je dvojicí (`římské číslo, hodnota`). Nejsou zde jen jednoznaková římská čísla. Jsou zde definována i dvojznaková čísla jako `CM` („o jedno sto méně než jeden tisíc“). Tím se kód funkce `to_roman()` zjednoduší.

[2] Zde je to místo, kde se bohatá datová struktura `roman_numeral_map` uplatní, protože díky ní k realizaci odečítacího pravidla nepotřebujeme žádnou speciální logiku. Při převodu na římské číslo jednoduše procházíme strukturou `roman_numeral_map` a hledáme největší celočíselnou hodnotu, která je menší nebo rovna vstupu. Jakmile ji nalezneme, přidáme její reprezentaci římským číslem na konec výstupu, odečteme odpovídající celočíselnou hodnotu od vstupu, namydlíme, opláchneme, zopakujeme.

Pokud vám pořád není jasné, jak funkce `to_roman()` pracuje, přidejte na konec cyklu `while` volání funkce `print()`:

```
while n >= integer:
    result += numeral
    n -= integer
    print('subtracting {0} from input, adding {1} to output'.format(integer, numeral))
```

S ladicími příkazy `print()` vypadá výstup takto:

```
>>> import roman1
>>> roman1.to_roman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

Takže se zdá, že funkce `to_roman()` pracuje přinejmenším v tomto ručně zkoušeném případě. Ale projde testovacím případem, který jsme napsali?

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok                [1]

----------------------------------------------------------------------
Ran 1 test in 0.016s

OK
```

[1] Hurá! Funkce `to_roman()` prošla testovacím případem nazvaným „známé hodnoty“. Není sice všeobsažný, ale prověřil schopnosti funkce celou škálou vstupů, včetně vstupů, které produkují každé jednoznakové římské číslo, největší možný vstup (`3999`), a vstupu, který produkuje nejdelší možné římské číslo (`3888`). V tomto okamžiku už můžeme docela důvěřovat tomu, že funkce pracuje pro libovolnou správnou vstupní hodnotu, kterou bychom mohli zadat.

„Správný“ vstup? Hmm. A co takhle chybný vstup?

9.3. „Zastav a začni hořet“

Pythonovská signalizace typu „zastav a začni hořet“ spočívá ve vyvolání výjimky.

Ono ale nestačí, když funkce uspějí při zadání správného vstupu. Musíme otestovat také to, že při chybném vstupu dojde k jejich selhání. Ale nemůže jít o jakýkoliv způsob selhání. Funkce musí selhat očekávaným způsobem.

```
>>> import roman1
>>> roman1.to_roman(4000)
'MMMM'
>>> roman1.to_roman(5000)
'MMMMM'
>>> roman1.to_roman(9000)        [1]
'MMMMMMMMM'
```

[1] Tohle určitě není to, co jsme chtěli. Vždyť se dokonce nejedná ani o platné římské číslo! Každé z těchto čísel leží ve skutečnosti mimo rozsah přijatelných vstupů, ale funkce pro ně stejně vrací falešné, vykonstruované hodnoty. Pokud potichu vracíme špatné hodnoty, je to *velmi špatné*. Pokud má program selhat, pak je mnohem lepší, když selže rychle a nahlas. Jak se říká, „zastav a začni hořet“. (Jde o překlad anglické fráze „Halt And Catch Fire“, která se při práci na úrovních blízkých hardwaru vztahuje k mechanismu velmi dobře pozorovatelného projevu nějaké neočekávané chyby. Vysvětlení původu této hlášky se různí, od skutečně kouřících přežhavených drátků feritové paměti při dynamické realizaci instrukce HALT, až po speciální

nedokumentované strojové instrukce, které uvedou procesor do testovacího režimu.) Pythonovská signalizace typu „zastav a začni hořet“ spočívá ve vyvolání výjimky.

Měli byste si položit otázku: „Jak bychom to mohli vyjádřit formou testovatelného požadavku?“ Co kdybychom začali nějak takto:

> Pokud funkci `to_roman()` zadáme celé číslo větší než `3999`, měla by vyvolat výjimku `OutOfRangeError`.

Jak by vypadal příslušný test?

```
class ToRomanBadInput(unittest.TestCase):                                    [1]
    def test_too_large(self):                                                [2]
        '''to_roman should fail with large input'''
        self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)     [3]
```

[1] Podobně jako v předchozím testovacím případě vytvoříme třídu, která dědí z `unittest.TestCase`. Jedna třída sice může obsahovat více než jeden test (jak si ukážeme v této kapitole později), ale já jsem se rozhodl, že vytvořím novou třídu, protože tento test dělá něco jiného než ten minulý. Všechny testy správných vstupů budeme udržovat v jedné třídě a o všechny testy chybných vstupů se bude starat druhá třída.

[2] Vlastní test, stejně jako v předchozím testovacím případě, má podobu metody třídy. Její jméno začíná písmeny `test`.

[3] Třída `unittest.TestCase` poskytuje metodu `assertRaises`, která přebírá následující argumenty: očekávanou výjimku, testovanou funkci a argumenty, které jí chceme předat. (Pokud testovaná funkce očekává více než jeden argument, předejte je metodě `assertRaises` všechny v daném pořadí. Ona už se postará o jejich předání testované funkci.)

Věnujte zvláštní pozornost tomu poslednímu řádku kódu. Místo toho, abychom volali `to_roman()`, přímo a ručně zkontrolovali, že vyvolává konkrétní výjimku (obalením do bloku `try...except`), metoda `assertRaises` to vše udělá za nás. Musíme jí jen říct, jakou výjimku očekáváme (`roman2.OutOfRangeError`), předat funkci (`to_roman()`) a její argumenty (`4000`). Metoda `assertRaises` se postará o zavolání `to_roman()` a o kontrolu toho, že vyvolala výjimku `roman2.OutOfRangeError`.

Poznamenejme také, že funkci `to_roman()` předáváme jako argument. Nevoláme ji a ani nepředáváme její jméno jako řetězec. Zmínil jsem se už dříve o tom, jak je šikovné, že v Pythonu je vše objektem?

Takže co se stane, když spustíme sadu testů doplněnou o tento nový test?

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
```

```
to_roman should fail with large input ... ERROR                          [1]

======================================================================
ERROR: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AttributeError: 'module' object has no attribute 'OutOfRangeError'      [2]

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

[1] Asi jste očekávali, že dojde k selhání (protože zatím jsme nenapsali žádný kód, aby to prošlo), ale... ono to ve skutečnosti „neselhalo“ (fail). Místo toho došlo k „chybě“ (error). Je to sice jemný, ale důležitý rozdíl. Jednotkový test má ve skutečnosti *tři návratové hodnoty*: prošel (pass), selhal (fail) a chyba (error). „Pass“ (prošel) samozřejmě znamená, že test prošel. Kód dělá to, co jsme očekávali. „Fail“ (selhal) vyjadřuje to, co udělal minulý test (než jsme napsali kód, díky kterému prošel). Kód se provedl, ale výsledek neodpovídá tomu, co jsme očekávali. „Error“ (chyba) se objeví, když kód ani správně nedoběhl.

[2] A proč vlastně kód správně neproběhl? Vše se dozvíme z trasovacího hlášení. Testovaný modul vůbec nedefinuje výjimku zvanou `OutOfRangeError` (tj. hodnota mimo platný rozsah). Připomeňme si, že uvedenou výjimku jsme předali metodě `assertRaises()`, protože právě tohle má být výjimka, kterou má funkce vyvolat, když zadáme vstup mimo platný rozsah. Ale tato výjimka vůbec neexistuje, takže volání metody `assertRaises()` selhalo. Metoda neměla vůbec šanci otestovat funkci `to_roman()`. Tak daleko se vůbec nedostala.

K vyřešení zmíněného problému musíme v `roman2.py` doplnit definici výjimky `OutOfRangeError`.

```
class OutOfRangeError(ValueError):      [1]
    pass                                [2]
```

[1] Výjimky mají podobu tříd. Chyba „mimo platný rozsah“ je druhem chyby hodnoty. Hodnota argumentu se nachází mimo přijatelné meze. Z tohoto důvodu výjimka dědí ze zabudované výjimky `ValueError`. Není to nezbytně nutné (mohli bychom prostě dědit od bázové třídy `Exception`, tj. obecná výjimka), ale zdá se to být správné.

[2] Výjimky samy o sobě ve skutečnosti nic nedělají, ale potřebujete nejméně jeden řádek kódu, abychom definovali třídu. Volání `pass` sice nic nedělá, ale je to řádek pythonovského kódu, který zajistí, že třída vznikne.

Teď spustíme sadu testů znovu.

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... FAIL                          [1]

======================================================================
FAIL: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AssertionError: OutOfRangeError not raised by to_roman                  [2]

----------------------------------------------------------------------
Ran 2 tests in 0.016s

FAILED (failures=1)
```

[1] Nový test sice stále neprošel, ale už také nevrací chybu. Místo toho došlo k selhání testu. To je pokrok! To znamená, že volání metody `assertRaises()` tentokrát prošlo a rámec pro testování jednotek (unit test framework) skutečně testoval funkci `to_roman()`.

[2] Funkce `to_roman()` zatím, samozřejmě, nevyvolává právě definovanou výjimku `OutOfRangeError`, protože jsme jí ještě neřekli, že to má dělat. To je ale výborná zpráva! Znamená to, že máme platný testovací případ — selhává (fails) před napsáním kódu, který zajistí, že projde.

Teď napíšeme kód, který zajistí, aby funkce testem prošla.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if n > 3999:
        raise OutOfRangeError('number out of range (must be less than 4000)')    [1]

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

[1] Přímočaré řešení: Pokud je daný vstup (`n`) větší než `3999`, vyvolej výjimku `OutOfRangeError`. Tento jednotkový test nekontroluje, zda výjimku doprovází lidsky čitelný řetězec. Mohli bychom napsat další test, který by to kontroloval (ale pozor na problémy s internacionalizací; řetězce se mohou lišit v závislosti na jazyku uživatele a v závislosti na prostředí).

Vede úprava k tomu, že test projde? Pojďme to zjistit.

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok                              [1]

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

[1] Hurá! Oba testy prošly. Protože jsme pracovali po krocích (přebíhali jsme mezi testováním a psaním kódu), můžeme si být jisti, že ty dva řádky kódu, které jsme právě napsali, byly příčinou toho, že se výsledek testu změnil z „fail“ (selhal) na „pass“ (prošel). Tento druh (sebe)důvěry sice nebyl zadarmo, ale během života našeho kódu se ještě vyplatí.

9.4. Více zastávek, více ohně

Spolu s testováním čísel, která jsou příliš velká, bychom měli testovat i čísla, která jsou příliš malá. Přesně jak jsme poznamenali v našich požadavcích na funkčnost, římská čísla nemohou vyjádřit nulu nebo záporná čísla.

```
>>> import roman2
>>> roman2.to_roman(0)
''
>>> roman2.to_roman(-1)
''
```

Hmm, *tohle* není dobré. Přidejme testy pro každou z těchto podmínek.

```
class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 4000)         [1]

    def test_zero(self):
        '''to_roman should fail with 0 input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)            [2]

    def test_negative(self):
        '''to_roman should fail with negative input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)           [3]
```

[1] Metoda `test_too_large()` se od minulého kroku nezměnila. Ponechal jsem ji zde, abych ukázal, kam nový kód zapadá.

[2] Máme tu nový test, metodu `test_zero()`. Je to stejné jako u metody `test_too_large()`. Metodě `assertRaises()` z třídy `unittest.TestCase` říkáme, aby zavolala naši funkci `to_roman()` s parametrem `0` a zkontrolovala, zda vyvolá příslušnou výjimku `OutOfRangeError`.

[3] Metoda `test_negative()` je téměř shodná až na to, že funkci `to_roman()` předává hodnotu `-1`. Pokud kterýkoliv z těchto nových testů *nevyvolá* výjimku `OutOfRangeError` (protože funkce buď vrátí nějakou skutečnou hodnotu nebo vyvolá nějakou jinou výjimku), bude se to považovat za selhání testu.

Teď zkontrolujme, že testy selhávají:

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... FAIL

======================================================================
FAIL: to_roman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 86, in test_negative
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman

======================================================================
FAIL: to_roman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 82, in test_zero
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=2)
```

Výborně. Oba testy podle očekávání selhaly. Teď se přepněme na psaní kódu a uvidíme, co můžeme dělat, aby testy prošly.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 4000):                                              [1]
        raise OutOfRangeError('number out of range (must be 1..3999)')  [2]

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

[1] Tohle je pěkná pythonovská zkratka — více porovnání najednou. Je to ekvivalentní zápisu `if not ((0 < n) and (n < 4000))`, ale je to mnohem čitelnější. Tento řádek kódu by měl zachytit vstupy, které jsou příliš velké, záporné nebo nulové.

[2] Pokud podmínky změníte, nezapomeňte odpovídajícím způsobem upravit i lidsky čitelný řetězec. Rámci `unittest` je to jedno. Pokud by ale váš kód vyvolával nesprávně popsané výjimky, ztížilo by se tím ruční ladění.

Mohl bych vám ukázat celou sérii nesouvisejících příkladů, které ukazují, že zkratka umožňující několik porovnání najednou funguje. Místo toho ale spustím testy jednotek a dokážu vám to.

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.016s

OK
```

9.5. A ještě jedna věc...

Mezi požadavky na převod na římská čísla byl ještě jeden, který se týkal neceločíselného vstupu.

```
>>> import roman3
>>> roman3.to_roman(0.5)        [1]
''
>>> roman3.to_roman(1.0)        [2]
'I'
```

[1] A jéje, to je špatné.

[2] Jejda, tohle je ještě horší. V obou uvedených případech by měla být vyvolána výjimka. Místo toho produkují falešné výstupy.

Testování na neceločíselný vstup není obtížné. Nejdříve si definujeme výjimku `NotIntegerError`.

```
# roman4.py
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
```

Dále napíšeme testovací případ, který kontroluje výskyt výjimky `NotIntegerError`.

```
class ToRomanBadInput(unittest.TestCase):
    .
    .
    .
    def test_non_integer(self):
        '''to_roman should fail with non-integer input'''
        self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
```

Teď zkontrolujme, zda test správně selhává.

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok
```

```
======================================================================
FAIL: to_roman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest4.py", line 90, in test_non_integer
    self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
AssertionError: NotIntegerError not raised by to_roman

----------------------------------------------------------------------
Ran 5 tests in 0.000s

FAILED (failures=1)
```

Napíšeme kód, který má zajistit, aby test prošel.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 4000):
        raise OutOfRangeError('number out of range (must be 1..3999)')
    if not isinstance(n, int):                                          [1]
        raise NotIntegerError('non-integers can not be converted')      [2]

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

[1] Zabudovaná funkce `isinstance()` testuje, zda je daná proměnná určitého typu (nebo, z technického hlediska, nějakého z něj odvozeného typu).

[2] Pokud argument `n` není typu `int`, vyvolej naši zbrusu novou výjimku `NotIntegerError`.

Nakonec zkontrolujeme, že tento kód zajistil průchod testem.

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
```

```
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Funkce `to_roman()` prošla všemi testy a žádné další testy mě nenapadají. Takže nastal čas, abychom se přesunuli k `from_roman()`.

9.6. Symetrie, která potěší

Převod řetězce vyjadřujícího římské číslo na číselnou hodnotu vypadá složitěji než převod čísla na římské číslo. Určitě budeme muset zajistit ověření platnosti. Zkontrolovat, zda je číslo rovno nule, je snadné. O něco obtížněji se kontroluje, zda je řetězec platným římským číslem. Jenže my už jsme zkonstruovali regulární výraz, který zkontroluje, zda jde o římské číslo. Takže tuhle část už máme hotovou.

Zbývá nám problém samotné konverze řetězce. Jak za chvíli uvidíme, díky existenci datové struktury, kterou jsme definovali pro převod určitých římských čísel na celočíselné hodnoty, bude jádro funkce `from_roman()` stejně přímočaré jako u funkce `to_roman()`.

Ale nejdříve testy. Pro ověření správnosti konkrétních hodnot budeme potřebovat test „známých hodnot“. Naše testovací sada již tabulku známých hodnot obsahuje, takže ji využijme.

```
    def test_from_roman_known_values(self):
        '''from_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

Najdeme zde potěšitelnou symetrii. Funkce `to_roman()` a `from_roman()` jsou vzájemně inverzní. První z nich převádí čísla na zvláštně formátované řetězce a druhá převádí zvláštně formátované řetězce na celá čísla. Teoreticky bychom měli být schopni dospět ke zvolenému číslu oklikou tak, že je nejdříve předáme funkci `to_roman()`. Získaný řetězec předáme funkci `from_roman()` a výsledné číslo by se mělo shodovat s počátečním.

```
n = from_roman(to_roman(n)) pro všechny hodnoty n
```

V tomto případě „všechny hodnoty“ znamená jakoukoliv hodnotu `1..3999`, protože toto je platný rozsah vstupů pro funkci `to_roman()`. Tuto symetrii můžeme vyjádřit testovacím případem, který prochází všechny hodnoty `1..3999`, volá `to_roman()`, volá `from_roman()` a kontroluje, zda se výstup shoduje s původním vstupem.

```
class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 4000):
            numeral = roman5.to_roman(integer)
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

Tyto nové testy zatím ani neselžou (fail). Zatím jsme vůbec nedefinovali funkci `from_roman()`, takže způsobí chyby (errors).

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
E.E....
======================================================================
ERROR: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 78, in test_from_roman_known_values
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

======================================================================
ERROR: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 103, in test_roundtrip
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

----------------------------------------------------------------------
Ran 7 tests in 0.019s

FAILED (errors=2)
```

Problém vyřešíme rychlým vytvořením náhradní funkce.

```
# roman5.py
def from_roman(s):
    '''convert Roman numeral to integer'''
```

(Hej, všimli jste si toho? Definoval jsem funkci, která neobsahuje nic než `docstring` (dokumentační řetězec). Tohle je v Pythonu legální. Někteří programátoři vás ve skutečnosti zapřísahají: „Nepište náhrady. Dokumentujte!“)

Teď už testovací případy opravdu selžou (fail).

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
F.F....
======================================================================
FAIL: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 79, in test_from_roman_known_values
    self.assertEqual(integer, result)
AssertionError: 1 != None

======================================================================
FAIL: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 104, in test_roundtrip
    self.assertEqual(integer, result)
AssertionError: 1 != None

----------------------------------------------------------------------
Ran 7 tests in 0.002s

FAILED (failures=2)
```

Nastal čas napsat funkci `from_roman()`.

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:            [1]
            result += integer
            index += len(numeral)
    return result
```

[1] Základní vzorec je zde stejný jako u funkce `to_roman()`. Procházíme datovou strukturou s římskými čísly (n-tice n-tic), ale místo hledání nejvyšších možných číselných hodnot se snažíme hledat řetězec znaků s „nejvyšším“ možným římským číslem.

Pokud vám pořád není jasné, jak funkce `from_roman()` pracuje, přidejte na konec cyklu `while` volání funkce `print`:

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print('found', numeral, 'of length', len(numeral), ', adding', integer)
>>> import roman5
>>> roman5.from_roman('MCMLXXII')
found M of length 1, adding 1000
found CM of length 2, adding 900
found L of length 1, adding 50
found X of length 1, adding 10
found X of length 1, adding 10
found I of length 1, adding 1
found I of length 1, adding 1
1972
```

Nastal opět čas ke spuštění testů.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.060s

OK
```

Máme tady dvě vzrušující zprávy. Ta první je, že funkce `from_roman()` funguje pro správné vstupy — přinejmenším pro všechny známé hodnoty. Ta druhá zpráva je, že test „kruhovým voláním“ (round trip test) také prošel. Když to zkombinujeme dohromady, můžeme si být docela jistí tím, že jak funkce `to_roman()`, tak funkce `from_roman()` pracují správně pro všechny možné správné hodnoty. (Není to ale zaručeno. Teoreticky je možné, že `to_roman()` obsahuje chybu, která pro určité hodnoty vstupů produkuje špatná římská čísla, a *současně* funkce `from_roman()` obsahuje obrácenou chybu, která pro-

dukuje stejná, ale špatná čísla přesně pro tu množinu římských čísel, která funkce `to_roman()` vygenerovala nesprávně. V závislosti na vaší aplikaci a na požadavcích by vám to mohlo dělat starosti. Pokud tomu tak je, napište obsažnější testovací případy, které vaše starosti rozptýlí.)

9.7. Více špatných vstupů

Teď, když už funkce `from_roman()` pracuje správně pro korektní vstup, nastal čas k umístění posledního kousku skládanky — zajištění správné funkce pro špatné vstupy. To znamená, že musíme najít způsob, jak se podívat na řetězec a určit, zda je platným římským číslem. To už je ze své podstaty obtížnější než ověřování správnosti číselného vstupu ve funkci `to_roman()`. Ale máme k dispozici mocný nástroj — regulární výrazy. (Pokud regulární výrazy neznáte, pak je vhodná doba na to, abyste si přečetli kapitolu o regulárních výrazech.)

Kap.

Kap. V podkapitole Případová studie: Římská čísla jsme viděli, že existuje několik jednoduchých pravidel pro konstrukci římského čísla, která jsou založena na využití písmen `M`, `D`, `C`, `L`, `X`, `V` a `I`. Pojďme si tato pravidla zopakovat:

V některých případech se znaky sčítají. `I` je `1`, `II` je rovno `2` a `III` znamená `3`. `VI` se rovná `6` (doslova „`5` a `1`“), `VII` je `7` a `VIII` je `8`.

- Desítkové znaky (`I`, `X`, `C` a `M`) se mohou opakovat nanejvýš třikrát. Hodnotu `4` musíme vyjádřit odečtením od dalšího vyššího pětkového znaku. Hodnotu `4` nemůžeme zapsat jako `IIII`. Místo toho ji musíme zapsat jako `IV` („o `1` méně než `5`“). `40` se zapisuje jako `XL` („o `10` méně než `50`“), `41` jako `XLI`, `42` jako `XLII`, `43` jako `XLIII` a následuje `44` jako `XLIV` („o `10` méně než `50` a k tomu o `1` méně než `5`“).
- Někdy znaky vyjadřují… opak sčítání. Když některé znaky umístíme před jiné, provádíme odčítání od konečné hodnoty. Například hodnotu `9` musíme vyjádřit odečtením od dalšího vyššího desítkového znaku: `8` zapíšeme jako `VIII`, ale `9` zapíšeme `IX` („o `1` méně než `10`“) a ne jako `VIIII` (protože znak `I` nemůžeme opakovat čtyřikrát). `90` je `XC`, `900` je `CM`.
- Pětkové znaky se nesmí opakovat. `10` se vždy zapisuje jako `X` a nikdy jako `VV`. 100 je vždy `C`, nikdy `LL`.
- Římská čísla se čtou zleva doprava, takže na pořadí znaků velmi záleží. `DC` znamená `600`, ale `CD` je úplně jiné číslo (`400`, „o `100` méně než `500`“). `CI` je `101`, ale `IC` není dokonce vůbec platné římské číslo (protože `1` nemůžeme přímo odčítat od `100`; musíme to napsat jako `XCIX`, „o `10` méně než `100` a k tomu o `1` méně než `10`“).

Takže jeden z užitečných testů bude ověřovat, že by funkce `from_roman()` měla selhat (fail) v případě, kdy jí předáme řetězec s příliš mnoha opakujícími se římskými číslicemi. Co znamená „příliš mnoho“, závisí na konkrétní číslici.

```
class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Další užitečný test bude založen na kontrole, že se neopakují některé vzory. Například `IX` je `9`, ale `IXIX` je vždy neplatné.

```
    def test_repeated_pairs(self):
        '''from_roman should fail with repeated pairs of numerals'''
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Třetí test by mohl kontrolovat, zda se číslice objevují ve správném pořadí, od nejvyšších k nejnižším hodnotám. Například `CL` je `150`, ale `LC` je vždy neplatné, protože číslice pro `50` se nesmí nikdy vyskytovat před číslicí pro `100`. Tento test zahrnuje náhodně zvolenou množinu nesprávných předchůdců: `I` před `M`, `V` před `X` a tak dále.

```
    def test_malformed_antecedents(self):
        '''from_roman should fail with malformed antecedents'''
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Každý z těchto testů spoléhá na to, že funkce `from_roman()` vyvolává novou výjimku `InvalidRomanNumeralError`, kterou jsme ještě nedefinovali.

```
# roman6.py
class InvalidRomanNumeralError(ValueError): pass
```

Všechny tři testy by měly selhat (fail), protože funkce `from_roman()` momentálně neprovádí žádnou kontrolu platnosti. (Pokud by neselhaly teď, tak co by vlastně testovaly?)

```
you@localhost:~/diveintopython3/examples$ python3 romantest6.py
FFF.......
======================================================================
FAIL: test_malformed_antecedents (__main__.FromRomanBadInput)
from_roman should fail with malformed antecedents
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "romantest6.py", line 113, in test_malformed_antecedents
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_repeated_pairs (__main__.FromRomanBadInput)
from_roman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 107, in test_repeated_pairs
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_too_many_repeated_numerals (__main__.FromRomanBadInput)
from_roman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 102, in test_too_many_repeated_numerals
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 10 tests in 0.058s

FAILED (failures=3)
```

Fajn. Teď už do funkce `from_roman()` potřebujeme přidat jen regulární výraz, který testuje platnost římských čísel.

```
roman_numeral_pattern = re.compile('''
    ^                   # začátek řetězce
    M{0,3}              # tisíce - 0 až 3 M
    (CM|CD|D?C{0,3})    # stovky - 900 (CM), 400 (CD), 0-300 (0 až 3 C),
                        #        nebo 500-800 (D následované 0 až 3 C)
    (XC|XL|L?X{0,3})    # desítky - 90 (XC), 40 (XL), 0-30 (0 až 3 X),
                        #        nebo 50-80 (L následované 0 až 3 X)
    (IX|IV|V?I{0,3})    # jednotky - 9 (IX), 4 (IV), 0-3 (0 až 3 I),
                        #        nebo 5-8 (V následované 0 až 3 I)
    $                   # konec řetězce
    ''', re.VERBOSE)
```

```
def from_roman(s):
    '''convert Roman numeral to integer'''
    if not roman_numeral_pattern.search(s):
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))

    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index : index + len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

A znovu spustíme testy…

```
you@localhost:~/diveintopython3/examples$ python3 romantest7.py
..........
----------------------------------------------------------------------
Ran 10 tests in 0.066s

OK
```

A cenu za zklamání roku dostává… slovo „OK“, které modul `unittest` zobrazí poté, co všechny testy prošly.

> **“ After one has played a vast quantity of notes and more notes, it is simplicity that emerges as the crowning reward of art.”**
> (Poté, co jste zahráli ohromné množství not a ještě více not, se jako vrcholná odměna umění objeví jednoduchost.)
>
> — Frédéric Chopin

10. Refaktorizace

10. Refaktorizace — 219
10.1. Ponořme se — **221**
10.2. Zvládání měnících se požadavků — **223**
10.3. Refaktorizace — **228**
10.4. Shrnutí — **232**

10.1. Ponořme se

K chybám dochází, ať se vám to líbí nebo ne. Chyby se objeví navzdory vašemu nejlepšímu úsilí o vytvoření všezahrnujících testů jednotek (unit test). Co vlastně myslím slovem „chyba“? Chybou rozumím testovací případ (test case), který jste ještě nenapsali.

```
>>> import roman7
>>> roman7.from_roman('')                [1]
0
```

[1] Tohle je chyba. Prázdný řetězec by měl vyvolat výjimku `InvalidRomanNumeralError` stejně jako jiné posloupnosti znaků, které nevyjadřují platné římské číslo.

Jakmile chybu umíte navodit, měli byste napsat testovací případ (test case) ještě dříve, než ji opravíte. Tím chybu popíšete.

```
class FromRomanBadInput(unittest.TestCase):
    .
    .
    .
    def testBlank(self):
        '''from_roman should fail with blank string'''
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, '')      [1]
```

[1] Je to docela jednoduché. Voláme funkci `from_roman()` s prázdným řetězcem a ujišťujeme se, že vyvolává výjimku `InvalidRomanNumeralError`. Nalezení chyby je obtížnou částí úkolu. Pokud už o ní víme, představuje její otestování snadnou část úkolu.

Protože náš kód obsahuje chybu a protože už máme k dispozici testovací případ, který ji popisuje, dojde k jeho selhání:

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... FAIL
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok
```

```
======================================================================
FAIL: from_roman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest8.py", line 117, in test_blank
    self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, '')
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 11 tests in 0.171s

FAILED (failures=1)
```

Teď už chybu můžeme opravit.

```
def from_roman(s):
    '''convert Roman numeral to integer'''
    if not s:                                                              [1]
        raise InvalidRomanNumeralError('Input can not be blank')
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError('Invalid Roman numeral: {}'.format(s))    [2]

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

[1] Musíme přidat jen dva řádky kódu: explicitní kontrolu na prázdný řetězec a příkaz `raise`.

[2] Myslím, že o tomhle jsem se v této knize zatím ještě nezmínil. Nechť to slouží jako závěrečná lekce z formátování řetězců. Počínaje verzí Python 3.1 můžete při specifikaci formátu vynechat čísla pozičních indexů. To znamená, že místo specifikátoru `{0}`, kterým se odkazujeme na první parametr metody `format()`, můžeme jednoduše použít `{}` a Python doplní správný poziční index za nás. Funguje to pro libovolný počet argumentů. První `{}` se chápe jako `{0}`, druhý výskyt `{}` znamená `{1}` a tak dále.

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... ok                            [1]
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
```

```
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 11 tests in 0.156s

OK                                                                    [2]
```

[1] Testovací případ pro prázdný řetězec prošel, takže chyba je opravena.

[ě] Všechny ostatní testovací případy prošly také. To znamená, že jsme opravou chyby nic jiného nepokazili. Přestaňte psát kód.

Tento přístup k programování opravu chyb nijak neusnadňuje. Jednoduché chyby (jako je tato) vyžadují jednodušší testovací případy, složité chyby povedou k složitým testovacím případům. V prostředí soustředěném kolem testů se může *zdát*, že oprava chyby trvá déle. Musíme chybu přesně popsat v kódu (tj. musíme napsat testovací případ) a teprve potom ji opravit. Pokud testovací případ hned neprojde, musíme zjistit, zda jsme udělali chybu v opravě, nebo zda je chyba v kódu testovacího případu. Ale z dlouhodobého hlediska se střídavá tvorba testovacího a testovaného kódu vyplatí, protože se tím zvyšuje pravděpodobnost správné opravy chyb napoprvé. S vaším novým testem se také snadno opakovaně spouštějí *všechny* testy. Proto je málo pravděpodobné, že opravou nového kódu pokazíte původní kód. Dnešní test jednotky (unit test) je zítřejším regresním testem.

10.2. Zvládání měnících se požadavků

Navzdory vašemu nejlepšímu úsilí o připíchnutí zákazníka k zemi, poté co z něj při bolestivé proceduře zahrnující hrůzné odpornosti (jako jsou nůžky a horký vosk) vytáhnete přesné požadavky... ty požadavky se změní. Většina zákazníků neví, co chce, dokud to neuvidí. A dokonce když už to vidí, nejsou dost dobří na to, aby vyjádřili, co chtějí, tak přesně, aby to k něčemu bylo. A dokonce i když se vyjádří přesně, v příští verzi toho stejně budou chtít víc. Takže v souvislosti s měnícími se požadavky buďte připraveni na úpravy svých testovacích případů (test case).

Dejme tomu, že bychom například chtěli rozšířit rozsah funkce pro převod římských čísel. V římských číslech se žádný znak nemůže opakovat víc než třikrát. Ale Římané byli ochotni připustit výjimku z tohoto pravidla a reprezentovat hodnotu `4000` uvedením čtyř `M` za sebou. Pokud takovou změnu provedeme, budeme schopni rozšířit rozsah převáděných čísel z `1..3999` na `1..4999`. Ale nejdříve provedeme úpravy testovacích případů.

```
class KnownValues(unittest.TestCase):
    known_values = ( (1, 'I'),
                     .
                     .
                     .
                    (3999, 'MMMCMXCIX'),
                    (4000, 'MMMM'),                                    [1]
                    (4500, 'MMMMD'),
                    (4888, 'MMMMDCCCLXXXVIII'),
                    (4999, 'MMMMCMXCIX') )

class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''
        self.assertRaises(roman8.OutOfRangeError, roman8.to_roman, 5000) [2]

    .
    .
    .

class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):    [3]
            self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, s)

    .
    .
    .

class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 5000):                                 [4]
            numeral = roman8.to_roman(integer)
            result = roman8.from_roman(numeral)
            self.assertEqual(integer, result)
```

[1] Stávající známé hodnoty se nemění (pořád jde o rozumné testovací hodnoty), ale potřebujeme přidat pár dalších v rozsahu od `4000`. Přidali jsme `4000` (nejkratší), `4500` (druhé nejkratší), `4888` (nejdelší) a `4999` (největší).

[2] Změnila se definice „velké vstupní hodnoty“. U tohoto testu se při volání `to_roman()` s hodnotou `4000` očekávala chyba. Teď se ale rozsah `4000-4999` považuje za správné hodnoty, proto musíme hranici zvýšit na `5000`.

[3] Změnila se také definice „příliš mnoho opakujících se znaků“. U tohoto testu se při volání `tfrom_roman()` se vstupem `'MMMM'` očekávala chyba. Teď je `MMMM` považováno za platné římské číslo. Testovací hodnotu musíme zvětšit na `'MMMMM'`.

[4] Test funkčnosti procházel v cyklu každým číslem z intervalu `1` až `3999`. Rozsah se teď rozšířil, takže cyklus `for` musíme upravit, aby se dostal až k `4999`.

Teď máme testovací případy upraveny ve shodě s novými požadavky, ale kód zatím ne. Takže se dá čekat, že některé z testů selžou.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ERROR           [1]
to_roman should give known result with known input ... ERROR             [2]
from_roman(to_roman(n))==n for all n ... ERROR                           [3]
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
ERROR: from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 82, in test_from_roman_known_values
    result = roman9.from_roman(numeral)
  File "C:\home\diveintopython3\examples\roman9.py", line 60, in from_roman
    raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
roman9.InvalidRomanNumeralError: Invalid Roman numeral: MMMM

======================================================================
ERROR: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 76, in test_to_roman_known_values
    result = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)
```

```
======================================================================
ERROR: from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 131, in testSanity
    numeral = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)

----------------------------------------------------------------------
Ran 12 tests in 0.171s

FAILED (errors=3)
```

[1] Test známých hodnot pro `from_roman()` selže v okamžiku, kdy se dostane k hodnotě `'MMMM'`. Funkce `from_roman()` si totiž pořád myslí, že jde o neplatné římské číslo.

[2] Test známých hodnot pro `to_roman()` selže v okamžiku, kdy se narazí na hodnotu `4000`, protože `to_roman()` ji stále považuje za hodnotu mimo rozsah.

[3] Kruhový test selže rovněž u hodnoty `4000`, protože `to_roman()` ji považuje za hodnotu mimo rozsah.

Máme tedy testovací případy, které selhávají v důsledku nových požadavků, a můžeme uvažovat o opravení kódu do odpovídajícího stavu. (Když s psaním testů jednotek (unit test) začínáte, můžete mít divný pocit, že testovaný kód nikdy „nepředbíhá“ testovací případy. Dokud je pozadu, máme pořád nějakou práci před sebou. Jakmile doběhne testovací případy, přestaneme jej upravovat. Jakmile si na to jednou zvyknete, budete se divit, jak jste vůbec dříve mohli programovat bez testů.)

```
roman_numeral_pattern = re.compile('''
    ^                   # začátek řetězce
    M{0,4}              # tisíce - 0 až 4 M                             [1]
    (CM|CD|D?C{0,3})    # stovky - 900 (CM), 400 (CD), 0-300 (0 až 3 C),
                        #        nebo 500-800 (D následované 0 až 3 C)
    (XC|XL|L?X{0,3})    # desítky - 90 (XC), 40 (XL), 0-30 (0 až 3 X),
                        #        nebo 50-80 (L následované 0 až 3 X)
    (IX|IV|V?I{0,3})    # jednotky - 9 (IX), 4 (IV), 0-3 (0 až 3 I),
                        #        nebo 5-8 (V následované 0 až 3 I)
    $                   # konec řetězce
    ''', re.VERBOSE)
```

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):                                          [2]
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if not isinstance(n, int):
        raise NotIntegerError('non-integers can not be converted')

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def from_roman(s):
    .
    .
    .
```

[1] Funkci `from_roman()` nemusíme vůbec upravovat. Změna se týká jen vzorku `roman_numeral_pattern`. Při podrobnějším pohledu zjistíte, že jsem v první části regulárního výrazu změnil maximální počet nepovinných znaků `M` z `3` na `4`. Tím povolíme čísla odpovídající hodnotě až `4999` místo původní `3999`. Samotná funkce `from_roman()` je zcela obecná. Zkrátka jen hledá opakující se znaky římského čísla a sčítá odpovídající hodnoty. Nestará se o to, kolikrát se opakují. Dříve nezvládala `'MMMM'` pouze z toho důvodu, že jsme ji explicitně zastavili na základě porovnání s regulárním výrazem.

[2] Funkce `to_roman()` si vyžádá jen jednu malou změnu v místě kontroly rozsahu. Kde jsme dříve testovali `0 < n < 4000`, budeme teď kontrolovat `0 < n < 5000`. A hlášení o chybě vyvolávané příkazem `raise` změníme tak, aby odpovídalo novému povolenému rozsahu (`1..4999` místo `1..3999`). Zbytek funkce nemusíme měnit. Nové případy zvládá. (Vesele přidává `'M'` pro každou nalezenou tisícovku. Když dostane `4000` vychrlí `'MMMM'`. Dříve tento případ nezvládala jen proto, že jsme ji explicitně zastavili při kontrole rozsahu.)

Možná pochybujete o tom, že by tyhle dvě malé změny vyřešily vše, co potřebujeme. Nemusíte mi to věřit. Zkontrolujte si to sami.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
```

```
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.203s

OK                                                                    [1]
```

[1] Všechny testovací případy prošly. Přestaňte psát kód.

Při používání obsáhlých testů jednotek nemusíte spoléhat na programátora, který říká: „Věř mi.“

10.3. Refaktorizace

Na komplexním používání testů jednotek (unit testing) není nejlepší to, jak se cítíte, když všechny testovací případy nakonec projdou, dokonce ani to, jak se cítíte, když vás někdo nařkne, že jste mu pokazili jeho kód, a vy ve skutečnosti můžete *dokázat*, že tomu tak není. Na testech jednotek je nejlepší věcí to, že vám dává volnost nemilosrdně refaktorizovat.

Refaktorizace je činností, kdy vezmete fungující kód a uděláte z něj ještě lepší. „Lepší“ obvykle znamená „rychlejší“, ale může to taky znamenat „používající méně paměti“ nebo „používající menší diskový prostor“ nebo je prostě „elegantnější“. Refaktorizace je z hlediska dlouhodobého zdraví každého programu důležitá, ať už to znamená cokoliv pro vás, pro váš projekt nebo pro vaše okolí.

V případě našeho kódu bude „lepší“ znamenat jak „rychlejší“, tak „snadněji udržovatelný“. Konkrétně funkce `from_roman()` je pomalejší a složitější, než by se mi líbilo. Je to dáno oním velkým, hnusným regulárním výrazem, který se používá pro ověřování, zda jde o římské číslo. Teď si možná pomyslíte: „No jo. Ten regulární výraz sice je velký a střapatý, ale jak jinak by se dalo ověřit, zda je libovolný řetězec platným římským číslem?“

Odpověď zní: Těch čísel je jen 5000. Proč bychom pro ně prostě nemohli vytvořit vyhledávací tabulku? Ta myšlenka se vám bude líbit ještě víc, když zjistíte, že *vůbec nebudeme potřebovat regulární výrazy*. Při budování vyhledávací tabulky pro převod čísel na římská čísla můžeme současně vytvářet opačnou vyhledávací tabulku pro konverzi římských čísel na celá čísla. Při testu, zda je libovolný řetězec platným římským číslem, budeme mít k dispozici všechna platná římská čísla. „Ověření platnosti“ se redukuje na jedno vyhledání ve slovníku.

A ze všeho nejlepší je, že už máme k dispozici úplnou sadu testů jednotek (unit test). V modulu můžeme vyměnit klidně polovinu kódu, ale testy jednotek zůstanou stejné. To znamená, že můžete dokázat — sami sobě a ostatním —, že nový kód funguje stejně dobře jako ten původní.

```
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
class InvalidRomanNumeralError(ValueError): pass

roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
                     ('L',  50),
                     ('XL', 40),
                     ('X',  10),
                     ('IX', 9),
                     ('V',  5),
                     ('IV', 4),
                     ('I',  1))

to_roman_table = [ None ]
from_roman_table = {}

def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]
```

```
def build_lookup_tables():
    def to_roman(n):
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)
        to_roman_table.append(roman_numeral)
        from_roman_table[roman_numeral] = integer

build_lookup_tables()
```

Rozdělme si to na stravitelné kousky. Prokazatelně nejdůležitějším řádkem je ten poslední:

```
build_lookup_tables()
```

Jistě si všimnete, že jde o volání funkce. Ale není tu žádný obalující příkaz `if`. Tady nejde o blok uvnitř `if __name__ == '__main__'`. Funkce se zavolá *v okamžiku importu modulu*. (Zde je důležité vědět, že se moduly importují jen jednou a poté se pamatují ve vyrovnávací paměti (cache). Pokud importujeme už jednou importovaný modul, nic se neděje. Takže uvedený kód bude zavolán jen při prvním importu tohoto modulu.)

Co vlastně funkce `build_lookup_tables()` dělá? To jsem rád, že se ptáte.

```
to_roman_table = [ None ]
from_roman_table = {}
.
.
.
def build_lookup_tables():
    def to_roman(n):                                    [1]
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
```

```
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)                [2]
        to_roman_table.append(roman_numeral)             [3]
        from_roman_table[roman_numeral] = integer
```

[1] Tohle je takový chytrý programátorský obrat... možná až příliš chytrý. Funkce `to_roman()` je definována výše. Vyhledává hodnoty ve vyhledávací tabulce a vrací je. Ale funkce `build_lookup_tables()` si pro realizaci převodu vytváří svou vlastní definici funkce `to_roman()` (stejnou, jaká se používala v předchozích případech, než jsme přidali vyhledávací tabulku). Uvnitř funkce `build_lookup_tables()` se bude volat ta redefinovaná verze funkce `to_roman()`. Jakmile funkce `build_lookup_tables()` skončí, redefinovaná verze zmizí. Její definice je platná jen lokálně, uvnitř funkce `build_lookup_tables()`.

[2] Na tomto řádku kódu se volá redefinovaná funkce `to_roman()`, která ve skutečnosti vytváří římské číslo.

[3] Jakmile máme k dispozici výsledek (redefinované funkce `to_roman()`), přidáme číslo a jemu odpovídající římské číslo do obou vyhledávacích tabulek.

Jakmile jsou vyhledávací tabulky naplněny, je zbývající kód jednoduchý a rychlý.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]                                          [1]

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]                                        [2]
```

[1] Funkce `to_roman()` provede stejné kontroly hraničních případů (jako dříve) a potom jednoduše najde odpovídající hodnotu ve vyhledávací tabulce a vrátí ji.

[2] Také funkce `from_roman()` je redukována na kontroly a jeden řádek kódu. Už žádné regulární výrazy. Už žádné cykly. Převod na a z římského čísla se složitostí `O(1)` — tj. v konstantním čase.

Ale funguje to? Proč se ptáte? Jasně že funguje. A můžu to dokázat.

```
you@localhost:~/diveintopython3/examples$ python3 romantest10.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.031s                                                [1]

OK
```

[1] Tedy, ne že byste se ptali, ale ono je to taky rychlé! Skoro 10krát rychlejší. Není to, samozřejmě, úplně férové srovnání, protože u této verze trvá déle import (budují se vyhledávací tabulky). Ale protože se import dělá jen jednou, rozpustí se nákladnost při startu mezi volání funkcí `to_roman()` a `from_roman()`. A protože se při testech provádí několik tisíc volání funkcí (jen samotný kruhový test jich provede 10 000), úspory se rychle nasčítají!

A jak zní ponaučení?

- V jednoduchosti je síla.
- Zvláště tehdy, když jsou do toho zapletené regulární výrazy.
- Díky testům jednotek (unit test) získáte sebedůvěru a odvahu k provádění rozsáhlé refaktorizace.

10.4. Shrnutí

Unit testing (testování jednotek) představuje mocný koncept, který při správné implementaci vede u dlouhodobých projektů jak k redukci nákladů na údržbu, tak ke zvýšení pružnosti. Současně si ale musíme uvědomit, že testování jednotek není všelék. Napsat dobré testové případy není jednoduchá věc a udržet je v aktuálním stavu vyžaduje disciplínu (zvlášť když zákazníci vřískají, aby byly opra-

veny kritické chyby). Unit testing není náhradou ostatních forem testování, zahrnujících testování funkčnosti celého systému, integrační testování (tj. test spolupráce jednotek) a uživatelské akceptační testy. Testy jednotek jsou ale přesto rozumné, fungují, a když už je jednou uvidíte v činnosti, budete se divit, jak jste se bez nich mohli obejít.

V pár posledních kapitolách jsme se šířeji zabývali základy, z nichž mnohé dokonce nejsou specifické jen pro Python. Rámce pro testování jednotek (unit testing frameworks) jsou dostupné pro mnoho jazyků a všechny vyžadují, abyste porozuměli týmž konceptům:

- Návrh testovacích případů (test case), které jsou specifické, automatizované a nezávislé.
- Napsání testovacích případů *před psaním kódu*, který mají testovat.
- Psaní testů, které testují správné vstupy a kontrolují očekávané výsledky.
- Psaní testů, které testují chybné vstupy a kontrolují očekávané chybové reakce.
- Psaní a aktualizace testovacích případů tak, aby odrážely nové požadavky.
- Nemilosrdná refaktorizace za účelem zvýšení výkonnosti, škálovatelnosti, čitelnosti, udržovatelnosti a jakýchkoliv jiných -ostí, po kterých toužíte.

“ A nine mile walk is no joke, especially in the rain.”
(Jít devět mil není žádná legrace, zvlášť v dešti.)

— Harry Kemelman, *The Nine Mile Walk*

11. Soubory

11. Soubory — 235
11.1. Ponořme se — **237**
11.2. Čtení z textových souborů — **237**
11.2.1. Kódování znaků vystrkuje svou ošklivou hlavu — **237**
11.2.2. Objekty typu stream — **238**
11.2.3. Čtení dat z textového souboru — **239**
11.2.4. Zavírání souborů — **241**
11.2.5. Automatické zavírání souborů — **242**
11.2.6. Čtení dat po řádcích — **243**
11.3. Zápis do textových souborů — **245**
11.3.1. A znovu kódování znaků — **246**
11.4. Binární soubory — **246**
11.5. Objekty typu stream z nesouborových zdrojů — **247**
11.5.1. Práce s komprimovanými soubory — **249**
11.6. Standardní vstup, výstup a chybový výstup — **250**
11.6.1. Přesměrování standardního výstupu — **251**
11.7. Přečtěte si — **254**

11.1. Ponořme se
11.2. Čtení z textových souborů

11.1. Ponořme se

Než jsem začal instalovat první aplikaci, obsahovaly Windows na mém laptopu 38 493 souborů. Po instalaci Pythonu 3 k nim přibylo téměř 3000 dalších. Každý významnější operační systém považuje soubory za základ ukládání dat. Koncepce souborů je tak zakořeněná, že by představa jiné možnosti dělala většině lidí problémy. Obrazně řečeno, váš počítač se topí v souborech.

11.2. Čtení z textových souborů

Než můžeme ze souboru číst, musíme jej otevřít. Otvírání souborů v Pythonu už nemohlo být jednodušší.

```
a_file = open('examples/chinese.txt', encoding='utf-8')
```

V Pythonu najdeme zabudovanou funkci `open()`, která přebírá jméno souboru jako argument. Jménem souboru je zde `'examples/chinese.txt'`. Na uvedeném jméně souboru najdeme pět zajímavostí:

1. Není to pouhé jméno souboru. Je to kombinace adresářové cesty a jména souboru. Hypotetická funkce pro otvírání souboru by mohla požadovat dva argumenty — adresářovou cestu a jméno souboru. Ale funkce `open()` požaduje jen jeden. Kdykoliv se po vás v Pythonu požaduje „jméno souboru“, můžete do něj zahrnout také celou adresářovou cestu nebo její část.
2. Uvedená adresářová cesta používá normální lomítko, ale neupřesnil jsem, jaký operační systém používám. Windows používají pro oddělování podadresářů zpětná lomítka, zatímco Mac OS X a Linux používají obyčejná lomítka. Ale v Pythonu fungují obyčejná lomítka i pod Windows.
3. Uvedená adresářová cesta nezačíná lomítkem nebo písmenem disku, takže ji nazýváme *relativní cesta*. Mohli byste se zeptat — relativní k čemu? Zachovejte klid.
4. Je to řetězec. Všechny moderní operační systémy (dokonce i Windows!) ukládají jména souborů a adresářů v Unicode. Python 3 plně podporuje jména cest, která nemusí být výhradně v `ASCII`.
5. A nemusí vést jen na váš lokální disk. Můžete mít připojený síťový disk. Daný „soubor“ může být fiktivní součástí zcela virtuálního souborového systému. Pokud jej váš počítač považuje za soubor a může k němu jako k souboru přistupovat, může jej Python otevřít také.

Ale volání funkce `open()` nekončí zadáním jména souboru. Máme zde další argument nazvaný `encoding` (kódování). No nazdar. To zní příšerně povědomě.

11.2.1. Kódování znaků vystrkuje svou ošklivou hlavu

Bajty jsou bajty, znaky jsou abstrakce. Řetězec je posloupností znaků v Unicode. Ale soubor na disku není posloupností Unicode znaků. Soubor na disku je posloupností bajtů. Takže jak Python převádí posloupnost bajtů na posloupnost znaků, když čteme „textový soubor“ z disku? Dekóduje bajty podle určitého algoritmu pro kódování znaků a vrací posloupnost znaků v Unicode (známou také jako řetězec).

```
# Tento příklad byl vytvořen pod Windows. Z důvodů popsaných
# níže se na ostatních platformách může chovat jinak.
>>> file = open('examples/chinese.txt')
>>> a_string = file.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python31\lib\encodings\cp1252.py", line 23, in decode
    return codecs.charmap_decode(input,self.errors,decoding_table)[0]
UnicodeDecodeError: 'charmap' codec can't decode byte 0x8f in position 28: character maps to <undefined>
>>>
```

Výchozí kódování je závislé na platformě.

Co se to vlastně stalo? Neurčili jsme znakové kódování, takže Python byl donucen použít výchozí kódování. Co to je výchozí kódování? Pokud se pořádně podíváme na trasovací výpis, vidíme, že skončil uvnitř `cp1252.py`. To znamená, že Python použil jako výchozí kódování CP-1252. (CP-1252 je běžné kódování, které se používá na počítačích s Microsoft Windows. To se týká západní Evropy. Čeština a slovenština používají kódování CP-1250.) Znaková sada CP-1252 nepodporuje znaky, které se v souboru nacházejí, takže čtení selhává s nepěknou chybou `UnicodeDecodeError`.

Ale počkejte. Ono je to ještě horší! Výchozí kódování je *závislé na platformě*, takže stejný kód by na vašem počítači fungovat *mohl* (pokud by vaším výchozím kódováním bylo `UTF-8`). Ale pokud program přenesete k někomu jinému (kdo používá jiné výchozí kódování, jako třeba CP-1252), dojde k selhání.

> Pokud potřebujete zjistit výchozí znakové kódování, importujte modul `locale` a zavolejte `locale.getpreferredencoding()`. Na mém laptopu s Windows funkce vrací `'cp1252'`, ale na mém linuxovém stroji v horním pokoji se vrací `'UTF8'`. Nejsem schopen udržet shodu dokonce ani ve svém vlastním domě! Ve vašem případě mohou být výsledky jiné (dokonce i pod Windows) v závislosti na verzi operačního systému, který jste nainstalovali, a na konfiguraci regionálních a jazykových nastavení. To je důvod, proč je tak důležité uvádět kódování pokaždé, když otvíráme soubor.

11.2.2. Objekty typu stream

Zatím víme jen to, že Python má zabudovanou funkci zvanou `open()`. Funkce `open()` vrací *objekt typu stream* (čti [strím], proud dat), který poskytuje metody a atributy pro získávání informací o proudu znaků a pro manipulaci s ním.

```
>>> a_file = open('examples/chinese.txt', encoding='utf-8')
>>> a_file.name                                                    [1]
'examples/chinese.txt'
>>> a_file.encoding                                                [2]
'utf-8'
>>> a_file.mode                                                    [3]
'r'
```

[1] Atribut `name` zachycuje jméno, které jsme při otvírání souboru předali funkci `open()`. Není upraveno do podoby absolutní cesty.

[2] Podobně atribut `encoding` zachycuje kódování, které jsme při otvírání souboru předali funkci `open()`. Pokud byste při otvírání souboru kódování neuvedli (nepořádný vývojář!), pak by atribut `encoding` odpovídal výsledku `locale.getpreferredencoding()`.

[3] Z atributu `mode` poznáme, v jakém režimu byl soubor otevřen. Funkci `open()` můžeme předat nepovinný parametr `mode` (režim). Při otvírání tohoto souboru jsme režim neurčili, takže Python použije výchozí hodnotu `'r'`, která má význam „otevřít jen pro čtení, v textovém režimu“. Jak uvidíme v této kapitole později, plní režim otevření souboru několik účelů. Různé režimy nám umožní do souboru zapisovat, připojovat na konec souboru nebo otvírat soubor v binárním režimu (ve kterém místo s řetězci pracujeme s bajty).

> Seznam všech možných režimů najdete v dokumentaci pro funkci `open()`.

11.2.3. Čtení dat z textového souboru

Po otevření souboru pro čtení z něj pravděpodobně v určitém místě budete chtít číst.

```
>>> a_file = open('examples/chinese.txt', encoding='utf-8')
>>> a_file.read()                                                  [1]
'Dive Into Python 是为有经验的程序员编写的一本 Python 书。\n'
>>> a_file.read()                                                  [2]
''
```

[1] Jakmile soubor otevřeme (při zadání správného kódování), spočívá čtení z něj v prostém volání metody `read()` objektu typu stream. Výsledkem je řetězec.

[2] Trochu překvapující je možná to, že další čtení ze souboru nevyvolá výjimku. Python nepovažuje čtení za koncem souboru za chybu. Vrátí se jednoduše prázdný řetězec.

A co kdybychom chtěli soubor číst znovu?

Při otvírání souboru vždy uvádějte parametr encoding.

```
# pokračování předchozího příkladu
>>> a_file.read()                                [1]
''
>>> a_file.seek(0)                               [2]
0
>>> a_file.read(16)                              [3]
'Dive Into Python'
>>> a_file.read(1)                               [4]
' '
>>> a_file.read(1)
'是'
>>> a_file.tell()                                [5]
20
```

[1] Protože jsme ještě pořád na konci souboru, další volání metody `read()` vrací prázdný řetězec.
[2] Metoda `seek()` zajistí přesun v souboru na určenou bajtovou pozici.
[3] Metodě `read()` můžeme zadat nepovinný parametr, který určuje počet znaků, které se mají načíst.
[4] Pokud budeme chtít, můžeme číst klidně i po jednom znaku.
[5] 16 + 1 + 1 = … 20?

Zkusme to znovu.

```
# pokračování předchozího příkladu
>>> a_file.seek(17)                              [1]
17
>>> a_file.read(1)                               [2]
'是'
>>> a_file.tell()                                [3]
20
```

[1] Přesuneme se na 17. bajt.
[2] Přečteme jeden znak.
[3] A najednou jsme na 20. bajtu.

Už jste na to přišli? Metody `seek()` a `tell()` počítají vždy po bajtech, ale protože jsme soubor otevřeli v textovém režimu, čte metoda `read()` *po znacích*. Pro zakódování čínských znaků v `UTF-8` potřebujeme více bajtů. Pro každý anglický znak potřebujeme v souboru jen jeden bajt, takže by vás to mohlo svést k mylnému závěru, že metody `seek()` a `read()` počítají stejné jednotky. To ale platí jen pro některé znaky.

Ale moment, začíná to být ještě horší!

```
>>> a_file.seek(18)                                [1]
18
>>> a_file.read(1)                                 [2]
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a_file.read(1)
  File "C:\Python31\lib\codecs.py", line 300, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0x98 in position 0: unexpected code byte
```

[1] Přesuneme se na 18. bajt a zkusíme přečíst jeden znak.

[2] Proč to selhalo? Protože na 18. bajtu není znak. Nejbližší znak začíná na 17. bajtu (a zabírá tři bajty). Pokus o čtení znaku od středu jeho kódované posloupnosti vede k chybě `UnicodeDecodeError`.

11.2.4. Zavírání souborů

Otevřené soubory zabírají systémové prostředky a v závislosti na režimu otevření souboru k nim některé programy nemusí být schopny přistupovat. Proto je důležité, abychom soubory zavírali hned poté, co s nimi přestaneme pracovat.

```
# pokračování předchozího příkladu
>>> a_file.close()
```

Tak *tohle* bylo zklamání.

Objekt `a_file` typu stream pořád existuje. Volání jeho metody `close()` nevede k jeho zrušení. Ale už není nějak zvlášť užitečný.

```
# pokračování předchozího příkladu
>>> a_file.read()                                  [1]
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    a_file.read()
ValueError: I/O operation on closed file.
>>> a_file.seek(0)                                 [2]
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    a_file.seek(0)
ValueError: I/O operation on closed file.
>>> a_file.tell()                                  [3]
```

```
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    a_file.tell()
ValueError: I/O operation on closed file.
>>> a_file.close()                                [4]
>>> a_file.closed                                 [5]
True
```

[1] Ze zavřeného objektu nemůžeme číst. Vyvolá se tím výjimka `IOError`.

[2] V zavřeném souboru nemůžeme ani přesunovat pozici (seek).

[3] U zavřeného souboru neexistuje žádná aktuální pozice, takže metoda `tell()` také selže.

[4] Překvapením možná je, že volání metody `close()` pro objekt typu stream, jehož soubor byl už zavřený, nevyvolá výjimku. Jde o prázdnou operaci.

[5] Zavřený objekt typu stream má přece jen jeden užitečný atribut. Atribut `closed` potvrzuje, že soubor byl uzavřen.

11.2.5. Automatické zavírání souborů

Objekty typu stream mají explicitní metodu `close()`, ale co se stane, když je ve vašem programu chyba a zhavaruje předtím, než zavoláte `close()`? Soubor by teoreticky mohl zůstat otevřený mnohem déle, než bychom potřebovali. Pokud zrovna něco ladíte na svém lokálním počítači, není to takový problém. Ale na používaném serveru už možná ano.

Konstrukce try..finally je dobrá. with je lepší.

Python 2 pro tento případ nabízel řešení v podobě bloku `try..finally`. V Pythonu 3 tento obrat stále funguje. Proto se s ním můžete setkat v kódu některých programátorů nebo ve starším kódu, který byl převeden pro Python 3. Ale Python 2.6 zavedl čistší řešení, které se v Pythonu 3 stalo preferovaným. Jde o příkaz `with`.

```
with open('examples/chinese.txt', encoding='utf-8') as a_file:
    a_file.seek(17)
    a_character = a_file.read(1)
    print(a_character)
```

V tomto kódu se volá `open()`, ale nikde se v něm nevolá `a_file.close()`. Příkaz `with` zahajuje blok kódu podobně, jako je tomu u příkazu `if` nebo u cyklu `for`. Uvnitř bloku kódu můžeme používat proměnnou `a_file`, kterou objekt typu stream vrátil jako výsledek volání `open()`. K dispozici máme všechny obvyklé metody objektu typu stream, jako jsou `seek()`, `read()` a všechny ostatní. Když blok `with` skončí, *Python automaticky* zavolá `a_file.close()`.

Když to shrneme, Python soubor uzavře nezávisle na tom, jak a kdy blok `with` skončí... i kdyby „skončil“ v důsledku neošetřené výjimky. Tak to opravdu je. I v případě, kdy kód vyvolá výjimku a celý váš program se skřípěním zastaví, dotčený soubor bude uzavřen. Je to zaručeno.

> Z technického pohledu příkaz `with` vytváří operační kontext (runtime context). Objekt typu stream je v těchto příkladech využit jako správce kontextu (context manager). Python vytvoří objekt `a_file` typu stream a řekne mu, že vstupuje do operačního kontextu. Jakmile blok příkazu with skončí, Python sdělí objektu typu stream, že opouští operační kontext a objekt zavolá svou vlastní metodu `close()`. Detaily hledejte v příloze B, „Třídy, které mohou být použity v bloku with“.

Kap.

Příkaz `with` není nijak zvlášť zaměřen na soubory. Je to prostě obecný rámec pro vytvoření operačního kontextu. Objekt se dozví, že vstupuje do operačního kontextu nebo že z něj vystupuje. Pokud je dotčený objekt typu stream, pak provede užitečné „souborové“ věci (jako je například automatické uzavření souboru). Ale toto chování je definováno uvnitř objektu typu stream a ne v příkazu `with`. Správce kontextu může být použit mnoha jinými způsoby, které nemají se soubory nic společného. Můžete si dokonce vytvořit svého vlastního správce kontextu. Ukážeme si to o něco později, ale ještě v této kapitole.

11.2.6. Čtení dat po řádcích

„Řádek“ textového souboru je to, co si myslíte, že by to mělo být — napíšete pár slov, stisknete ENTER a najednou jste na novém řádku. Řádek textu je posloupnost znaků oddělená... čím vlastně? Ono je to komplikované, protože textové soubory mohou pro označení konce řádků použít několik různých znaků. Každý operační systém má svou vlastní konvenci. Některé používají znak návratu vozíku (carriage return), jiné používají znak přechodu na nový řádek (line feed) a některé používají na konci každého řádku oba zmíněné znaky.

Teď si můžete s úlevou oddechnout, protože *Python zpracovává konce řádků automaticky*. Pokud řeknete „chci přečíst tento textový soubor řádek po řádku“, Python zjistí, který typ konců řádků se v textovém souboru používá, a zařídí, že to prostě bude fungovat.

> Pokud potřebujete získat detailní kontrolu nad tím, co se považuje za konec řádku, můžete funkci `open()` předat nepovinný parametr `newline`. Detaily najdete v dokumentaci funkce `open()`.

Takže jak se to vlastně dělá? Čtěte ze souboru po řádcích. Je to tak jednoduché. V jednoduchosti je krása.

```
line_number = 0
with open('examples/favorite-people.txt', encoding='utf-8') as a_file:    [1]
    for a_line in a_file:                                                  [2]
        line_number += 1
        print('{:>4} {}'.format(line_number, a_line.rstrip()))             [3]
```

[1] Použitím vzoru `with` dosáhneme bezpečného otevření souboru a necháme Python, aby ho zavřel za nás.

[2] Pro čtení souboru po řádcích využijeme cyklus `for`. To je vše. Objekty typu stream podporují metody jako `read()`, ale kromě toho *je objekt typu stream také iterátorem*, který vrátí jeden řádek pokaždé, když jej požádáte o další hodnotu.

[3] Číslo řádku a řádek samotný můžeme zobrazit s využitím řetězcové metody `format()`. Specifikátor formátu `{:>4}` říká „vytiskni tento argument zarovnaný doprava na šířku čtyř pozic“. Proměnná `a_line` obsahuje celý řádek, včetně znaků ukončujících řádek. Řetězcová metoda `rstrip()` odstraní všechny koncové bílé znaky (whitespace) včetně znaků ukončujících řádek.

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
   1 Dora
   2 Ethan
   3 Wesley
   4 John
   5 Anne
   6 Mike
   7 Chris
   8 Sarah
   9 Alex
  10 Lizzie
```

Setkali jste se s následující chybou?

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
Traceback (most recent call last):
  File "examples/oneline.py", line 4, in <module>
    print('{:>4} {}'.format(line_number, a_line.rstrip()))
ValueError: zero length field name in format
```

Pokud ano, pravděpodobně používáte Python 3.0. Měli byste provést aktualizaci na Python 3.1.

Python 3.0 sice podporuje nový způsob formátování řetězců, ale vyžaduje explicitní formátování specifikátorů formátu. Python 3.1 vám umožní ve specifikátorech formátu indexy argumentů vynechávat. Verze kompatibilní s Pythonem 3.0 je pro porovnání zde:

```
print('{0:>4} {1}'.format(line_number, a_line.rstrip()))
```

11.3. Zápis do textových souborů

Soubor prostě otevřete a začněte zapisovat.

Do souborů můžeme zapisovat velmi podobným způsobem, jakým z nich čteme. Soubor nejdříve otevřeme a získáme objekt typu stream. Pro zápis do souboru použijeme jeho metody. Nakonec soubor zavřeme.

Při otvírání souboru pro zápis použijeme funkci `open()` a předepíšeme režim zápisu. U souborů můžeme použít dva režimy zápisu:

- Režim „write“ (zápis) vede k přepsání obsahu souboru. Funkci `open()` předáme `mode='w'`.
- Režim „append“ přidává data na konec souboru. Funkci `open()` předáme `mode='a'`.

Pokud soubor dosud neexistuje, bude při obou uvedených režimech vytvořen automaticky. To znamená, že se nikdy nemusíme piplat s funkčností jako „pokud soubor ještě neexistuje, vytvoř nový, prázdný soubor, abychom jej mohli poprvé otevřít“. Prostě soubor otevřeme a začneme zapisovat.

Jakmile zápis do souboru dokončíme, měli bychom jej vždy zavřít, aby došlo k uvolnění deskriptoru souboru (file handle) a abychom zajistili, že došlo ke skutečnému zápisu dat na disk. Stejně jako v případě čtení dat můžeme soubor zavřít voláním metody `close()` objektu typu stream nebo můžeme použít příkaz `with` a předat starost o zavření souboru Pythonu. Vsadím se, že uhodnete, kterou techniku doporučuji.

```
>>> with open('test.log', mode='w', encoding='utf-8') as a_file:          [1]
...     a_file.write('test succeeded')                                    [2]
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
test succeeded
>>> with open('test.log', mode='a', encoding='utf-8') as a_file:          [3]
...     a_file.write('and again')
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
test succeededand again                                                  [4]
```

[1] Začali jsme odvážně vytvořením nového souboru `test.log` (nebo přepsáním existujícího souboru) a jeho otevřením pro zápis. Parametr `mode='w'` znamená „otevři soubor pro zápis“. Ano, je to opravdu tak nebezpečné, jak to zní. Doufám, že vám na dřívějším obsahu tohoto souboru nezáleželo (pokud existoval), protože jeho obsah právě zmizel.

[2] Do nově otevřeného souboru můžeme data přidávat metodou `write()` objektu, který vrátila funkce `open()`. Jakmile blok `with` skončí, Python soubor automaticky uzavře.

[3] To bylo zábavné. Zkusme to znovu. Ale tentokrát použijeme `mode='a'`, abychom místo přepsání souboru připojili data na jeho konec. Připsání na konec (append) *nikdy* nezničí existující obsah souboru.

[4] Jak původně zapsaný řádek, tak druhý řádek, který jsme připojili teď, se nacházejí v souboru `test.log`. Všimněte si také, že nepřibyly žádné znaky pro návrat vozíku nebo pro odřádkování. Soubor je neobsahuje, protože jsme je do něj ani při jedné příležitosti explicitně nezapsali. Znak pro návrat vozíku (carriage return) můžeme zapsat jako `'\r'`, znak pro odřádkování (line feed) můžeme zapsat `'\n'`. Protože jsme nic z toho neudělali, skončilo vše, co jsme zapsali do souboru, na jediném řádku.

11.3.1. A znovu kódování znaků

Všimli jste si parametru `encoding`, který jsme při otvírání souboru pro zápis předávali funkci `open()`? Je důležitý. Nikdy ho nevynechávejte! Jak jsme si ukázali na začátku kapitoly, soubory neobsahují řetězce. Soubory obsahují bajty. Z textového souboru můžeme číst „řetězce“ jen díky tomu, že jsme Pythonu řekli, jaké má při převodu proudu bajtů na řetězec použít kódování. Zápis textu do souboru představuje stejný problém, jen z opačné strany. Do souboru nemůžeme zapisovat znaky, protože znaky jsou abstraktní. Při zápisu do souboru musí Python vědět, jak má řetězce převádět na posloupnost bajtů. Jediný způsob, jak se ujistit, že se provede správný převod, spočívá v uvedení parametru `encoding` při otvírání souboru pro zápis.

11.4. Binární soubory

Všechny soubory neobsahují text. Některé mohou obsahovat obrázky mého psa.

```
>>> an_image = open('examples/beauregard.jpg', mode='rb')                [1]
>>> an_image.mode                                                        [2]
'rb'
>>> an_image.name                                                        [3]
'examples/beauregard.jpg'
>>> an_image.encoding                                                    [4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_io.BufferedReader' object has no attribute 'encoding'
```

[1] Otevření souboru v binárním režimu je jednoduché, ale záludné. Ve srovnání s otvíráním v textovém režimu spočívá jediný rozdíl v tom, že parametr `mode` obsahuje znak `'b'`.

[2] Objekt typu stream, který získáme otevřením souboru v binárním režimu, má mnoho stejných atributů, včetně atributu `mode`, který odpovídá stejnojmennému parametru předanému funkci `open()`.

[3] Binární objekty typu stream mají také atribut `name` — stejně jako textové objekty typu stream.

[4] Ale jeden rozdíl tady přesto je. Binární objekty typu stream nemají atribut `encoding`. Dává to smysl, že? Čteme (nebo zapisujeme) bajty a ne řetězce. Python tedy nemusí dělat žádný převod. Z binárního souboru dostaneme přesně to, co jsme do něj vložili. Žádná konverze není nutná.

Už jsem řekl, že čteme bajty? Ano, je to tak.

```
# pokračování předchozího příkladu
>>> an_image.tell()
0
>>> data = an_image.read(3)              [1]
>>> data
b'\xff\xd8\xff'
>>> type(data)                           [2]
<class 'bytes'>
>>> an_image.tell()                      [3]
3
>>> an_image.seek(0)
0
>>> data = an_image.read()
>>> len(data)
3150
```

[1] Stejně jako v případě textových souborů také z binárních souborů můžeme číst po kouscích. Ale je tu jeden zásadní rozdíl...

[2] ... čteme bajty, ne řetězce. Protože jsme soubor otevřeli v binárním režimu, přebírá metoda `read()` jako argument počet *bajtů*, které se mají načíst, a ne počet znaků.

[3] To znamená, že zde nikdy nedojde k neočekávanému nesouladu mezi číslem, které jsme předali metodě `read()`, a pozičním indexem, který nám vrací metoda `tell()`. Metoda `read()` čte bajty a metody `seek()` a `tell()` sledují počet přečtených bajtů. U binárních souborů budou vždy v souladu.

11.5. Objekty typu stream z nesouborových zdrojů

Představte si, že píšete knihovnu a jedna z vašich knihovních funkcí má číst data ze souboru. Funkce by mohla jednoduše převzít jméno souboru v řetězcové podobě, otevřít soubor pro čtení, přečíst jeho obsah a před skončením funkce jej uzavřít. Ale takhle byste to dělat neměli. Místo toho by rozhraní vaší funkce (API) mělo přebírat **libovolný objekt typu stream**.

Z předstíraného souboru čteme jednoduše voláním read().

V nejjednodušším případě je objektem typu stream cokoliv, co má metodu `read()`, která přebírá nepovinný parametr `size` (velikost) a vrací řetězec. Pokud je metoda `read()` zavolána bez uvedení parametru `size`, měla by ze zdroje informací přečíst všechna zbývající data a vrátit je jako jednu hodnotu. Pokud je metoda zavolána s paramet-

rem `size`, přečte ze zdroje požadované množství dat a vrátí je. Pokud je zavolána znovu, pokračuje od místa, kde se čtením přestala, a vrací další část dat.

Vypadá to, jako kdybychom používali objekt typu stream vzniklý otevřením skutečného souboru. Rozdíl je v tom, že se *neomezujeme na skutečné soubory*. Zdrojem informací, ze kterého „čteme“, může být cokoliv: webová stránka, řetězec v paměti nebo dokonce výstup z jiného programu. Pokud vaše funkce přebírá objekt typu stream a jednoduše volá jeho metodu `read()`, můžete zpracovávat libovolný zdroj informací, který se tváří jako soubor, aniž byste museli pro každý druh vstupu psát různý kód.

```
>>> a_string = 'PapayaWhip is the new black.'
>>> import io                                    [1]
>>> a_file = io.StringIO(a_string)               [2]
>>> a_file.read()                                [3]
'PapayaWhip is the new black.'
>>> a_file.read()                                [4]
''
>>> a_file.seek(0)                               [5]
0
>>> a_file.read(10)                              [6]
'PapayaWhip'
>>> a_file.tell()
10
>>> a_file.seek(18)
18
>>> a_file.read()
'new black.'
```

[1] Modul `io` definuje třídu `StringIO`, kterou můžeme dosáhnout toho, aby se řetězec v paměti choval jako soubor.

[2] Když chceme z řetězce vytvořit objekt typu stream, vytvoříme instanci třídy `io.StringIO()` a předáme jí řetězec, který chceme použít jako zdroj „souborových“ dat. Teď máme k dispozici objekt typu stream a můžeme s ním dělat všechny možné odpovídající věci.

[3] Voláním metody `read()` „přečteme“ celý „soubor“. V takovém případě objekt třídy `StringIO` jednoduše vrátí původní řetězec.

[4] Opakované volání metody `read()` vrací prázdný řetězec — stejně jako u opravdového souboru.

[5] Použitím metody `seek()` objektu třídy `StringIO` se můžeme explicitně nastavit na začátek řetězce — stejně jako při volání téže metody u opravdového souboru.

[6] Pokud metodě `read()` předáme parametr `size`, můžeme číst po větších kouscích i z řetězce.

> Třída `io.StringIO` vám umožní chovat se k řetězci jako k textovému souboru. Existuje také třída `io.BytesIO`, která vám umožní chovat se k poli bajtů jako k binárnímu souboru.

11.5.1. Práce s komprimovanými soubory

Pythonovská standardní knihovna obsahuje moduly, které podporují čtení a zápis komprimovaných souborů. Různých komprimačních schémat existuje celá řada. Mezi newindowsovskými systémy patří mezi dva nejpopulárnější `gzip` a `bzip2`. (Mohli jste se setkat také s archivy PKZIP a s archivy GNU Tar. V Pythonu najdete moduly i pro tyto dva.)

Modul `gzip` nám umožní vytvořit objekt typu stream pro čtení a zápis souborů komprimovaných algoritmem `gzip`. Příslušný objekt podporuje metodu `read()` (pokud jsme jej otevřeli pro čtení) nebo metodu `write()` (pokud jsme jej otevřeli pro zápis). To znamená, že *k přímému zápisu nebo čtení souborů komprimovaných algoritmem gzip* můžeme použít metody, které jsme se už naučili používat s normálními soubory. Nemusíme vytvářet pomocné soubory k ukládání dekomprimovaných dat.

Jako bonus navíc podporuje modul `gzip` i příkaz `with`, takže uzavření komprimovaného souboru můžete ponechat na Pythonu.

```
you@localhost:~$ python3

>>> import gzip
>>> with gzip.open('out.log.gz', mode='wb') as z_file:                                    [1]
...   z_file.write('A nine mile walk is no joke, especially in the rain.'.encode('utf-8'))
...
>>> exit()

you@localhost:~$ ls -l out.log.gz                                                         [2]
-rw-r--r--  1 mark mark    79 2009-07-19 14:29 out.log.gz
you@localhost:~$ gunzip out.log.gz                                                        [3]
you@localhost:~$ cat out.log                                                              [4]
A nine mile walk is no joke, especially in the rain.
```

[1] Soubory zabalené `gzip` bychom měli vždy otvírat v binárním režimu. (Všimněte si znaku `'b'` v argumentu `mode`.)

[2] Tento příklad jsem vytvořil na Linuxu. Pokud vám tento příkazový řádek nic neříká, zobrazuje výpis položky souboru „v dlouhém formátu“ (v pracovním adresáři). Soubor jsme právě vytvořili v pythonovském shellu s využitím komprese `gzip`. Tento soubor ukazuje, že soubor existuje (fajn) a že má velikost 79 bajtů. Ve skutečnosti je větší než řetězec, se kterým jsme začali! Soubor ve formátu gzip zahrnuje hlavičku pevné délky, která obsahuje nějaké informace o souboru. Pro velmi malé soubory je to tedy neefektivní.

[3] Příkaz `gunzip` (vyslovuje se „dží anzip“) dekomprimuje daný soubor a ukládá jeho obsah do nového souboru se stejným jménem, ale bez přípony `.gz`.

[4] Příkaz `cat` zobrazuje obsah souboru. Soubor obsahuje řetězec, který jsme původně zapsali v pythonovském shellu přímo do komprimovaného souboru `out.log.gz`.

Setkali jste se s následující chybou?

```
>>> with gzip.open('out.log.gz', mode='wb') as z_file:
...         z_file.write('A nine mile walk is no joke, especially in the
rain.'.encode('utf-8'))
...
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'GzipFile' object has no attribute '__exit__'
```

Pokud ano, pravděpodobně používáte Python 3.0. Měli byste provést aktualizaci na Python 3.1.

V Pythonu 3.0 se sice modul `gzip` nacházel, ale nepodporoval použití objektů komprimovaných souborů jako správců kontextu. V Pythonu 3.1 byla přidána možnost používat objekty `gzip` souborů i v příkazu `with`.

11.6. Standardní vstup, výstup a chybový výstup

Machři na práci přes příkazový řádek už koncept standardního vstupu, standardního výstupu a standardního chybového výstupu znají. Tato podkapitola je určena těm ostatním.

sys.stdin, sys.stdout, sys.stderr.

Standardní výstup a standardní chybový výstup (běžně se zkracují jako `stdout` a `stderr`) jsou roury (pipe), které jsou zabudovány do každého systému, který je odvozen od UNIXU. Platí to i pro Mac OS X a pro Linux. Pokud voláte funkci `print()`, tištěný obsah je odeslán do roury `stdout`. Pokud váš program zhavaruje a tiskne trasovací výpis, posílá jej do roury `stderr`. Ve výchozím stavu jsou obě uvedené roury napojeny na terminálové okno, ve kterém pracujete. Když váš program něco tiskne, zobrazuje se jeho výstup ve vašem terminálovém okně. Když program zhavaruje, vidíte trasovací výpis také ve svém terminálovém okně. V grafickém pythonovském shellu jsou roury `stdout` and `stderr` přesměrovány do vašeho „interaktivního okna“.

```
>>> for i in range(3):
...     print('PapayaWhip')                    [1]
PapayaWhip
PapayaWhip
PapayaWhip
>>> import sys
>>> for i in range(3):
```

```
...     l = sys.stdout.write('is the')          [2]
is theis theis the
>>> for i in range(3):
...     l = sys.stderr.write('new black')       [3]
new blacknew blacknew black
```

[1] Funkce `print()` volaná v cyklu. Tady nic překvapujícího nenajdeme.

[2] `stdout` je definován v modulu `sys` a jde o objekt typu stream. Když zavoláme jeho metodu `write()`, vytiskne každý řetězec, který jí předáme. Funkce `print` ve skutečnosti dělá právě tohle. Na konec každého tištěného řetězce přidá znak ukončující řádek a pak volá `sys.stdout.write`.

[3] V nejjednodušším případě posílají `sys.stdout` a `sys.stderr` výstup do stejného místa: do pythonovského integrovaného vývojového prostředí (`IDE`, pokud v něm pracujeme) nebo do terminálového okna (pokud Python spouštíme z příkazového řádku). Standardní chybový výstup (stejně jako standardní výstup) přechod na nový řádek nepřidávají. Pokud chceme přejít na nový řádek, musíme zapsat příslušné znaky pro přechod na nový řádek.

`sys.stdout` a `sys.stderr` jsou objekty typu stream, ale dá se do nich pouze zapisovat. Pokus o volání jejich metody `read()` vždy vyvolá výjimku `IOError`.

```
>>> import sys
>>> sys.stdout.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: not readable
```

11.6.1. Přesměrování standardního výstupu

`sys.stdout` a `sys.stderr` jsou objekty typu stream, i když podporují pouze zápis. Ale nejsou konstantní. Jde o proměnné. To znamená, že do nich můžeme přiřadit novou hodnotu — nějaký jiný objekt typu stream — a přesměrovat jejich výstup.

```
import sys

class RedirectStdoutTo:
    def __init__(self, out_new):
        self.out_new = out_new

    def __enter__(self):
        self.out_old = sys.stdout
        sys.stdout = self.out_new
```

```
    def __exit__(self, *args):
        sys.stdout = self.out_old

print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Podívejte se na tohle:

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
A
C
you@localhost:~/diveintopython3/examples$ cat out.log
B
```

Setkali jste se s následující chybou?

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
  File "stdout.py", line 15
    with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
                                                              ^
SyntaxError: invalid syntax
```

Pokud ano, pravděpodobně používáte Python 3.0. Měli byste provést aktualizaci na Python 3.1.

Python 3.0 podporoval příkaz `with`, ale každý příkaz mohl používat jen jednoho správce kontextu. Python 3.1 umožňuje použít v jednom příkazu `with` více správců kontextu.

Podívejme se nejdříve na poslední část.

```
print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Tenhle příkaz `with` je docela komplikovaný. Přepíšu ho do trochu srozumitelnější podoby.

```
with open('out.log', mode='w', encoding='utf-8') as a_file:
    with RedirectStdoutTo(a_file):
        print('B')
```

Z přepisu je vidět, že ve skutečnosti jde o dva příkazy `with`, z nichž jeden je zanořen do druhého. „Vnější“ příkaz `with` by nám měl být povědomý. Otvírá textový soubor zakódovaný v `UTF-8` a pojmenovaný `out.log` pro zápis a přiřazuje objekt typu stream do proměnné pojmenované `a_file`. Ale je tu ještě jedna zvláštnost.

```
with RedirectStdoutTo(a_file):
```

Kdepak je část `as`? Příkaz `with` ji ve skutečnosti nevyžaduje. Podobně, jako když voláte funkci a ignorujete její návratovou hodnotu, můžete použít i příkaz `with`, který nepřiřazuje kontext příkazu `with` do nějaké proměnné. V tomto případě nás zajímají pouze vedlejší efekty kontextu `RedirectStdoutTo`.

A jaké jsou ty vedlejší efekty? Nahlédněme do třídy `RedirectStdoutTo`. Tato třída je uživatelsky definovaným správcem kontextu. Roli správce kontextu může hrát každá funkce, která definuje speciální metody `__enter__()` a `__exit__()`.

```
class RedirectStdoutTo:
    def __init__(self, out_new):          [1]
        self.out_new = out_new

    def __enter__(self):                  [2]
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):            [3]
        sys.stdout = self.out_old
```

[1] Metoda `__init__()` se volá bezprostředně po vytvoření instance. Přebírá jeden parametr — objekt typu stream, který chceme po dobu životnosti kontextu používat jako standardní výstup. Metoda uloží odkaz na objekt typu stream do instanční proměnné, aby jej mohly později používat ostatní metody.

[2] Metoda `__enter__()` patří mezi speciální metody třídy. Python ji volá v okamžiku vstupu do kontextu (tj. na začátku příkazu `with`). Metoda ukládá aktuální hodnotu `sys.stdout` do `self.out_old` a poté přesměruje standardní výstup přiřazením `self.out_new` do `sys.stdout`.

[3] Metoda `__exit__()` je další speciální metodou třídy. Python ji volá při opouštění kontextu (tj. na konci příkazu `with`). Metoda obnoví původní nasměrování standardního výstupu přiřazením uložené hodnoty `self.out_old` do `sys.stdout`.

Spojme to všechno dohromady:

```
print('A')                                                                    [1]
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):  [2]
    print('B')                                                                [3]
print('C')                                                                    [4]
```

[1] Výsledek se vytiskne v „interaktivním okně“ `IDE` (nebo v terminálovém okně, pokud skript spouštíme z příkazového řádku).

[2] Tento příkaz `with` přebírá *čárkou oddělený seznam kontextů*. Uvedený seznam se chová jako posloupnost vnořených bloků `with`. První kontext v seznamu je chápán jako „vnější“ blok, poslední jako „vnitřní“ blok. První kontext otvírá soubor, druhý kontext přesměrovává `sys.stdout` do objektu typu stream, který byl vytvořen v prvním kontextu.

[3] Funkce `print()` je provedena v kontextu vytvořeném příkazem `with`, a proto nebude tisknout na obrazovku. Místo toho provede zápis do souboru `out.log`.

[4] Blok kódu v příkazu `with` skončil. Python každému správci kontextu oznámil, že má udělat to, co se má udělat při opouštění kontextu. Správci kontextu jsou uloženi v zásobníku (LIFO). Druhý kontext při ukončování změnil obsah `sys.stdout` zpět na původní hodnotu a potom první kontext uzavřel soubor pojmenovaný `out.log`. A protože bylo přesměrování standardního výstupu obnoveno na původní hodnotu, bude funkce `print()` tisknout zase na obrazovku.

Přesměrování standardního chybového výstupu funguje naprosto stejně. Jen se místo `sys.stdout` použije `sys.stderr`.

11.7. Přečtěte si (vše anglicky)

- **Reading and writing files** v oficiální učebnici Python.org (http://docs.python.org/py3k/tutorial/inputoutput.html#reading-and-writing-files)
- **`io module`** — standardní dokumentace (http://docs.python.org/py3k/library/io.html)
- **Stream objects** — standardní dokumentace (http://docs.python.org/py3k/library/stdtypes.html)
- **Context manager types** — standardní dokumentace (http://docs.python.org/py3k/library/stdtypes.html)
- **`sys.stdout and sys.stderr`** — standardní dokumentace (http://docs.python.org/py3k/library/sys.html)
- **`FUSE` na Wikipedii** (anglicky; lze přepnout na odpovídající české heslo) (http://en.wikipedia.org/wiki/Filesystem_in_Userspace)

“In the archonship of Aristaechmus, Draco enacted his ordinances.”
(Za vlády Aristaechma uzákonil Drakon svá pravidla.)

— Aristoteles

12. XML

12. XML — 255
12.1. Ponořme se — **257**
12.2. Pětiminutový rychlokurz XML — **258**
12.3. Struktura Atom Feed — **261**
12.4. Analýza XML — **263**
12.4.1. Elementy jsou reprezentovány seznamy — **264**
12.4.2. Atributy jsou reprezentovány slovníky — **264**
12.5. Vyhledávání uzlů v XML dokumentu — **265**
12.6. lxml jde ještě dál — **268**
12.7. Generování XML — **270**
12.8. Analýza porušeného XML — **273**
12.9. Přečtěte si — **275**

12.1. Ponořme se

Téměř všechny kapitoly této knihy se točí kolem příkladů kódu. XML nesouvisí s kódem, ale s daty. Jedním z míst, kde se XML běžně používá, je „publikovaný obsah“ (syndication feeds), ve kterém se udržuje seznam posledních článků blogu, fóra nebo jiného, často aktualizovaného obsahu webového místa. Nejpopulárnější blogovací programy vytvářejí obsah (feed), a kdykoliv je publikován nový článek, diskusní vlákno nebo zpráva na blogu, tento obsah aktualizují. Blog můžeme sledovat tak, že se „přihlásíme k odběru“ jeho obsahu (feed). Více blogů můžeme sledovat tak, že použijeme k tomu určený „nástroj pro sdružování obsahu (feed aggregator)“, jako je například Google Reader.

V této kapitole budeme pracovat s následujícími XML daty. Jde o publikovaný obsah (feed) — konkrétně o Atom syndication feed.

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  <link rel='self' type='application/atom+xml' href='http://diveintomark.org/feed/'/>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Dive into history, 2009 edition</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'/>
    <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
    <updated>2009-03-27T21:56:07Z</updated>
    <published>2009-03-27T17:20:42Z</published>
    <category scheme='http://diveintomark.org' term='diveintopython'/>
    <category scheme='http://diveintomark.org' term='docbook'/>
    <category scheme='http://diveintomark.org' term='html'/>
  <summary type='html'>Putting an entire chapter on one page sounds
    bloated, but consider this &amp;mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds&amp;hellip;
    On dialup.</summary>
  </entry>
  <entry>
    <author>
      <name>Mark</name>
```

```
    <uri>http://diveintomark.org/</uri>
  </author>
  <title>Accessibility is a harsh mistress</title>
  <link rel='alternate' type='text/html'
    href='http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress'/>
  <id>tag:diveintomark.org,2009-03-21:/archives/20090321200928</id>
  <updated>2009-03-22T01:05:37Z</updated>
  <published>2009-03-21T20:09:28Z</published>
  <category scheme='http://diveintomark.org' term='accessibility'/>
  <summary type='html'>The accessibility orthodoxy does not permit people to
    question the value of features that are rarely useful and rarely used.</summary>
</entry>
<entry>
  <author>
    <name>Mark</name>
  </author>
  <title>A gentle introduction to video encoding, part 1: container formats</title>
  <link rel='alternate' type='text/html'
    href='http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats'/>
  <id>tag:diveintomark.org,2008-12-18:/archives/20081218155422</id>
  <updated>2009-01-11T19:39:22Z</updated>
  <published>2008-12-18T15:54:22Z</published>
  <category scheme='http://diveintomark.org' term='asf'/>
  <category scheme='http://diveintomark.org' term='avi'/>
  <category scheme='http://diveintomark.org' term='encoding'/>
  <category scheme='http://diveintomark.org' term='flv'/>
  <category scheme='http://diveintomark.org' term='GIVE'/>
  <category scheme='http://diveintomark.org' term='mp4'/>
  <category scheme='http://diveintomark.org' term='ogg'/>
  <category scheme='http://diveintomark.org' term='video'/>
  <summary type='html'>These notes will eventually become part of a
    tech talk on video encoding.</summary>
</entry>
</feed>
```

12.2. Pětiminutový rychlokurz XML

Pokud už o XML něco víte, můžete tuto podkapitolu přeskočit.

XML představuje zobecněný způsob popisu hierarchických strukturovaných dat. XML-*dokument* obsahuje jeden nebo více *elementů*, které jsou *ohraničeny počátečními a koncovými značkami* (tag). Tohle je kompletní (i když poněkud nudný) XML dokument:

```
<foo>                                [1]
</foo>                               [2]
```

[1] Toto je *počáteční značka* elementu `foo`.

[2] Toto je odpovídající *koncová značka* elementu `foo`. Každá počáteční značka musí být uzavřena (spárována s) odpovídající koncovou značkou stejně, jako musíme párovat závorky v matematice nebo v textu.

Elementy lze *zanořovat* do libovolné hloubky. O elementu `bar` uvnitř elementu `foo` se říká, že je *subelementem* nebo *potomkem* (child) elementu `foo`.

```
<foo>
  <bar></bar>
</foo>
```

Prvnímu elementu v každém `XML` dokumentu se říká *kořenový element* (root element). `XML` dokument může mít jen jeden kořenový element. Následující text **není** `XML` **dokumentem**, protože obsahuje dva kořenové elementy:

```
<foo></foo>
<bar></bar>
```

Elementy mohou nést *atributy*, což jsou dvojice jméno-hodnota. Atributy se uvádějí uvnitř počáteční značky elementu a oddělují se bílými znaky. Uvnitř jednoho elementu se *jména atributů* nesmějí opakovat. *Hodnoty atributů* musí být uzavřeny v uvozovkách nebo v apostrofech.

```
<foo lang='en'>                          [1]
  <bar id='papayawhip' lang="fr"></bar>  [2]
</foo>
```

[1] Element `foo` má jeden atribut pojmenovaný `lang`. Hodnotou jeho atributu `lang` je `en`.

[2] Element `bar` má dva atributy pojmenované `id` a `lang`. Jeho atribut `lang` má hodnotu `fr`. Nedochází vůbec k žádnému konfliktu s elementem `foo`. Každý element má svou vlastní sadu atributů.

Pokud je v jednom elementu uvedeno víc atributů, pak jejich pořadí není významné. Atributy elementu tvoří neuspořádanou množinu dvojic klíčů a hodnot — jako pythonovský slovník. Počet atributů, které můžeme u každého elementu definovat, není nijak omezen.

Elementy mohou *obsahovat text*.

```
<foo lang='en'>
  <bar lang='fr'>PapayaWhip</bar>
</foo>
```

Elementy, které neobsahují žádný text a nemají žádné potomky, jsou *prázdné*.

```
<foo></foo>
```

Prázdné elementy můžeme zapisovat zkráceně. Když do počáteční značky umístíme znak /, můžeme koncovou značku úplně vynechat. XML dokument z předchozího příkladu můžeme zkráceně zapsat takto:

```
<foo/>
```

Podobně jako můžeme pythonovské funkce deklarovat *v různých modulech*, XML elementy můžeme deklarovat v různých *prostorech jmen*. Prostory jmen se obvykle podobají zápisu URL. *Výchozí prostor jmen* definujeme pomocí deklarace xmlns. Deklarace prostoru jmen vypadá podobně jako zápis atributu, ale plní odlišný účel.

```
<feed xmlns='http://www.w3.org/2005/Atom'>                [1]
  <title>dive into mark</title>                           [2]
</feed>
```

[1] Element feed se nachází v prostoru jmen http://www.w3.org/2005/Atom.

[2] Element title se také nachází v prostoru jmen http://www.w3.org/2005/Atom. Deklarace prostoru jmen ovlivní element, ve kterém se deklarace nachází, a dále všechny jeho dětské elementy (potomky).

Při deklaraci prostoru jmen můžeme použít také zápis xmlns:prefix, čímž prostor jmen spřáhneme se zadaným *prefixem*. V takovém případě musí být každý element tohoto prostoru jmen explicitně deklarován se stejným prefixem.

```
<atom:feed xmlns:atom='http://www.w3.org/2005/Atom'>      [1]
  <atom:title>dive into mark</atom:title>                 [2]
</atom:feed>
```

[1] Element feed se nachází v prostoru jmen http://www.w3.org/2005/Atom.

[2] Element title se také nachází v prostoru jmen http://www.w3.org/2005/Atom.

Z pohledu syntaktického analyzátoru pro XML jsou přecházející dva XML dokumenty *identické*. Prostor jmen + jméno elementu = XML identita. Prefixy se používají pouze k odkazu na prostor jmen. To znamená, že konkrétní jméno prefixu (atom:) je nepodstatné. Prostory jmen pasují, jména elementů se shodují, atributy (nebo neuvedení atributů) sedí, textový obsah každého elementu se také shoduje. To znamená, že se jedná o stejné XML dokumenty.

Na závěr uveďme, že XML dokumenty mohou na prvním řádku, před kořenovým elementem, uvádět informaci o znakovém kódování. (Pokud vás zajímá, jak může dokument obsahovat informaci, která musí být známa předtím, než se dokument zpracovává, pak detaily řešení této Hlavy XXII hledejte v sekci F specifikace XML (anglicky).)

```
<?xml version='1.0' encoding='utf-8'?>
```

Tak a teď už o XML víte dost na to, abyste mohli být nebezpeční!

12.3. Struktura Atom Feed

Vezměme si nějaký weblog nebo v podstatě libovolný webový server s často aktualizovaným obsahem, jako je například CNN.com. Server má svůj nadpis („CNN.com“), podnadpis („Breaking News, U.S., World, Weather, Entertainment & Video News“), datum poslední aktualizace („updated 12:43 p.m. EDT, Sat May 16, 2009“) a seznam článků zveřejněných v různých časech. Každý článek má také nadpis, datum prvního zveřejnění (a možná také datum poslední aktualizace, pokud zveřejnili upřesnění nebo opravili překlep) a jedinečné URL.

The Atom syndication format je navržen tak, aby všechny tyto informace zachytil ve standardním tvaru. Můj weblog a CNN.com se sice velmi liší v návrhu, rozsahu a v návštěvnosti, ale oba mají stejnou základní strukturu. CNN.com má nadpis, můj blog má nadpis. CNN.com zveřejňuje články, já zveřejňuji články.

Na nejvyšší úrovni se nachází *kořenový element*, který používají všechny „Atom feed“ — element `feed` v prostoru jmen `http://www.w3.org/2005/Atom`.

```
<feed xmlns='http://www.w3.org/2005/Atom'        [1]
      xml:lang='en'>                             [2]
```

[1] `http://www.w3.org/2005/Atom` je prostor jmen pro Atom.

[2] Libovolný element může obsahovat atribut `xml:lang`, který deklaruje jazyk elementu a jeho potomků. V tomto případě je atribut `xml:lang` deklarován jen jednou, v kořenovém elementu. To znamená, že celý obsah (feed) je v angličtině.

Atom feed (chápejte tento název jako pojem) obsahuje pár informací i o dokumentu samotném (tedy o sobě). Jsou deklarovány jako potomci kořenového elementu `feed`.

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>                                          [1]
  <subtitle>currently between addictions</subtitle>                      [2]
  <id>tag:diveintomark.org,2001-07-29:/</id>                             [3]
  <updated>2009-03-27T21:56:07Z</updated>                                [4]
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>  [5]
```

[1] Nadpis obsahu je `dive into mark`.

[2] Podnadpis obsahu je `currently between addictions`.

[3] Každý obsah (feed) potřebuje globálně jednoznačný identifikátor. V dokumentu RFC 4151 najdete, jak se dá vytvořit.

[4] Tento obsah byl naposledy aktualizován 27. března 2009 v 21.56 GMT. Obvykle se shoduje s časem poslední modifikace nejnovějšího článku.

[5] Teď to začne být zajímavé. Tento element `link` nemá žádný textový obsah, ale má tři atributy: `rel`, `type` a `href`. Hodnota atributu `rel` říká, jakého druhu odkaz je. Hodnota `rel='alternate'` vyjadřuje, že jde o odkaz na alternativní reprezentaci tohoto obsahu (feed). Atribut `type='text/html'` říká, že jde o odkaz na HTML stránku. Cíl odkazu je uveden v atributu `href`.

Teď už víme, že jde o obsah (feed) pro místo zvané „dive into mark“, které se nachází na `http://diveintomark.org/` a bylo naposledy aktualizováno 27. března 2009.

> Ačkoliv v některých XML dokumentech může být pořadí elementů důležité, pro Atom feed to neplatí.

Po metadatech vázaných na celý dokument (feed) se nachází seznam nejnovějších článků. Článek vypadá takto:

```
<entry>
  <author>                                                                  [1]
    <name>Mark</name>
    <uri>http://diveintomark.org/</uri>
  </author>
  <title>Dive into history, 2009 edition</title>                            [2]
  <link rel='alternate' type='text/html'                                    [3]
    href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'/>
  <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>         [4]
  <updated>2009-03-27T21:56:07Z</updated>                                   [5]
  <published>2009-03-27T17:20:42Z</published>
  <category scheme='http://diveintomark.org' term='diveintopython'/>        [6]
  <category scheme='http://diveintomark.org' term='docbook'/>
  <category scheme='http://diveintomark.org' term='html'/>
  <summary type='html'>Putting an entire chapter on one page sounds         [7]
    bloated, but consider this &amp;mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds&amp;hellip;
    On dialup.</summary>
</entry>                                                                    [8]
```

[1] Element `author` říká, kdo článek napsal: nějaký maník jménem Mark, který se poflakuje někde na `http://diveintomark.org/`. (Je to stejná hodnota, jako alternativní odkaz v metadatech k feed, ale nemusí tomu tak být. Mnoho weblogů využívá více autorů najednou a každý z nich mívá jiný osobní webový server.)

[2] Element `title` nese název článku — „Dive into history, 2009 edition“.

[3] Element `link` obsahuje adresu `HTML` verze tohoto článku, podobně jako v případě alternativního odkazu na úrovni celého obsahu (feed).

[4] Položky (`entry`), stejně jako celý obsah (feed), potřebují jednoznačný identifikátor.

[5] Položky nesou dvě data: datum prvního zveřejnění (`published`) a datum poslední modifikace (`updated`).

[6] Položky mohou nést libovolný počet kategorií (`category`). Tento článek je zařazen pod `diveintopython`, `docbook` a `HTML`.

[7] Element `summary` nese stručné shrnutí obsahu článku. (Existuje i `element` content — tj. obsah —, který zde není použit. Je určen pro vložení celého textu článku.) Tento element `summary` nese atribut `type='html'`, který je specifický pro Atom. Říká, že uvedené shrnutí není prostý text, ale úryvek ve formátu `HTML`. Ta informace je důležitá, protože se v něm nacházejí věci specifické pro `HTML` (`—` a `…`), které se nemají zviditelňovat jako text, ale jako „–“ a „…“.

[8] A na závěr je tu koncová značka elementu `entry`, která signalizuje konec metadat pro tento článek.

12.4. Analýza XML

Python dovede analyzovat `XML` dokumenty několika způsoby. Najdeme zde tradiční syntaktické analyzátory (také parsery) dom a sax. My se ale zaměříme na jinou knihovnu zvanou ElementTree.

```
>>> import xml.etree.ElementTree as etree        [1]
>>> tree = etree.parse('examples/feed.xml')      [2]
>>> root = tree.getroot()                        [3]
>>> root                                         [4]
<Element {http://www.w3.org/2005/Atom}feed at cd1eb0>
```

[1] Knihovna ElementTree je součástí standardní pythonovské knihovny. Nachází se v `xml.etree.ElementTree`.

[2] Primárním vstupním bodem knihovny ElementTree je funkce `parse()`, která přebírá buď jméno souboru nebo souboru se podobající objekt. Funkce zpracuje celý dokument najednou. Pokud chceme šetřit pamětí, existují způsoby, jak můžeme `XML` dokument zpracovávat postupně.

[3] Funkce parse() vrací objekt, který reprezentuje celý dokument. Ale *není* to kořenový element. Pokud chceme získat odkaz na kořenový element, zavoláme metodu `getroot()`.

[4] Jak se dalo čekat, kořenovým elementem je element `feed`, který se nachází v prostoru jmen `http://www.w3.org/2005/Atom`. Řetězcová reprezentace tohoto objektu v nás posiluje důležitý pohled: `XML` element je kombinací svého prostoru jmen a jména své značky (která se též nazývá *lokální jméno*). Každý element tohoto dokumentu se nachází v prostoru jmen Atom, takže kořenový element je reprezentován jako `{http://www.w3.org/2005/Atom}feed`.

> ElementTree reprezentuje `XML` elementy jako `{prostor_jmen}lokální_jméno`. Tento formát uvidíme a budeme používat na mnoha místech aplikačního rozhraní ElementTree.

12.4.1. Elementy jsou reprezentovány seznamy

V aplikačním rozhraní ElementTree se elementy chovají jako seznamy. Položkami seznamu jsou elementy potomků (child).

```
# pokračování předchozího příkladu
>>> root.tag                              [1]
'{http://www.w3.org/2005/Atom}feed'
>>> len(root)                             [2]
8
>>> for child in root:                    [3]
...   print(child)                        [4]
...
<Element {http://www.w3.org/2005/Atom}title at e2b5d0>
<Element {http://www.w3.org/2005/Atom}subtitle at e2b4e0>
<Element {http://www.w3.org/2005/Atom}id at e2b6c0>
<Element {http://www.w3.org/2005/Atom}updated at e2b6f0>
<Element {http://www.w3.org/2005/Atom}link at e2b4b0>
<Element {http://www.w3.org/2005/Atom}entry at e2b720>
<Element {http://www.w3.org/2005/Atom}entry at e2b510>
<Element {http://www.w3.org/2005/Atom}entry at e2b750>
```

[1] Pokračujme v předchozím příkladu. Kořenový element je `{http://www.w3.org/2005/Atom}feed`.

[2] „Délkou“ kořenového elementu rozumíme počet dětských elementů (potomků, child).

[3] Objekt elementu můžeme použít jako iterátor, který zajistí průchod všemi svými dětskými elementy.

[4] Na výstupu vidíme, že obsahuje očekávaných 8 potomků: metadata patřící k feed (`title`, `subtitle`, `id`, `updated` a `link`) následovaná třemi elementy `entry`.

Asi už jste to odhadli, ale zdůrazněme to ještě explicitně: seznam dětských elementů zahrnuje pouze *přímé* potomky. Každý z elementů `entry` obsahuje své vlastní potomky, ale ti v tomto seznamu uvedeni nejsou. Jako dětské elementy jsou součástí seznamů elementů `entry`, ale nejsou zahrnuty mezi potomky elementu `feed`. Existují způsoby, jak můžeme elementy vyhledat nezávisle na tom, jak hluboko jsou zanořené. Na dva takové způsoby se v této kapitole podíváme později.

12.4.2. Atributy jsou reprezentovány slovníky

`XML` není jen kolekcí elementů. Každý element má svou vlastní sadu atributů. Jakmile máme odkaz na konkrétní element, můžeme jeho atributy snadno získat jako pythonovský slovník.

```
# pokračování předchozího příkladu
>>> root.attrib                                          [1]
{'{http://www.w3.org/XML/1998/namespace}lang': 'en'}
>>> root[4]                                              [2]
<Element {http://www.w3.org/2005/Atom}link at e181b0>
>>> root[4].attrib                                       [3]
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}
>>> root[3]                                              [4]
<Element {http://www.w3.org/2005/Atom}updated at e2b4e0>
>>> root[3].attrib                                       [5]
{}
```

[1] Vlastnost `attrib` je slovníkem atributů elementu. Původní značka vypadala takto: `<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>`. Prefix `xml:` se vztahuje k zabudovanému prostoru jmen, který můžeme používat v každém `XML` dokumentu, aniž bychom jej museli deklarovat.

[2] Pátým potomkem — `[4]` odpovídá indexování seznamu od nuly — je element `link`.

[3] Element link má tři atributy: `href`, `type` a `rel`.

[4] Čtvrtým potomkem — `[3]` odpovídá indexování seznamu od nuly — je element `updated`.

[5] Element `updated` nemá žádné atributy, takže jeho vlastnost `.attrib` je prostě prázdný slovník.

12.5. Vyhledávání uzlů v XML dokumentu

Zatím jsme s uvedeným `XML` dokumentem pracovali „shora dolů“. Začali jsme u kořenového elementu, zpřístupnili jsme si elementy jeho potomků a tak dále napříč dokumentem. Ale při mnoha použitích `XML` se požaduje nalezení určitého elementu. Etree to umí také.

```
>>> import xml.etree.ElementTree as etree
>>> tree = etree.parse('examples/feed.xml')
>>> root = tree.getroot()
>>> root.findall('{http://www.w3.org/2005/Atom}entry')          [1]
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> root.tag
'{http://www.w3.org/2005/Atom}feed'
>>> root.findall('{http://www.w3.org/2005/Atom}feed')           [2]
[]
>>> root.findall('{http://www.w3.org/2005/Atom}author')         [3]
[]
```

[1] Metoda `findall()` najde všechny dětské elementy, které odpovídají určitému dotazu. (O formátu dotazu si řekneme za minutku.)

[2] Každý element — včetně kořenového elementu, ale také dětských elementů — má metodu `findall()`. Ta mezi potomky najde všechny odpovídající elementy. Ale proč tu nejsou žádné výsledky? Ačkoliv to nemusí být úplně zřejmé, tento dotaz prohledává jen elementy potomků. A protože kořenový element `feed` nemá žádného potomka jménem `feed`, vrací dotaz prázdný seznam.

[3] Tento výsledek vás možná také překvapí. V tomto dokumentu se nachází element `author`. Ve skutečnosti jsou v něm tři (jeden v každém elementu `entry`). Ale elementy `author` nejsou *přímými potomky* kořenového elementu. Jsou to jeho „vnuci“ (doslova potomci potomků). Pokud hledáte elementy `author` na libovolné úrovni zanoření, je to možné provést, ale formát dotazu se mírně liší.

```
>>> tree.findall('{http://www.w3.org/2005/Atom}entry')           [1]
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> tree.findall('{http://www.w3.org/2005/Atom}author')          [2]
[]
```

[1] Z praktických důvodů má objekt `tree` (vracený funkcí `etree.parse()`) několik metod, které odpovídají metodám kořenového elementu. Výsledky jsou stejné, jako kdybychom zavolali metodu `tree.getroot().findall()`.

[2] Tento dotaz, možná trošku překvapivě, v dokumentu nenajde elementy `author`. Proč ne? Protože je to zkratka pro `tree.getroot().findall('{http://www.w3.org/2005/Atom}author')`, což znamená „najdi všechny elementy `author`, které jsou potomky kořenového elementu“. Elementy `author` nejsou potomky kořenového elementu. Jsou to potomci elementů `entry`. Takže uvedený dotaz nenajde žádnou shodu.

Existuje také metoda `find()`, která vrací první vyhovující element. Hodí se v situacích, kdy očekáváme pouze jeden výskyt, nebo když je výskytů víc, ale zajímá nás jen první.

```
>>> entries = tree.findall('{http://www.w3.org/2005/Atom}entry')                [1]
>>> len(entries)
3
>>> title_element = entries[0].find('{http://www.w3.org/2005/Atom}title')       [2]
>>> title_element.text
'Dive into history, 2009 edition'
>>> foo_element = entries[0].find('{http://www.w3.org/2005/Atom}foo')           [3]
>>> foo_element
>>> type(foo_element)
<class 'NoneType'>
```

[1] Tohle jsme viděli v předchozím příkladu. Naleznou se všechny elementy `atom:entry`.
[2] Metoda `find()` přebírá dotaz a vrací první vyhovující element.
[3] Uvnitř elementu nejsou žádné položky nazvané `foo`, takže se vrací `None`.

> S metodou `find()` je spojen „chyták“, který vás jednou dostane. Objekt elementu z ElementTree se v booleovském kontextu vyhodnocuje jako `False` v případě, kdy neobsahuje žádné potomky (tj. jestliže `len(element)` je rovno nule). To znamená, že zápis `if element.find('...')` netestuje, zda metoda `find()` nalezla vyhovující element. Testuje, zda vyhovující element má nějaké potomky! Pokud chceme testovat, zda metoda `find()` vrátila nějaký element, musíme použít zápis `if element.find('...') is not None`.

On ale existuje způsob, jak najít elementy veškerých *příbuzných potomků*, tj. dětí, vnuků a dalších elementů na libovolné úrovni zanoření.

```
>>> all_links = tree.findall('//{http://www.w3.org/2005/Atom}link')        [1]
>>> all_links
[<Element {http://www.w3.org/2005/Atom}link at e181b0>,
 <Element {http://www.w3.org/2005/Atom}link at e2b570>,
 <Element {http://www.w3.org/2005/Atom}link at e2b480>,
 <Element {http://www.w3.org/2005/Atom}link at e2b5a0>]
>>> all_links[0].attrib                                                    [2]
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[1].attrib                                                    [3]
{'href': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[2].attrib
{'href': 'http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[3].attrib
{'href': 'http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats',
 'type': 'text/html',
 'rel': 'alternate'}
```

[1] Tento dotaz — `//{http://www.w3.org/2005/Atom}link` — je těm z předchozích příkladů velmi podobný. Jedinou odlišností jsou dvě lomítka na začátku dotazu. Tato dvě lomítka znamenají: „Nedívej se jen na přímé potomky. Chci najít *jakékoliv* elementy, nezávisle na úrovni zanoření.“ Takže výsledkem je seznam čtyř elementů `link` a nejen jednoho.

[2] První výsledek je přímým potomkem kořenového elementu. Jak vidíme z jeho atributů, jde o alternativní odkaz z úrovně celého obsahu (feed). Odkazuje na HTML verzi webového místa, které zveřejňovaný obsah popisuje.

[3] Ostatní tři výsledky jsou alternativní odkazy z každého elementu `entry`. Každý element `entry` obsahuje jeden dětský element `link`. A protože je na začátku dotazu uvedena dvojice lomítek, najde dotaz všechny.

Celkově vzato je metoda `findall()` objektu třídy ElementTree velmi mocným nástrojem, ale dotazovací jazyk může přinést pár překvapení. Oficiálně se o něm píše jako o „omezené podpoře výrazů XPath". XPath je W3C standardem pro dotazování v XML dokumentech. Dotazovací jazyk implementovaný třídou ElementTree se XPath podobá do té míry, že se hodí pro základní vyhledávání. Ale pokud už znáte XPath, mohou vás rozdíly rozčilovat. Teď se podíváme na XML knihovnu třetí strany, která rozšiřuje aplikační rozhraní ElementTree o plnou podporu XPath.

12.6. lxml jde ještě dál

`lxml` je open source knihovna třetí strany, která je vybudována nad populárním parserem `libxml2`. Poskytuje aplikační rozhraní, které je 100% slučitelné s ElementTree a rozšiřuje ho o plnou podporu XPath 1.0 a o pár dalších vylepšení. K dispozici jsou instalátory pro Windows. Uživatelé Linuxu by měli zkusit nainstalovat předkompilovaný binární tvar z archivů prostřednictvím nástrojů příslušné distribuce, jako je třeba `yum` nebo `apt-get`. Pokud by to nešlo, museli byste `lxml` nainstalovat ručně.

```
>>> from lxml import etree                                          [1]
>>> tree = etree.parse('examples/feed.xml')                         [2]
>>> root = tree.getroot()                                           [3]
>>> root.findall('{http://www.w3.org/2005/Atom}entry')              [4]
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
```

[1] Jakmile `lxml` naimportujeme, máme k dispozici stejné aplikační rozhraní jako u zabudované knihovny ElementTree.

[2] Funkce `parse()` — stejná jako u ElementTree.

[3] Metoda `getroot()` — také stejná.

[4] Metoda `findall()` — naprosto stejná.

Pro velké `xml` dokumenty je `lxml` výrazně rychlejší než zabudovaná knihovna ElementTree. Pokud používáte pouze aplikační rozhraní ElementTree a chcete používat nejrychlejší dostupnou implementaci, můžete vyzkoušet naimportovat `lxml` se záchranou v podobě zabudované ElementTree.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

Ale `lxml` je víc než pouhá rychlejší podoba ElementTree. Její implementace metody `findall()` podporuje komplikovanější výrazy.

```
>>> import lxml.etree                                                          [1]
>>> tree = lxml.etree.parse('examples/feed.xml')
>>> tree.findall('//{http://www.w3.org/2005/Atom}*[@href]')                    [2]
[<Element {http://www.w3.org/2005/Atom}link at eeb8a0>,
 <Element {http://www.w3.org/2005/Atom}link at eeb990>,
 <Element {http://www.w3.org/2005/Atom}link at eeb960>,
 <Element {http://www.w3.org/2005/Atom}link at eeb9c0>]
>>> tree.findall("//{http://www.w3.org/2005/Atom}*[@href='http://diveintomark.org/']")  [3]
[<Element {http://www.w3.org/2005/Atom}link at eeb930>]
>>> NS = '{http://www.w3.org/2005/Atom}'
>>> tree.findall('//{NS}author[{NS}uri]'.format(NS=NS))                        [4]
[<Element {http://www.w3.org/2005/Atom}author at eeba80>,
 <Element {http://www.w3.org/2005/Atom}author at eebba0>]
```

[1] V tomto příkladu provedeme `import lxml.etree`. Chceme zde zdůraznit, že jde o vlastnosti specifické pro `lxml` (takže nenapíšeme, dejme tomu, `from lxml import etree`).

[2] Tento dotaz najde všechny elementy z prostoru jmen Atom, které mají atribut `href` — ať už se nacházejí v dokumentu kdekoliv. Dvě lomítka (`//`) na začátku dotazu znamenají „elementy nacházející se kdekoliv (ne jenom potomci nebo kořenový element)“. `{http://www.w3.org/2005/Atom}` znamená „jen elementy z prostoru jmen Atom“. `*` znamená „elementy s libovolným lokálním jménem“. A `[@href]` znamená, „které mají atribut `href`”.

[3] Tento dotaz najde všechny elementy z Atom, které mají `href` s hodnotou `http://diveintomark.org/`.

[4] S využitím jednoduchého formátovacího řetězce (protože jinak by se tyto složené dotazy staly neúnosně dlouhé) získáme dotaz, který v prostoru Atom hledá elementy `author`, které mají mezi svými potomky element `uri`. Vrátí se jen dva elementy `author` — jen z prvního a druhého elementu `entry`. Element author v posledním `entry` obsahuje jen `name` — `uri` mu chybí.

Ještě toho nemáte dost? Do `lxml` je zahrnuta i podpora pro libovolné výrazy XPath 1.0. Nebudu se do hloubky zabývat syntaxí XPath. To by samo o sobě vydalo na celou knihu! Ale ukážeme si, jakým způsobem je podpora XPath do `lxml` zahrnuta.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed.xml')
>>> NSMAP = {'atom': 'http://www.w3.org/2005/Atom'}                    [1]
>>> entries = tree.xpath("//atom:category[@term='accessibility']/..",  [2]
...     namespaces=NSMAP)
>>> entries                                                            [3]
[<Element {http://www.w3.org/2005/Atom}entry at e2b630>]
>>> entry = entries[0]
>>> entry.xpath('./atom:title/text()', namespaces=NSMAP)               [4]
['Accessibility is a harsh mistress']
```

[1] Abychom mohli provádět dotazy XPath nad elementy z nějakého prostoru jmen, musíme definovat zobrazení prefixu na prostor jmen. Je to prostě pythonovský slovník.

[2] Tady máme dotaz v XPath. Výraz v XPath hledá elementy `category` (z prostoru jmen Atom), které obsahují atribut `term` s hodnotou `accessibility`. To ale ještě není výsledkem dotazu. Podívejte se na úplný konec řetězce dotazu. Všimli jste si úseku /..? Ten znamená „a vrať k právě nalezenému elementu `category` jeho rodičovský element“. Takže tento jediný dotaz XPath najde všechny elementy potomky `<category term='accessibility'>`.

[3] Funkce `xpath()` vrací seznam objektů třídy ElementTree. V tomto dokumentu se nachází jediný záznam obsahující `category`, jehož term má hodnotu `accessibility`.

[4] XPath výraz nevrací vždycky seznam elementů. `DOM` (Document Object Model; objektový model dokumentu), který vznikl na základě zpracování (parsing) `XML` dokumentu, neobsahuje z technického hlediska elementy, ale *uzly*. Uzly mohou (podle typu) reprezentovat elementy, atributy nebo dokonce textový obsah. Výsledkem XPath dotazu je seznam uzlů. Tento dotaz vrací seznam textových uzlů: textový obsah (`text()`) elementu `title` (`atom:title`), který je potomkem aktuálního elementu (`./`).

12.7. Generování XML

Podpora `XML` v Pythonu není omezena na analýzu (parsing) existujících dokumentů. Můžeme také vytvářet `XML` dokumenty zcela od základů.

```
>>> import xml.etree.ElementTree as etree
>>> new_feed = etree.Element('{http://www.w3.org/2005/Atom}feed',      [1]
...     attrib={'{http://www.w3.org/XML/1998/namespace}lang': 'en'})   [2]
>>> print(etree.tostring(new_feed))                                    [3]
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

[1] Nový element vznikne vytvořením instance třídy `Element`. Jako první argument předáváme jméno elementu (prostor jmen + lokální jméno). Tímto příkazem se vytvoří element `feed` v prostoru jmen Atom. To bude kořenový element našeho nového dokumentu.

[2] Atributy k nově vytvořenému elementu přidáme předáním slovníku se jmény a hodnotami atributů argumentem `attrib`. Poznamenejme, že jména atributů musí být uvedena ve tvaru pro ElementTree — `{prostor jmen}lokální jméno`.

[3] Kterýkoliv element (a jeho potomky) můžeme kdykoliv převést na řetězec (serializovat) voláním funkce `tostring()` z ElementTree.

Jste výsledkem serializace překvapeni? Způsob, jakým ElementTree serializuje `XML` elementy s prostorem jmen, je sice z technického hlediska přesný, ale není optimální. Vzorový `XML` dokument ze začátku této kapitoly definoval *výchozí prostor jmen* (`xmlns='http://www.w3.org/2005/Atom'`). U dokumentů, kde se všechny elementy nacházejí ve stejném prostoru jmen — jako u Atom feeds — je definice výchozího prostoru jmen užitečná, protože ji uvedeme jen jednou a elementy pak můžeme deklarovat jen jejich lokálním jménem (`<feed>`, `<link>`, `<entry>`). Pokud nepotřebujeme deklarovat elementy z jiného prostoru jmen, nemusíme prefixy uvádět.

`XML` parser „nevidí“ mezi `XML` dokumentem s výchozím prostorem jmen a mezi `XML` dokumentem s prefixovaným prostorem jmen žádný rozdíl. Výsledný `DOM` s následující serializací:

```
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

je totožný s `DOM` s touto serializací:

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

Jediný praktický rozdíl spočívá v tom, že druhá serializace je o pár znaků kratší. Kdybychom chtěli celý vzorek našeho obsahu (feed) přepsat s prefixem `ns0`: v každé počáteční a koncové značce, přidalo by to 4 znaky na každou značku × 79 značek + 4 znaky pro vlastní deklaraci prostoru jmen, to je celkem 320 znaků. Za předpokladu, že používáme kódování `UTF-8`, to je 320 bajtů navíc. (Po zabalení pomocí `gzip` se rozdíl zmenší na 21 bajtů, ale 21 bajtů je pořád 21 bajtů.) Pro vás to možná nic neznamená, ale pro něco takového jako je Atom feed, který může být stahován několikatisíckrát, kdykoliv dojde ke změně, se může úspora pár bajtů na dotaz rychle nasčítat.

Zabudovaná knihovna ElementTree tak jemné ovládání serializace elementů z prostoru jmen nenabízí, ale `lxml` ano.

```
>>> import lxml.etree
>>> NSMAP = {None: 'http://www.w3.org/2005/Atom'}                          [1]
>>> new_feed = lxml.etree.Element('feed', nsmap=NSMAP)                     [2]
>>> print(lxml.etree.tounicode(new_feed))                                  [3]
<feed xmlns='http://www.w3.org/2005/Atom'/>
>>> new_feed.set('{http://www.w3.org/XML/1998/namespace}lang', 'en')       [4]
>>> print(lxml.etree.tounicode(new_feed))
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

[1] Začneme tím, že definujeme zobrazení prostorů jmen v podobě slovníku. Hodnotami slovníku jsou prostory jmen, klíči jsou požadované prefixy. Použitím `None` v roli klíče definujeme výchozí prostor jmen.

[2] Když teď při vytváření elementu předáme slovník argumentem `nsmap` (je specifický pro `lxml`), bude `lxml` respektovat prefixy prostorů jmen, které jsme definovali.

[3] Tato serializace podle očekávání definuje prostor jmen Atom jako výchozí prostor jmen a deklaruje element `feed` bez prefixu prostoru jmen.

[4] Jejda! Zapomněli jsme přidat atribut `xml:lang`. Libovolný atribut můžeme k libovolnému elementu přidat metodou `set()`. Přebírá dva argumenty: jméno atributu ve formátu pro ElementTree a hodnotu atributu. (Tato metoda není specifická pro `lxml`. Jedinou částí specifickou pro `lxml` byl v tomto příkladu argument `nsmap`, který v serializovaném výstupu ovládá prefixování prostorem jmen.)

Může se v `XML` dokumentech vyskytovat jen jeden element na dokument? Samozřejmě že ne. Snadno můžeme vytvořit i elementy potomků.

```
>>> title = lxml.etree.SubElement(new_feed, 'title',                    [1]
...     attrib={'type':'html'})                                         [2]
>>> print(lxml.etree.tounicode(new_feed))                               [3]
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type='html'/></feed>
>>> title.text = 'dive into &hellip;'                                   [4]
>>> print(lxml.etree.tounicode(new_feed))                               [5]
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type='html'>dive into
&amp;hellip;</title></feed>
>>> print(lxml.etree.tounicode(new_feed, pretty_print=True))            [6]
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
<title type='html'>dive into&amp;hellip;</title>
</feed>
```

[1] Při vytváření dětského elementu k existujícímu elementu vytváříme instanci třídy `SubElement`. Jedinými povinnými argumenty jsou zde rodičovský element (v našem případě `new_feed`) a jméno nového elementu. Protože má dětský element dědit mapování (zobrazení) prostoru jmen od svého rodiče, nemusíme zde prostoj jmen nebo prefix znovu deklarovat.

[2] Můžeme také předat slovník atributů. Klíče hrají roli jmen atributů, hodnoty jsou hodnotami atributů.

[3] Podle očekávání byl v prostoru jmen Atom vytvořen element `title` a byl vložen jako potomek do elementu `feed`. Protože element `title` neobsahoval žádný text a neměl své vlastní potomky, serializuje jej `lxml` jako prázdný element (zkrácený zápis s `/>` na konci).

[4] Pokud chceme elementu nastavit textový obsah, přiřadíme jej jednoduše do vlastnosti `.text`.

[5] Teď už se element `title` serializuje i se svým textovým obsahem. Každý text, který obsahuje znaky menší než nebo `ampersand`, musí být při serializaci převeden na speciální posloupnosti. `lxml` se o to postará automaticky.

[6] Při serializaci můžeme předepsat také „tisk v pěkném tvaru“. Za koncové značky a za počáteční značky elementů, které obsahují potomky, ale ne text, se vloží přechody na nový řádek. Vyjádřeno technickými pojmy, `lxml` přidá „nevýznamné bílé znaky“ za účelem zvýšení čitelnosti výstupu.

> Možná byste se chtěli mrknout také na `xmlwitch`, což je další knihovna třetí strany pro generování `XML`. Aby byl kód pro generování `XML` čitelnější, široce se v ní využívá příkazu `with`.

12.8. Analýza porušeného XML

Specifikace `XML` nařizuje, aby všechny `XML` parsery, které chtějí specifikaci vyhovět, používaly „drakonickou obsluhu chyb“. To znamená, že musí s výrazným efektem zastavit, jakmile v `XML` dokumentu narazí na jakýkoliv prohřešek proti korektní podobě. Prohřešky proti správné formě zahrnují nespárované počáteční a koncové značky, nedefinované entity (speciální posloupnosti pro znaky), nelegální Unicode znaky a řadu dalších esoterických pravidel. To je v příkrém kontrastu s jinými běžnými formáty, jako je například `HTML`. Váš prohlížeč nepřestane zobrazovat stránku, ve které zapomenete uvést uzavírací značku `HTML` nebo když zapomenete zapsat ampersand v atributu jako speciální sekvenci. (Běžným omylem je, že `HTML` nemá definováno ošetření chyb. Ošetřování chyb v `HTML` je ve skutečnosti definováno velmi dobře, ale je výrazně komplikovanější, než „zastav a začni hořet“ v okamžiku, kdy se narazí na první chybu.)

Někteří lidé věří (a já patřím mezi ně), že požadavek na drakonickou obsluhu chyb byl ze strany tvůrců `XML` nepřiměřený. Nechápejte mě špatně. Zjednodušení pravidel pro ošetření chyb má své kouzlo. Ale v praxi je koncepce „korektnosti formátu“ ošidnější, než to vypadá — zvlášť u `XML` (jako je Atom feeds), které jsou zveřejňovány na webu a zpřístupňovány protokolem `HTTP`. I přes vyzrálost formátu `XML`, který standardizoval drakonická pravidla pro ošetřování chyb v roce 1997, průzkumy stále ukazují, že významná část dokumentů Atom feeds nacházejících se na webu je zamořena chybami formátu.

Takže mám jak teoretické, tak praktické důvody ke zpracování (parse) `XML` dokumentů „za každou cenu“. To znamená, že *nechci* s kraválem zastavit při prvním prohřešku proti korektnosti formátu. Pokud zjistíte, že to cítíte podobně, může vám pomoci `LXML`.

Tady máme kousek porušeného `XML` dokumentu. Prohřešky proti korektnosti jsem zvýraznil.

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into &hellip;</title>
...
</feed>
```

Tak tohle je chyba, protože entita `…` není v `XML` definována. (Je definována v `HTML`.) Pokud se takto porušený obsah (feed) pokusíte zpracovat (parse), `lxml` se zakucká na nedefinované entitě.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed-broken.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lxml.etree.pyx", line 2693, in lxml.etree.parse (src/lxml/lxml.etree.c:52591)
  File "parser.pxi", line 1478, in lxml.etree._parseDocument (src/lxml/lxml.etree.c:75665)
  File "parser.pxi", line 1507, in lxml.etree._parseDocumentFromURL (src/lxml/lxml.etree.c:75993)
  File "parser.pxi", line 1407, in lxml.etree._parseDocFromFile (src/lxml/lxml.etree.c:75002)
  File "parser.pxi", line 965, in lxml.etree._BaseParser._parseDocFromFile (src/lxml/lxml.etree.c:72023)
  File "parser.pxi", line 539, in lxml.etree._ParserContext._handleParseResultDoc (src/lxml/lxml.
etree.c:67830)
  File "parser.pxi", line 625, in lxml.etree._handleParseResult (src/lxml/lxml.etree.c:68877)
  File "parser.pxi", line 565, in lxml.etree._raiseParseError (src/lxml/lxml.etree.c:68125)
lxml.etree.XMLSyntaxError: Entity 'hellip' not defined, line 3, column 28
```

Abychom byli schopni takto porušený XML dokument zpracovat (navzdory prohřešku proti korektnímu formátu), musíme vytvořit vlastní XML parser.

```
>>> parser = lxml.etree.XMLParser(recover=True)                                [1]
>>> tree = lxml.etree.parse('examples/feed-broken.xml', parser)               [2]
>>> parser.error_log                                                          [3]
examples/feed-broken.xml:3:28:FATAL:PARSER:ERR_UNDECLARED_ENTITY: Entity 'hellip' not defined
>>> tree.findall('{http://www.w3.org/2005/Atom}title')
[<Element {http://www.w3.org/2005/Atom}title at ead510>]
>>> title = tree.findall('{http://www.w3.org/2005/Atom}title')[0]
>>> title.text                                                                [4]
'dive into '
>>> print(lxml.etree.tounicode(tree.getroot()))                               [5]
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into </title>
.
. [zbývající serializace pro stručnost vynechány]
.
```

[1] Uživatelský parser (syntaktický analyzátor) vznikne vytvořením instance třídy `lxml.etree.XMLParser`. Lze jí předat celou řadu pojmenovaných argumentů. Nás momentálně zajímá argument `recover`. Pokud jej nastavíme na hodnotu `True`, XML parser udělá, co je v jeho silách, aby se z chyb proti korektnímu formátu „zotavil“.

[2] Náš XML dokument zpracujeme pomocí uživatelského parseru tak, že objekt `parser` předáme funkci `parse()` jako druhý argument. Všimněte si, že `lxml` kvůli nedefinované entitě `…` nevyvolal žádnou výjimku.

[3] Syntaktický analyzátor veškeré prohřešky proti korektnímu formátu zaznamenává. (Ve skutečnosti je zaznamenává nezávisle na tom, zda jsme mu nastavili zotavovací režim po chybě nebo ne.)

[4] Protože nevěděl, co má s nedefinovanou entitou `…` dělat, parser ji jednoduše vypustil. Takže textový obsah, který se nachází za elementem `title`, se změní na `'dive into '`.

[5] Jak vidíte ze serializované hodnoty, entita `…` se nikam nepřesunula. Byla jednoduše vypuštěna.

Pokud používáme syntaktické analyzátory XML se „zotavením“, pak je nutné znovu zopakovat, že neexistuje **žádná záruka vzájemné součinnosti**. Jiný parser se mohl rozhodnout, že jde o entitu `…` z HTML, a nahradí ji posloupností `…`. Je to „lepší“? Možná. Je to „správnější“? Ne. Oba případy jsou stejně nesprávné. Správné chování (podle specifikace XML) spočívá v tom, že parser „zastaví a začne hořet“. Pokud jste se rozhodli, že to neuděláte, je to vaše věc.

12.9. Přečtěte si

- **XML na Wikipedia.org**
  (http://en.wikipedia.org/wiki/XML)
  (česky zde http://cs.wikipedia.org/wiki/Extensible_Markup_Language)
- **The ElementTree XML API**
  (http://docs.python.org/py3k/library/xml.etree.elementtree.html)
- **Elements and Element Trees**
  (http://effbot.org/zone/element.htm)
- **XPath Support in ElementTree**
  (http://effbot.org/zone/element-xpath.htm)
- **The ElementTree iterparse Function**
  (http://effbot.org/zone/element-iterparse.htm)
- **lxml**
  (http://codespeak.net/lxml/)
- **Parsing XML and HTML with lxml**
  (http://codespeak.net/lxml/1.3/parsing.html)
- **XPath and XSLT with lxml**
  (http://codespeak.net/lxml/1.3/xpathxslt.html)
- **xmlwitch**
  (https://github.com/galvez/xmlwitch)

> **“Every Saturday since we’ve lived in this apartment, I have awakened at 6:15, poured myself a bowl of cereal, added a quarter-cup of 2% milk, sat on this end of this couch, turned on BBC America, and watched Doctor Who.”**
> (Každou sobotu, od té doby co žiji v tomto bytě, jsem vstal v 6.15, nasypal do sebe misku cereálií, přidal jsem hrnek 2% mléka, sedl jsem si na **tento** konec **této** pohovky, zapnul jsem BBC America a díval jsem se na Doctor Who.)
>
> — Sheldon, *The Big Bang Theory*

13. Serializace pythonovských objektů

13. Serializace pythonovských objektů — 277
13.1. Ponořme se — **279**
13.1.1. Stručná poznámka k příkladům v této kapitole — **279**
13.2. Uložení dat do „pickle souboru“ — **280**
13.3. Načítání dat z „pickle souboru“ — **281**
13.4. „Piklení“ bez souboru — **283**
13.5. Bajty a řetězce znovu zvedají své ošklivé hlavy — **284**
13.6. Ladění „pickle souborů“ — **284**
13.7. Serializace pythonovských objektů pro čtení z jiných jazyků — **286**
13.8. Uložení dat do JSON souboru — **287**
13.9. Zobrazení pythonovských datových typů do JSON — **289**
13.10. Serializace datových typů, které JSON nepodporuje — **289**
13.11. Načítání dat z JSON souboru — **293**
13.12. Přečtěte si — **295**

13.1. Ponořme se

Myšlenka serializace vypadá na první pohled jednoduše. Máme datovou strukturu v paměti, kterou chceme uložit, znovu použít nebo zaslat někomu jinému. Jak bychom to udělali? Záleží to na tom, jak ji chceme uložit, jak ji chceme znovu použít a komu ji chceme poslat. Mnoho her umožňuje, abyste si při ukončení uložili stav a při příštím spuštění pokračovali od tohoto místa dál. (Ve skutečnosti to umožňuje i mnoho aplikací, které nemají s hrami nic společného.) V takovém případě musí být datová struktura, která zachycuje „váš dosavadní pokrok“, při ukončení uložena na disk a při opětném spuštění z disku načtena. Data jsou určena jen pro použití se stejným programem, který je vytvořil. Nikdy se neposílají po síti a nikdy je nečte nic jiného než program, který je vytvořil. To znamená, že záležitost součinnosti se omezuje pouze na to, aby byla následující verze programu schopna načíst data zapsaná předchozími verzemi.

Pro tyto případy se ideálně hodí modul `pickle`. Je součástí pythonovské standardní knihovny, takže je kdykoliv k dispozici. Je rychlý. Jeho větší část je napsána v jazyce C, stejně jako vlastní interpret Pythonu. Dokáže uložit libovolně složité pythonovské datové struktury.

Co vlastně modul `pickle` dokáže uložit?

- Všechny Pythonem podporované přirozené datové typy: boolean, celá i reálná čísla, komplexní čísla, řetězce, objekty typu `bytes`, pole bajtů a `None`.
- Seznamy, n-tice, slovníky a množiny, které obsahují libovolnou kombinaci přirozených datových typů.
- Seznamy, n-tice, slovníky a množiny, které obsahují libovolnou kombinaci seznamů, n-tic, slovníků a množin, které obsahují libovolnou kombinaci přirozených datových typů (a tak dále až do maximální hloubky zanoření, kterou Python podporuje).
- Funkce, třídy a instance tříd (s upozorněním na určitá nebezpečí).
- A pokud se vám to zdá málo, modul `pickle` je navíc rozšiřitelný. Pokud vás možnost rozšiřitelnosti zajímá, podívejte se na odkazy v podkapitole Přečtěte si na konci kapitoly.

13.1.1. Stručná poznámka k příkladům v této kapitole

Tato kapitola vypráví příběh s dvěma pythonovskými shelly. Všechny příklady v kapitole jsou částí jedné linie příběhu. Během předvádění modulů `pickle` a `json` budeme přecházet z jednoho pythonovského shellu do druhého.

Abychom oba od sebe poznali, otevřete jeden pythonovský shell a definujte následující proměnnou:

```
>>> shell = 1
```

Okno nechejte otevřené. Teď otevřete druhý pythonovský shell a definujte proměnnou:

```
>>> shell = 2
```

Během kapitoly budeme používat proměnnou shell k indikaci toho, který pythonovský shell se u každého příkladu používá.

13.2. Uložení dat do „pickle souboru“

Modul `pickle` pracuje s datovými strukturami. Jednu takovou si připravíme.

```
>>> shell                                                                    [1]
1
>>> entry = {}                                                              [2]
>>> entry['title'] = 'Dive into history, 2009 edition'
>>> entry['article_link'] = 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'
>>> entry['comments_link'] = None
>>> entry['internal_id'] = b'\xDE\xD5\xB4\xF8'
>>> entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> entry['published'] = True
>>> import time
>>> entry['published_date'] = time.strptime('Fri Mar 27 22:20:42 2009')      [3]
>>> entry['published_date']
time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4,
tm_yday=86, tm_isdst=-1)
```

[1] Budeme pracovat v pythonovském shellu č. 1.

[2] Základní myšlenka spočívá ve vytvoření pythonovského slovníku, který reprezentuje něco užitečného, jako například záznam v Atom feed. Ale současně by měl obsahovat několik různých typů dat, abychom mohli modul `pickle` předvést. Nestudujte uvedené hodnoty zbytečně podrobně.

[3] Modul time definuje datovou strukturu (`struct_time`), která se používá k reprezentaci času (s přesností na milisekundy), a funkce, které s touto strukturou manipulují. Funkce `strptime()` přebírá formátovaný řetězec a převádí jej do podoby `struct_time`. Tento řetězec je ve výchozím tvaru, ale můžete jej ovlivnit formátovacími značkami. Podrobnosti hledejte v dokumentaci k modulu `time`.

Takže tu máme krásně vypadající pythonovský slovník. Uložme jej do souboru.

```
>>> shell                                          [1]
1
>>> import pickle
>>> with open('entry.pickle', 'wb') as f:          [2]
...     pickle.dump(entry, f)                      [3]
...
```

[1] Pořád se nacházíme v pythonovském shellu č. 1.

[2] K otevření souboru použijeme funkci `open()`. Režim souboru nastavíme na `'wb'`, abychom jej otevřeli pro zápis v binárním režimu. Zabalíme jej do příkazu `with`, abychom zajistili, že se po dokončení prací sám zavře.

[3] Funkce `dump()` z modulu pickle přebírá pythonovskou serializovatelnou datovou strukturu, serializuje ji do binárního podoby (je specifická pro Python a používá poslední verzi protokolu pro `pickle`) a uloží ji do otevřeného souboru.

Poslední věta je velmi důležitá.

- Modul `pickle` přebírá pythonovskou datovou strukturu a uloží ji do souboru.
- Aby to mohl udělat, *serializuje* datovou strukturu s využitím datového formátu zvaného „pickle protokol“. (Poznámka překladatele: Miluju anglicky mluvící tvůrce, kteří dávají konstrukcím a mechanismům „roztomilá“ jména. Pravděpodobně základním významem anglického pickle je „nálev“ a má také řadu dalších významů. Jenže zkuste to napasovat na český text věnovaný programovacímu jazyku. Jediné, co mi spolehlivě přichází na mysl, jsou úryvky písničky... „Kujme pikle, pikle kujme, spekulujme, intrikujme, lepšího nic není nad pořádný piklení.“ Kdo neví, gůůůglí.)
- Pickle protokol je specifický pro Python. Žádná záruka mezijazykové kompatibility neexistuje. Pravděpodobně není možné, abyste vzali soubor `entry.pickle`, který jsme zrovna vytvořili, a udělali s ním něco rozumného v Perlu, v `PHP`, v Javě nebo v nějakém jiném jazyce.
- Modul `pickle` nedokáže serializovat každou pythonovskou datovou strukturu. `Pickle` protokol se několikrát změnil s tím, jak byly do jazyka Python přidávány nové datové typy. Ale některá omezení přetrvávají.
- Výsledkem těchto změn je i to, že neexistuje žádná záruka kompatibility dokonce ani mezi různými verzemi Pythonu. Novější verze Pythonu podporují starší serializační formáty, ale starší verze Pythonu nepodporují nové formáty (protože nepodporují novější datové typy).
- Pokud neurčíte jinak, budou funkce z modulu `pickle` používat poslední verze pickle protokolu. Tím je zajištěna maximální pružnost z hlediska typů serializovatelných dat, ale také to znamená, že výsledný soubor nebude čitelný staršími verzemi Pythonu, které poslední verzi pickle protokolu nepodporují.
- Poslední verze pickle protokolu používá binární formát. Ujistěte se, že soubory pro „piklení“ otvíráte v binárním režimu. V opačném případě dojde během zápisu k porušení dat.

13.3. Načítání dat z „pickle souboru“

Teď se přepneme do druhého pythonovského shellu — tj. do toho, ve kterém jsme nevytvářeli slovník `entry`.

```
>>> shell                                          [1]
2
>>> entry                                          [2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import pickle
>>> with open('entry.pickle', 'rb') as f:         [3]
...     entry = pickle.load(f)                     [4]
...
>>> entry                                          [5]
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link':
 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22,
tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
```

[1] Tohle je pythonovský shell č. 2.

[2] Není zde definována žádná proměnná `entry`. Proměnnou `entry` jsme definovali v pythonovském shellu č. 1, ale ten se nachází v úplně jiném prostředí a udržuje svůj vlastní stav.

[3] Otevřeme soubor `entry.pickle`, který jsme vytvořili v pythonovském shellu č. 1. Modul `pickle` používá binární datový formát, takže byste jej měli vždy otvírat v binárním režimu.

[4] Funkce `pickle.load()` přebírá objekt typu stream, čte z něj serializovaná data, vytváří nový pythonovský objekt, rekonstruuje v něm serializovaná data a nový pythonovský objekt vrací.

[5] Nyní proměnná `entry` obsahuje slovník s důvěrně známými klíči a hodnotami.

Kroky `pickle.dump() / pickle.load()` vedou k vytvoření nové datové struktury, která se shoduje s původní datovou strukturou.

```
>>> shell                                          [1]
1
>>> with open('entry.pickle', 'rb') as f:         [2]
...     entry2 = pickle.load(f)                    [3]
...
>>> entry2 == entry                                [4]
True
>>> entry2 is entry                                [5]
False
>>> entry2['tags']                                 [6]
```

```
('diveintopython', 'docbook', 'html')
>>> entry2['internal_id']
b'\xDE\xD5\xB4\xF8'
```

[1] Přepneme se zpět do pythonovského shellu č. 1.
[2] Otevřeme soubor `entry.pickle`.
[3] Načteme serializovaná data do nové proměnné `entry2`.
[4] Python potvrzuje, že se slovníky `entry` a `entry2` shodují. V tomto shellu jsme strukturu `entry` vybudovali od základů. Začali jsme prázdným slovníkem a ručně jsme jednotlivým klíčům přiřadili určité hodnoty. Slovník jsme serializovali a uložili do souboru `entry.pickle`. Teď jsme serializovaná data z uvedeného souboru načetli a vytvořili jsme perfektní repliku původní datové struktury.
[5] Shodnost ale nezaměňujme za totožnost. Řekl jsem, že jsme vytvořili *perfektní repliku* původní datové struktury, což je pravda. Ale pořád je to jen kopie.
[6] Z důvodů, které budou objasněny v této kapitole později, chci upozornit na to, že klíči `'tags'` byla přiřazena hodnota v podobě n-tice a klíči `'internal_id'` byl přiřazen objekt typu `bytes`.

13.4. „Piklení“ bez souboru

Serializaci pythonovských objektů přímo do souboru na disk jsme si ukázali na příkladech v předchozí podkapitole. Ale co když soubor nechceme nebo nepotřebujeme? Serializaci můžeme provést také do objektu typu `bytes`, který se nachází v paměti.

```
>>> shell
1
>>> b = pickle.dumps(entry)              [1]
>>> type(b)                              [2]
<class 'bytes'>
>>> entry3 = pickle.loads(b)             [3]
>>> entry3 == entry                      [4]
True
```

[1] Funkce `pickle.dumps()` (všimněte si `'s'` na konci jména funkce) provádí stejnou serializaci jako funkce `pickle.dump()`. Ale nepřevezme objekt typu stream a serializovaná data nezapíše do souboru na disk. Místo toho serializovaná data jednoduše vrátí.
[2] A protože `pickle` protokol používá binární datový formát, vrátí funkce `pickle.dumps()` objekt typu `bytes`.
[3] Funkce `pickle.loads()` (opět si všimněte `'s'` na konci jména funkce) provádí stejnou deserializaci jako funkce `pickle.load()`. Místo čtení serializovaných dat ze souboru (přes objekt typu stream) přebírá objekt typu `bytes`, který serializovaná data obsahuje — takový, jaký vrátila funkce `pickle.dumps()`.
[4] Konečný výsledek je stejný: perfektní replika původního slovníku.

13.5. Bajty a řetězce znovu zvedají své ošklivé hlavy
13.6. Ladění „pickle souborů“

13.5. Bajty a řetězce znovu zvedají své ošklivé hlavy

Pickle protokol se používá už celou řadu let a vyspíval spolu s dospíváním Pythonu. V současnosti existují čtyři různé verze pickle protokolu.

- Python 1.x používal dva pickle protokoly: textový formát („verze 0“) a binární formát („verze 1“).
- Python 2.3 zavedl nový pickle protokol („verze 2“), který se vyrovnával s novou funkčností v pythonovských objektech tříd. Jeho formát je binární.
- Python 3.0 zavedl další pickle protokol („verze 3“) s explicitní podporou pro objekty typu `bytes` a pro pole bajtů. Jeho formát je binární.

Pozor, rozdíl mezi bajty a řetězci zase vystrkuje svou ošklivou hlavu. (Pokud jste dávali pozor, nejste překvapeni.) V praxi to znamená, že zatímco Python 3 umí číst data serializovaná protokolem verze 2, Python 2 neumí číst data „zapiklená“ protokolem verze 3.

13.6. Ladění „pickle souborů“

Jak vlastně pickle protokol vypadá? Vyskočme na chvíli z pythonovského shellu a podívejme se na soubor `entry.pickle`, který jsme vytvořili.

```
you@localhost:~/diveintopython3/examples$ ls -l entry.pickle
-rw-r--r-- 1 you  you  358 Aug  3 13:34 entry.pickle
you@localhost:~/diveintopython3/examples$ cat entry.pickle
comments_linkqNXtagsqXdiveintopythonqXdocbookqXhtmlq?qX publishedq?
XlinkXJhttp://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition
q   Xpublished_dateq
ctime
struct_time
?qRqXtitleqXDive into history, 2009 editionqu.
```

No, moc nám to tedy nepomohlo. Vidíme řetězce, ale ostatní datové typy končí jako netisknutelné (nebo přinejmenším nečitelné) znaky. Pole zjevně nejsou oddělena mezerami nebo tabulátory. Není to zrovna formát, který bychom chtěli analyzovat sami.

```
>>> shell
1
>>> import pickletools
>>> with open('entry.pickle', 'rb') as f:
...     pickletools.dis(f)
    0: \x80 PROTO      3
    2: }    EMPTY_DICT
    3: q    BINPUT     0
```

```
  5: (    MARK
  6: X        BINUNICODE 'published_date'
 25: q        BINPUT     1
 27: c        GLOBAL     'time struct_time'
 45: q        BINPUT     2
 47: (        MARK
 48: M            BININT2    2009
 51: K            BININT1    3
 53: K            BININT1    27
 55: K            BININT1    22
 57: K            BININT1    20
 59: K            BININT1    42
 61: K            BININT1    4
 63: K            BININT1    86
 65: J            BININT     -1
 70: t            TUPLE      (MARK at 47)
 71: q        BINPUT     3
 73: }        EMPTY_DICT
 74: q        BINPUT     4
 76: \x86     TUPLE2
 77: q        BINPUT     5
 79: R        REDUCE
 80: q        BINPUT     6
 82: X        BINUNICODE 'comments_link'
100: q        BINPUT     7
102: N        NONE
103: X        BINUNICODE 'internal_id'
119: q        BINPUT     8
121: C        SHORT_BINBYTES 'ÞÕ´ø'
127: q        BINPUT     9
129: X        BINUNICODE 'tags'
138: q        BINPUT     10
140: X        BINUNICODE 'diveintopython'
159: q        BINPUT     11
161: X        BINUNICODE 'docbook'
173: q        BINPUT     12
175: X        BINUNICODE 'html'
184: q        BINPUT     13
186: \x87     TUPLE3
187: q        BINPUT     14
189: X        BINUNICODE 'title'
199: q        BINPUT     15
201: X        BINUNICODE 'Dive into history, 2009 edition'
```

```
  237: q        BINPUT     16
  239: X        BINUNICODE 'article_link'
  256: q        BINPUT     17
  258: X        BINUNICODE 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'
  337: q        BINPUT     18
  339: X        BINUNICODE 'published'
  353: q        BINPUT     19
  355: \x88     NEWTRUE
  356: u        SETITEMS   (MARK at 5)
  357: .    STOP
highest protocol among opcodes = 3
```

Nejzajímavější informaci v tomto reverzním překladu najdeme na posledním řádku. Obsahuje totiž verzi pickle protokolu, kterým byl tento soubor vytvořen. Pickle protokol neobsahuje žádnou explicitní značku, která by určovala verzi. Abychom verzi protokolu určili, musíme prohlížet značky („operační kódy“) uvnitř serializovaných dat a řídit se podle toho, který operační kód byl zaveden jakou verzí pickle protokolu. Přesně to dělá funkce `pickle.dis()`. Výsledek vytiskne na posledním řádku reverzního překladu. Tady máme funkci, která vrátí číslo verze, aniž by něco tiskla:

```
import pickletools

def protocol_version(file_object):
    maxproto = -1
    for opcode, arg, pos in pickletools.genops(file_object):
        maxproto = max(maxproto, opcode.proto)
    return maxproto
```

A tady ji vidíme v akci:

```
>>> import pickleversion
>>> with open('entry.pickle', 'rb') as f:
...     v = pickleversion.protocol_version(f)
>>> v
3
```

13.7. Serializace pythonovských objektů pro čtení z jiných jazyků

Datový formát používaný modulem `pickle` je specifický pro Python. Nijak se nepokouší o kompatibilitu s jinými programovacími jazyky. Pokud je vaším cílem mezijazyková kompatibilita, pak se musíte poohlédnout po jiných serializačních formátech. Jedním z nich je `JSON`. Zkratka „`JSON`“ znamená „JavaScript Object Notation“, ale nenechte se tím jménem zmást. `JSON` je explicitně navržen pro použití napříč různými programovacími jazyky.

V Pythonu 3 je modul json součástí standardní knihovny. Modul json má (stejně jako modul pickle) funkce pro serializaci datových struktur, pro ukládání serializovaných dat na disk, pro načítání serializovaných dat z disku a pro deserializaci dat zpět do podoby nového pythonovského objektu. Ale najdeme zde také důležité odlišnosti. Ze všeho nejdřív uveďme, že datový formát JSON je textový a ne binární. Formát json a způsob kódování různých typů dat je definován v RFC 4627. Například booleovská hodnota je uložena buď jako pětiznakový řetězec 'false' nebo jako čtyřznakový řetězec 'true'. Všechny hodnoty používané v json jsou citlivé na velikost písmen.

Za druhé tu máme — jako u všech textových formátů — problém s bílými znaky (whitespace). JSON dovoluje, aby se mezi hodnotami vyskytovalo libovolné množství bílých znaků (mezery, tabulátory, návrat vozíku CR, přechod na nový řádek LF). Tyto bílé znaky jsou nevýznamné. To znamená, že kodéry JSON mohou přidat bílé znaky dle vlastního uvážení. Po dekodérech JSON se požaduje, aby bílé znaky mezi hodnotami ignorovaly. To umožňuje, aby byla json data „pěkně naformátována“ (pretty-print). Hodnoty mohou být pěkně vnořeny do jiných hodnot při použití různých úrovní odsazení, takže data budou dobře čitelná v textovém editoru nebo ve standardním prohlížeči. V pythonovském modulu json najdeme volbu, která při procesu kódování zajistí „pěkné formátování“.

Za třetí tu máme přetrvávající problém s kódováním znaků. json kóduje hodnoty do podoby prostého textu, ale my už víme, že nic jako „prostý text“ neexistuje. json musí být uložen v kódování Unicode (v UTF-32, v UTF-16 nebo ve výchozím UTF-8). Sekce 3 dokumentu RFC 4627 definuje, jak máme říct, které kódování je použito.

13.8. Uložení dat do JSON souboru

JSON se nápadně podobá datovým strukturám, které byste mohli ručně definovat v JavaScriptu. Není to žádná náhoda. Ve skutečnosti můžete pro „dekódování“ dat serializovaných do JSON použít javascriptovou funkci eval(). (Platí zde obvyklá výstraha o nedůvěryhodných zdrojích, ale věc se má tak, že json opravdu je platný JavaScript.) V tomto smyslu už se vám JSON může zdát důvěrně známý.

```
>>> shell
1
>>> basic_entry = {}                                          [1]
>>> basic_entry['id'] = 256
>>> basic_entry['title'] = 'Dive into history, 2009 edition'
>>> basic_entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> basic_entry['published'] = True
>>> basic_entry['comments_link'] = None
>>> import json
>>> with open('basic.json', mode='w', encoding='utf-8') as f:    [2]
...     json.dump(basic_entry, f)                              [3]
```

[1] Místo znovupoužití existující datové struktury entry si teď vytvoříme novou datovou strukturu.

Později si v této kapitole ukážeme, co se stane, když se do JSON pokusíme zakódovat složitější datovou strukturu.

[2] JSON je textový formát, což znamená, že soubor musíme otevřít v textovém režimu a musíme určit znakové kódování. Nikdy neuděláte chybu, když použijete UTF-8.

[3] Modul JSON (stejně jako modul pickle) definuje funkci dump(), která přebírá pythonovskou datovou strukturu a objekt typu stream připravený pro zápis. Funkce dump() serializuje pythonovskou datovou strukturu a zapíše ji do objektu typu stream. Vložením volání do příkazu with zajistíme, že po dokončení operace bude soubor korektně uzavřen.

Takže jak vlastně výsledek serializace do json vypadá?

```
you@localhost:~/diveintopython3/examples$ cat basic.json
{"published": true, "tags": ["diveintopython", "docbook", "html"], "comments_link": null,
"id": 256, "title": "Dive into history, 2009 edition"}
```

Tak tohle je určitě mnohem čitelnější než „zapiklený“ soubor. Navíc JSON může mezi hodnotami obsahovat libovolné bílé znaky a modul json nabízí snadný způsob, jak toho využít. Díky tomu můžeme vytvořit ještě mnohem čitelnější JSON soubory.

```
>>> shell
1
>>> with open('basic-pretty.json', mode='w', encoding='utf-8') as f:
...     json.dump(basic_entry, f, indent=2)                              [1]
```

[1] Pokud funkci json.dump() předáme parametr indent (tj. odsazení), může být výsledný JSON soubor mnohem čitelnější — za cenu zvětšení velikosti souboru. Parametr indent je celé číslo. 0 znamená „umísti každou hodnotu na zvláštní řádek“. Číslo větší než 0 znamená „umísti každou hodnotu na zvláštní řádek a použij tento počet mezer pro odsazování zanořených datových struktur“.

A takhle vypadá výsledek:

```
you@localhost:~/diveintopython3/examples$ cat basic-pretty.json
{
  "published": true,
  "tags": [
    "diveintopython",
    "docbook",
    "html"
  ],
  "comments_link": null,
  "id": 256,
  "title": "Dive into history, 2009 edition"
}
```

13.9. Zobrazení pythonovských datových typů do JSON
13.10. Serializace datových typů, které JSON nepodporuje

13.9. Zobrazení pythonovských datových typů do JSON

Protože json není určen pro Python, najdeme při zobrazování pythonovských datových typů určité nesrovnalosti. Některé z nich jsou jen rozdíly v názvech, ale dva důležité pythonovské datové typy v něm úplně chybí. Schválně, jestli si jich všimnete:

Poznámky	JSON	Python 3
	objekt	slovník
	pole	seznam
	řetězec	řetězec
	integer	integer
	reálné číslo	float
*	`true`	`True`
*	`false`	`False`
*	`null`	`None`

* Všechny hodnoty používané v JSON jsou citlivé na velikost písmen.

Všimli jste si, co chybí? N-tice a bajty! JSON definuje typ pole, které modul JSON zobrazuje na pythonovský seznam, ale nedefinuje oddělený typ pro „zmrazená pole“ (n-tice). A ačkoliv JSON docela pěkně podporuje řetězce, nepodporuje objekty typu bytes nebo pole bajtů.

13.10. Serializace datových typů, které JSON nepodporuje

I když JSON nemá žádnou zabudovanou podporu pro bajty, neznamená to, že bychom objekty typu bytes nemohli serializovat. Modul json poskytuje rozšiřující rozhraní (extensibility hooks) pro kódování a dekódování neznámých datových typů. (Slovem „neznámý“ rozumějme „nedefinovaný v JSON“. Modul json zjevně pole bajtů zná, ale je svázán omezeními specifikace JSON.) Pokud chceme zakódovat bajty nebo jiné datové typy, které JSON v základu nepodporuje, musíme pro ně dodat uživatelské kodéry a dekodéry.

```
>>> shell
1
>>> entry                                                    [1]
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
```

```
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22,
tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
>>> import json
>>> with open('entry.json', 'w', encoding='utf-8') as f:                [2]
...     json.dump(entry, f)                                             [3]
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "C:\Python31\lib\json\encoder.py", line 170, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: b'\xDE\xD5\xB4\xF8' is not JSON serializable
```

[1] Nastal čas k tomu, abychom se znovu podívali na datovou strukturu `entry`. Obsahuje následující: booleovskou hodnotu, hodnotu `None`, řetězec, n-tici řetězců, objekt typu `bytes` a strukturu `time`.

[2] Já vím. Říkal jsem to už dříve, ale stojí to za zopakování: `JSON` je textový formát. `JSON` soubory se musí otvírat vždy v textovém režimu a se znakovým kódováním `UTF-8`.

[3] Hmm, tohle není dobré. Co se to vlastně stalo?

Stalo se následující: funkce `json.dump()` se pokusila o serializaci objektu typu bytes s hodnotou `b'\xDE\xD5\xB4\xF8'`, ale selhala, protože v `JSON` podpora objektů typu `bytes` chybí. Pokud je ale pro nás ukládání bajtů důležité, můžeme si definovat náš vlastní „miniserializační formát“.

```
def to_json(python_object):                                             [1]
    if isinstance(python_object, bytes):                                [2]
        return {'__class__': 'bytes',
                '__value__': list(python_object)}                       [3]
    raise TypeError(repr(python_object) + ' is not JSON serializable')  [4]
```

[1] Abychom definovali vlastní „miniserializační formát“ pro datový typ, který `JSON` přirozeně nepodporuje, musíme definovat funkci, která přebírá pythonovský objekt jako parametr. Tímto pythonovským objektem bude skutečný objekt, který funkce `json.dump()` není schopna sama serializovat. V našem případě je to objekt typu bytes s hodnotou `b'\xDE\xD5\xB4\xF8'`.

[2] Naše uživatelská serializační funkce by měla zkontrolovat typ pythonovského objektu, který jí předala funkce `json.dump()`. Pokud funkce serializuje jen jeden datový typ, není to nezbytně nutné. Na druhou stranu se tím vyjasňuje, čím se funkce zabývá. A pokud budeme později potřebovat přidat serializaci pro více datových typů, půjde to snadněji.

[3] V tomto případě jsem se rozhodl převést objekt typu `bytes` na slovník. Klíč `__class__` bude obsahovat původní datový typ (v řetězcové podobě, `'bytes'`) a klíč `__value__` bude obsahovat aktuální hodnotu. Nemůže to, samozřejmě, být objekt typu `bytes`. Celý vtip spočívá v převodu na něco, co může být serializováno v `JSON`! Objekt typu bytes je posloupností celých čísel, kde každé číslo nabývá hodnot z rozsahu 0–255. Pro převod objektu typu bytes na seznam čísel můžeme použít funkci `list()`. Takže z `b'\xDE\xD5\xB4\xF8'` se stane `[222, 213, 180, 248]`. (Počítejte! Funguje to! Bajt zapsaný šestnáctkově `\xDE` je dekadicky `222`, `\xD5` je `213` a tak dále.)

[4] Tento řádek je důležitý. Datová struktura, kterou serializujete, může obsahovat typy, které nejsou ani zabudované do serializátoru json a nezvládne je ani náš uživatelský serializátor. V takovém případě musí náš uživatelský serializátor vyvolat výjimku `TypeError`, aby se funkce `json.dump()` dozvěděla, že náš uživatelský serializátor daný typ nezná.

A to je vše. Nemusíme dělat nic jiného. Konkrétně tato uživatelská serializační funkce *vrací pythonovský slovník* a ne řetězec. Nemusíme sami realizovat celou „serializaci do `JSON`“. Provedeme pouze část „konverze na podporovaný datový typ“. Funkce `json.dump()` udělá zbytek.

```
>>> shell
1
>>> import customserializer                                                    [1]
>>> with open('entry.json', 'w', encoding='utf-8') as f:                       [2]
...     json.dump(entry, f, default=customserializer.to_json)                  [3]
...
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
    json.dump(entry, f, default=customserializer.to_json)
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "/Users/pilgrim/diveintopython3/examples/customserializer.py", line 12, in to_json
    raise TypeError(repr(python_object) + ' is not JSON serializable')       [4]
TypeError: time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20,
tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1) is not JSON serializable
```

[1] Jméno `customserializer` patří modulu, ve kterém jsme (v předchozím příkladu) definovali funkci `to_json()`.

[2] Textový režim, kódování UTF-8 atd., atd. (Jednou na to zapomenete! Já na to taky občas zapomenu! A všechno bude fungovat správně až do chvíle, kdy se to pokazí. Ale pak se to pokazí se vší parádou.)

[3] Tohle je důležitá část. Abychom navěsili svou převodní funkci na funkci `json.dump()`, předáme ji při volání funkce `json.dump()` jako hodnotu parametru `default`. (Hurá! V Pythonu je objektem všechno.)

[4] No dobrá, ono to všechno nefunguje. Ale podívejte se na výjimku. Funkce `json.dump()` už si nestěžuje na to, že není schopna serializovat objekt typu `bytes`. Teď už si stěžuje na úplně jiný objekt — `time.struct_time`.

Mohlo by se zdát, že výskyt jiné výjimky není známkou pokroku. Jenže on opravdu je známkou pokroku! Bude stačit jedno malé pošťouchnutí a překonáme i tohle.

```
import time

def to_json(python_object):
    if isinstance(python_object, time.struct_time):              [1]
        return {'__class__': 'time.asctime',
                '__value__': time.asctime(python_object)}        [2]
    if isinstance(python_object, bytes):
        return {'__class__': 'bytes',
                '__value__': list(python_object)}
    raise TypeError(repr(python_object) + ' is not JSON serializable')
```

[1] Při rozšiřování existující funkce `customserializer.to_json()` potřebujeme zkontrolovat, zda je pythonovský objekt (s kterým má funkce `json.dump()` potíže) typu `time.struct_time`.

[2] Pokud tomu tak je, uděláme podobný převod jako v případě objektu typu `bytes`. Objekt typu `time.struct_time` převedeme na slovník, který bude obsahovat pouze hodnoty, které lze serializovat do JSON. V našem případě je nejsnadnější způsob převodu data a času na hodnotu serializovatelnou do JSON založen na převodu na řetězec pomocí funkce `time.asctime()`. Funkce `time.asctime()` převádí odporně vypadající `time.struct_time` na řetězec `'Fri Mar 27 22:20:42 2009'`.

Při použití těchto dvou uživatelských konverzí proběhne serializace celé datové struktury `entry` do JSON bez dalších problémů.

```
>>> shell
1
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f, default=customserializer.to_json)
...
```

```
you@localhost:~/diveintopython3/examples$ ls -l example.json
-rw-r--r-- 1 you  you  391 Aug  3 13:34 entry.json
you@localhost:~/diveintopython3/examples$ cat example.json
{"published_date": {"__class__": "time.asctime", "__value__": "Fri Mar 27 22:20:42 2009"},
"comments_link": null, "internal_id": {"__class__": "bytes", "__value__": [222, 213, 180, 248]},
"tags": ["diveintopython", "docbook", "html"], "title": "Dive into history, 2009 edition",
"article_link": "http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition",
"published": true}
```

13.11. Načítání dat z JSON souboru

Modul `json` obsahuje (stejně jako modul `pickle`) funkci `load()`, která přebírá objekt typu stream, čte z něj data v notaci JSON a vytváří nový pythonovský objekt, který odráží datovou strukturu JSON.

```
>>> shell
2
>>> del entry                                                    [1]
>>> entry
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import json
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f)                                     [2]
...
>>> entry                                                       [3]
{'comments_link': None,
 'internal_id': {'__class__': 'bytes', '__value__': [222, 213, 180, 248]},
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': {'__class__': 'time.asctime', '__value__': 'Fri Mar 27 22:20:42 2009'},
 'published': True}
```

[1] Pro demonstrační účely se přepneme do pythonovského shellu č. 2 a zrušíme tam datovou strukturu `entry`, kterou jsme v této kapitole vytvořili dříve, použitím modulu `pickle`.

[2] V nejjednodušším případě pracuje funkce `json.load()` stejně jako funkce `pickle.load()`. Předáme jí objekt typu stream a vrací nový pythonovský objekt.

[3] Mám pro vás dobrou a špatnou zprávu. Nejdříve tu dobrou. Funkce `json.load()` úspěšně přečetla soubor `entry.json`, který jsme vytvořili v pythonovském shellu č. 1, a vytvořila nový pythonovský objekt, který data obsahuje. Teď ta špatná zpráva. Nevznikla tím původní datová struktura `entry`. Hodnoty `'internal_id'` a `'published_date'` byly vytvořeny jako slovníky.

Jde konkrétně o slovníky obsahující hodnoty slučitelné s JSON, které jsme vytvořili převodní funkcí to_json().

Funkce json.load() neví nic o konverzních funkcích, které jste mohli předat funkci json.dump(). Potřebujeme vytvořit funkci, která je opakem k funkci to_json(). Potřebujeme funkci, která převezme uživatelsky převedený objekt JSON a konvertuje jej zpět na původní pythonovský datový typ.

```
# do customserializer.py přidejte následující
def from_json(json_object):                                    [1]
    if '__class__' in json_object:                             [2]
        if json_object['__class__'] == 'time.asctime':
            return time.strptime(json_object['__value__'])     [3]
        if json_object['__class__'] == 'bytes':
            return bytes(json_object['__value__'])             [4]
    return json_object
```

[1] Tato převodní funkce také přebírá jeden parametr a vrací jednu hodnotu. Ale parametrem není řetězec. Je jím pythonovský objekt, který je výsledkem deserializace řetězce v notaci JSON do pythonovského objektu.

[2] Potřebujeme pouze zkontrolovat, zda tento objekt obsahuje klíč '__class__', který vytvořila funkce to_json(). Pokud tomu tak je, říká hodnota klíče '__class__', jak máme hodnotu dekódovat zpět na původní pythonovský datový typ.

[3] K dekódování řetězce s časem, který vrátila funkce time.asctime(), použijeme funkci time.strptime(). Tato funkce přebírá naformátovaný řetězec s datem a časem (v upravitelném formátu, ale s výchozím tvarem stejným, jaký používá funkce time.asctime()) a vrací time.struct_time.

[4] Pro převod seznamu celých čísel na objekt typu bytes můžeme použít funkci bytes().

A je to. Ve funkci to_json() se upravovaly jen dva datové typy. Stejné datové typy jsme teď zpracovali funkcí from_json(). A takhle vypadá výsledek:

```
>>> shell
2
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f, object_hook=customserializer.from_json)     [1]
...
>>> entry                                                               [2]
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
```

```
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22,
tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
```

[1] Funkci `from_json()` k deserializačnímu procesu připojíme tím, že ji předáme jako parametr `object_hook` funkci `json.load()`. Funkce, která přebírá funkci. Jak šikovné!

[2] Datová struktura `entry` teď obsahuje klíč `'internal_id'`, jehož hodnotou je objekt typu `bytes`. Obsahuje také klíč `'published_date'`, jehož hodnotou je objekt typu `time.struct_time`.

Ale má to ještě jednu mouchu.

```
>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry2 = json.load(f, object_hook=customserializer.from_json)
...
>>> entry2 == entry                                                    [1]
False
>>> entry['tags']                                                      [2]
('diveintopython', 'docbook', 'html')
>>> entry2['tags']                                                     [3]
['diveintopython', 'docbook', 'html']
```

[1] Dokonce ani po připojení funkce `to_json()` k serializaci a připojení funkce `from_json()` k deserializaci se nám stále nepodařilo vytvořit dokonalou repliku původní datové struktury. Proč tomu tak je?

[2] V původní datové struktuře `entry` byla hodnotou klíče `'tags'` n-tice tří řetězců (tedy trojice řetězců).

[3] Ale v datové struktuře `entry2`, kterou jsme dostali převodem tam a zase zpět, má klíč `'tags'` hodnotu seznamu těchto tří řetězců. `JSON` nedělá rozdíl mezi n-ticemi a seznamy. Zná jen jeden seznamu se podobající datový typ — typ pole. Modul `json` během serializace potichu konvertuje jak n-tice, tak seznamy na pole v `JSON`. Při většině použití můžete rozdíl mezi n-ticemi a seznamy ignorovat. Ale pokud pracujete s modulem `json`, měli byste na to myslet.

13.12. Přečtěte si

> Řada článků o modulu `pickle` se odkazuje na `cPickle`. V Pythonu 2 existovaly dvě implementace modulu `pickle`. Jedna byla napsána v Pythonu a druhá v jazyce C (ale dala se volat z Pythonu). V Pythonu 3 byly tyto moduly spojeny, takže pokaždé provádíme jen import `pickle`. Zmíněné články mohou být užitečné, ale informaci o `cPickle` (která je nyní zastaralá) byste měli ignorovat.

O „piklení“ s modulem `pickle`:

- **`pickle` module**
  (http://docs.python.org/py3k/library/pickle.html)
- **`pickle and cPickle` — Python object serialization**
  (http://www.doughellmann.com/PyMOTW/pickle)
- **Using `pickle`**
  (http://wiki.python.org/moin/UsingPickle)
- **Python persistence management**
  (www.ibm.com/developerworks/library/l-pypers.html)

O `JSON` a o modulu `json`:

- **`json` — JavaScript Object Notation Serializer**
  (www.doughellmann.com/PyMOTW/json/)
- **`JSON` encoding and ecoding with custom objects in Python**
  (http://blog.quaternio.net/2009/07/16/json-encoding-and-decoding-with-custom-objects-in-python/)

O rozšiřitelnosti modulu `pickle`:

- **Pickling class instances**
- **Persistence of external objects**
- **Handling stateful objects**
  (vše na http://docs.python.org/py3k/library/pickle.html)

“A ruffled mind makes a restless pillow.”
(Rozbouřená mysl je nepohodlný polštář.)

— Charlotte Bronteová

14. Webové služby nad HTTP

14. Webové služby nad HTTP — 297
14.1. Ponořme se — **299**
14.2. Vlastnosti HTTP — **300**
14.2.1. Používání mezipaměti — **300**
14.2.2. Kontrola Last-Modified — **301**
14.2.3. Kontrola ETag — **303**
14.2.4. Komprese — **304**
14.2.5. Přesměrování — **304**
14.3. Jak se nedostat k datům přes HTTP — **305**
14.4. Co že to máme na drátě? — **306**
14.5. Představujeme `httplib2` — **309**
14.5.1. Krátká odbočka vysvětlující, proč `httplib2` vrací bajty místo řetězců — **311**
14.5.2. Jak `httplib2` zachází s mezipamětí — **312**
14.5.3. Jak `httplib2` zachází s hlavičkami `Last-Modified` a `ETag` — **315**
14.5.4. Jak `http2lib` pracuje s kompresí — **318**
14.5.5. Jak `httplib2` řeší přesměrování — **318**
14.6. Za hranicemi HTTP GET — **322**
14.7. Za hranicemi HTTP POST — **326**
14.8. Přečtěte si — **328**

14.1. Ponořme se

Z filozofického hlediska můžeme webové služby nad HTTP (HyperText Transfer Protocol) popsat devíti slovy: výměna dat se vzdálenými servery pouze s použitím operací protokolu HTTP. Pokud chceme ze serveru získat data, použijeme HTTP GET. Pokud chceme nová data na server zaslat, použijeme HTTP POST. Některá pokročilejší aplikační rozhraní (API) webových služeb nad HTTP umožňují také vytváření, modifikaci a rušení dat použitím http PUT a HTTP DELETE. To je vše. Žádné registry, žádné obálky, žádný obalující kód, žádné tunelování. „Slovesa“, která jsou součástí http protokolu (GET, POST, PUT a DELETE) přímo odpovídají operacím na aplikační úrovni pro získávání, vytváření, modifikaci a rušení dat.

Hlavní výhodou tohoto přístupu je jednoduchost a právě jednoduchost vedla k jeho oblibě. Data — obvykle XML nebo JSON — mohou být vytvořena a uložena jako statická, nebo mohou být generována dynamicky, skriptem na straně serveru. Všechny hlavní programovací jazyky (samozřejmě včetně Pythonu) umožňují stahování těchto dat prostřednictvím svých HTTP-knihoven. Jednodušší je i ladění. Každý prostředek (resource) webové služby nad HTTP má jednoznačnou adresu v podobě URL. Po zadání do webového prohlížeče dojde k načtení a hned vidíte surová data.

Příklady webových služeb nad HTTP:

- **Aplikační rozhraní Google Data** vám umožní uživatelsky pracovat s celou řadou služeb Google, včetně **Blogger** a **YouTube**.
- **Flickr Services** vám umožní odesílat a stahovat fotografie z **Flickr**.
- **Twitter** API vám umožní zveřejňovat krátké zprávy na **Twitter**.
- **…a řada dalších**

Pro interakci s webovými službami nad HTTP jsou v Pythonu 3 k dispozici dvě různé knihovny:

- http.client je nízkoúrovňová knihovna, která implementuje RFC 2616, tedy HTTP-protokol.
- urllib.request je knihovna na vyšší úrovni abstrakce, vybudovaná nad http.client. Poskytuje standardní aplikační rozhraní pro zpřístupňování jak HTTP, tak FTP serverů, automaticky následuje přesměrování HTTP a podporuje některé běžné formy autentizace v HTTP.

Takže který mám použít? Z těchto dvou žádný. Místo toho byste měli použít httplib2, což je open source knihovna třetí strany, která implementuje HTTP do větších detailů než http.client. Současně používá lepší abstrakce než urllib.request.

Abyste porozuměli tomu, proč je httplib2 tou správnou volbou, musíte nejdříve porozumět HTTP.

14.2. Vlastnosti HTTP

Každý HTTP klient by měl podporovat pět důležitých vlastností.

14.2.1. Používání mezipaměti

Cache-Control: max-age znamená „neotravujte mě až do příštího týdne“..

Nejdůležitější věcí, které musíme v souvislosti s libovolným typem webové služby rozumět, je to, že přístup k síti je velmi drahý. Nemám na mysli cenu „v penězích“ (i když šířka přenosového pásma není zadarmo). Mám na mysli to, že hrozně dlouhou dobu zabere otevření spojení, odeslání požadavku a získání odezvy ze vzdáleného serveru. Dokonce i v případě nejrychlejšího dostupného spojení může být *latence* (tj. čas mezi zasláním požadavku a zahájením přijímání dat odpovědi) vyšší, než byste předpokládali. Směrovače mohou zafungovat divně, paket se ztratí, na mezilehlý server někdo zaútočil... Na veřejné internetové síti není nikdy klidná chvilka a nic s tím nenaděláte.

Při návrhu HTTP se počítalo s využíváním mezipaměti (cache). Existuje dokonce samostatná třída zařízení (zvaných „mezipaměťové proxy-servery“, anglicky „chaching proxies“), jejichž jedinou prací je ležet mezi vámi a zbytkem světa a minimalizovat zatěžování sítě. Vaše firma nebo váš poskytovatel připojení (ISP) téměř jistě mezipaměťové proxy-servery udržuje, i když si toho nemusíte být vědomi. Fungují, protože používání mezipaměti (caching) je součástí HTTP protokolu.

Následuje konkrétní příklad toho, jak to funguje. Prostřednictvím svého prohlížeče navštívíte `diveintomark.org`. Uvedená stránka používá pro pozadí obrázek `wearehugh.com/m.jpg`. Když váš prohlížeč obrázek stáhne, server k němu přiloží následující HTTP hlavičky:

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

Hlavičky `Cache-Control` a `Expires` říkají vašemu prohlížeči (a všem mezipaměťovým proxy-serverům mezi vámi a serverem), že se tento obrázek může získávat z mezipaměti až jeden rok. *Celý rok!* A pokud

někdy v příštím roce navštívíte jinou stránku, která také obsahuje odkaz na tento obrázek, váš prohlížeč jej načte ze své mezipaměti, *aniž by vyvolal jakoukoliv síťovou aktivitu*.

Ale počkejte, bude to ještě lepší. Dejme tomu, že váš prohlížeč obrázek z lokální mezipaměti z nějakého důvodu odstraní. Možná mu došlo místo na disku, možná jste mezipaměť vyprázdnili ručně. Z jakéhokoliv důvodu. Ale HTTP hlavičky říkají, že tato data mohou být uchovávána veřejnými mezipaměťovými proxy-servery. (Z technického pohledu je důležité, co hlavičky *neříkají*. Hlavička `Cache-Control` neuvádí klíčové slovo `private`, takže data mohou být uložena v mezipaměti automaticky.) Mezipaměťové proxy-servery jsou navrženy tak, že mají k dispozici obrovské množství úložného prostoru — pravděpodobně ho mají mnohem více, než má vyhrazeno váš lokální prohlížeč.

Pokud vaše firma nebo váš poskytovatel připojení spravuje mezipaměťový proxy-server, může se v jeho mezipaměti obrázek pořád ještě nacházet. Pokud navštívíte `diveintomark.org` znovu, podívá se váš prohlížeč po obrázku do lokální mezipaměti, ale nenajde jej. Takže vytvoří síťový požadavek a pokusí se obrázek stáhnout ze vzdáleného serveru. Pokud ale mezipaměťový proxy-server pořád má kopii uvedeného obrázku, váš požadavek zachytí a dodá vám obrázek ze své mezipaměti. To znamená, že se váš požadavek ke vzdálenému serveru nikdy nedostane. Ve skutečnosti nemusí opustit vaši firemní síť. Získání obrázku je rychlejší (méně skoků po síti) a vaše firma ušetří peníze (z vnějšího světa se stahuje méně dat).

Použití mezipamětí v HTTP funguje, pokud všechny strany dělají, co mají. Na jedné straně musí servery v odpovědích posílat správné hlavičky. Na druhé straně musí klienti hlavičkám rozumět, respektovat je a nežádat stejná data dvakrát. Mezilehlé proxy-servery nejsou všelékem. Mohou být „chytré“ jen do té míry, do jaké jim to servery a klienti umožní.

Standardní pythonovské knihovny pro HTTP používání mezipaměti nepodporují, ale `httplib2` ano.

14.2.2. Kontrola Last-Modified

Některá data se nemění nikdy, zatímco jiná data se mění pořád. A mezi tím je obrovské množství dat, která se mohla změnit, ale nezměnila se. Publikovaný obsah (feed) serveru CNN.com se mění každých pár minut, ale publikovaný obsah mého weblogu se nemusí změnit celé dny nebo týdny. I kdyby to byl ten druhý případ, nechci klientům říct, aby si můj publikovaný obsah brali z mezipaměti celé týdny, protože pokud bych doopravdy něco nového zveřejnil, lidé by se o tom celé týdny nedozvěděli (protože by respektovali mé hlavičky týkající se mezipaměti, které říkají „neobtěžujte se s kontrolou tohoto publikovaného obsahu po celé týdny“). Na druhou stranu zase nechci, aby klienti stahovali celý publikovaný obsah (feed) každou hodinu, pokud se vůbec nezměnil!

304: Not Modified znamená „stejné nesmysly, jiný den“.

HTTP nabízí řešení i pro tento případ. Pokud o data žádáme poprvé, server může zpět poslat hlavičku `Last-Modified` (naposledy změněno). Je to přesně to, jak to vypadá:

datum a čas, kdy se data naposledy změnila. Obrázek pozadí, na který vedl odkaz z `diveintomark.org`, doprovázela hlavička `Last-Modified`.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

Pokud požadujeme stejná data podruhé (nebo potřetí nebo počtvrté), můžeme v dotazu poslat hlavičku `If-Modified-Since` (pokud bylo změněno od) s hodnotou data a času, které jsme od serveru dostali minule. Pokud se data od té doby změnila, pak server vrátí nová data doplněná o stavový kód `200`. Ale pokud se data od té doby *nezměnila*, server pošle zpět speciální stavový kód protokolu `HTTP` — `304`. Ten říká „od doby, kdy ses naposledy ptal, se tato data nezměnila“. Z příkazového řádku si to můžeme ověřit nástrojem `curl`:

```
you@localhost:~$ curl -I -H "If-Modified-Since: Fri, 22 Aug 2008 04:28:16 GMT" http://wearehugh.com/m.jpg

HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

A proč by to mělo být vylepšení? Protože když server pošle `304`, *neposílá data znovu*. Dostaneme pouze stavový kód. Kontrola poslední modifikace zajistí, že se nezměněná data nebudou stahovat podruhé i v případě, kdy došlo k vypršení platnosti kopie v lokální mezipaměti. (Jako bonus navíc obsahuje odpověď `304` také hlavičky pro mezipaměť. Proxy-servery si kopii dat drží, dokonce i když oficiálně „expirovala“, v naději, že se data *ve skutečnosti* nezměnila a že další požadavek povede k odpovědi se stavovým kódem `304` a s aktualizovanými informacemi pro mezipaměť.)

Standardní pythonovské knihovny pro `HTTP` nepodporují kontrolu data poslední modifikace, ale `httplib2` ano.

14.2.3. Kontrola ETag

ETagy (tag = značka) představují alternativní způsob dosažení stejného efektu jako v případě kontroly last-modified. Při použití ETagů posílá server spolu s požadovanými daty v hlavičce `ETag` s heš-kódem (hash). (Jak se přesně heš-hodnota určí, to závisí zcela na serveru. Jediný požadavek je takový, aby se změnila, pokud se změní data.) Obrázek pozadí, na který vedl odkaz z `diveintomark.org`, doprovázela hlavička `ETag`.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

ETag vyjadřuje „nic nového pod sluncem“.

Pokud stejná data požadujeme podruhé, přiložíme heš-hodnotu v hlavičce pořadavku `If-None-Match` (pokud žádná data neodpovídají). Pokud se data nezměnila, server pošle zpět stavový kód `304`. Server — stejně jako v případě kontroly založené na čase poslední modifikace — pošle zpět *pouze* stavový kód `304`. Stejná data znovu neposílá. Přiložením heš-hodnoty v ETagu při druhém požadavku serveru říkáme, že při shodě heše není nutné posílat stejná data znovu, protože je pořád máme schovaná od minula.

Opět vyzkoušíme pomocí `curl`:

```
you@localhost:~$ curl -I -H "If-None-Match: \"3075-ddc8d800\"" http://wearehugh.com/m.jpg  [1]
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

[1] ETagy se běžně uzavírají do uvozovek, ale *tyto uvozovky jsou součástí hodnoty*. To znamená, že v hlavičce `If-None-Match` musíme serveru poslat zpět i uvozovky.

Standardní pythonovské knihovny pro HTTP používání ETagů nepodporují, ale httplib2 ano.

14.2.4. Komprese

Pokud se bavíme o webových službách nad HTTP, pak se téměř vždy bavíme o přesunování textových dat po drátech tam a zase zpět. Možná jsou ve formátu XML, možná jsou v JSON, možná je to prostý text. Text se dá dobře komprimovat nezávisle na použitém formátu. Příklad publikovaného obsahu (feed) z kapitoly XML má nekomprimovaný 3070 bajtů, ale po kompresi algoritmem gzip má 941 bajtů. To je jen 30 % původní velikosti!

Kap.

HTTP podporuje několik komprimačních algoritmů. Mezi dva nejběžnější patří gzip a deflate. Pokud přes HTTP požadujeme nějaký prostředek (resource), můžeme serveru říci, aby ho poslal v komprimovaném formátu. Do požadavku vložíme hlavičku Accept-encoding, ve které vyjmenujeme námi podporované komprimační algoritmy. Pokud server některý z těchto algoritmů podporuje, pošle nám zpět komprimovaná data (s hlavičkou Content-encoding, která říká, jaký algoritmus byl použit).
O dekompresi se už musíme postarat sami.

> Důležitý tip pro vývojáře kódu na straně serveru: Ujistěte se, že komprimovaná podoba zdroje dostane přidělenou jinou značku Etag než nekomprimovaná verze. V opačném případě by došlo ke zmatení mezipaměťových proxy-serverů a ty by mohly klientům vracet komprimovanou verzi, se kterou by si klient nemusel poradit. Více detailů o této delikátní záležitosti si můžete přečíst v diskusi Apache bug 39727.

Standardní pythonovské knihovny pro HTTP kompresi nepodporují, ale httplib2 ano.

14.2.5. Přesměrování

Senzační URI se nemění, ale mnohá uri jsou opravdu... nesenzační. Webová místa se reorganizují, stránky se přesouvají na nové adresy. Dokonce i webové služby mohou být reorganizovány. Publikovaný obsah (syndicated feed) mohl být přesunut z http://example.com/index.xml do http://example.com/xml/atom.xml.

Nebo se při rozšiřování a reorganizaci firmy mohla přesunout celá doména.
Z http://www.example.com/index.xml se mění na http://server-farm-1.example.com/index.xml.

Location znamená „podívej se támhle“!

Pokaždé, když HTTP server požádáme o nějaký zdroj (resource), vrací v odpovědi stavový kód. Stavový kód 200 znamená „vše v pořádku, tady je požadovaná stránka“. Stavový kód 404 znamená „stránka nenalezena“. (Chybu 404 jste už asi při brouzdání po webu viděli.) Stavové kódy ve skupině 300 vyjadřují nějakou formu přesměrování.

HTTP nabízí několik způsobů, jakými se dá oznámit, že se požadované zdroje přesunuly. Dvě nejběžnější techniky používají stavové kódy 302 a 301. Stavový kód 302 označuje *dočasné přesměrování*. Znamená „ejhle, je to dočasně přesunuté“ (a v hlavičce Location se vrátí dočasná adresa). Stavový kód 301 označuje *trvalé přesměrování*. Znamená „ejhle, je to trvale přesunuté“ (a v hlavičce Location se vrací nová adresa). Pokud obdržíte stavový kód 302 a novou adresu, pak máte podle specifikace HTTP pro požadovanou věc použít novou adresu. Ale až se budete na stejný zdroj informací ptát příště, máte to znovu zkusit s původní adresou. Pokud ale obdržíte stavový kód 301 a k němu novou adresu, očekává se od vás, že od toho okamžiku začnete používat novou adresu.

Modul urllib.request při obdržení příslušného stavového kódu od HTTP serveru sice „následuje“ přesměrování, ale neřekne vám, že tato situace nastala. Dostanete data, která jste požadovali, ale nikdy se nedozvíte, že se použitá knihovna zachovala „užitečně“ a následovala přesměrování za vás. Takže pořád bušíte na staré adrese a pokaždé jste serverem přesměrováni na novou adresu a modul urllib.request pokaždé „užitečně“ následuje přesměrování. Jinými slovy, tato knihovna se k trvalému přesměrování chová stejně jako k dočasnému přesměrování. To znamená, že se místo jednoho kola provedou vždycky dvě. To je špatné jak pro server, tak pro vás.

Knihovna httplib2 trvalé přesměrování zvládá. Nejen že vám řekne, že nastalo trvalé přesměrování, ale lokálně si je poznamená a přesměrovaná URL automaticky přepíše dříve, než vznese příslušný požadavek.

14.3. Jak se nedostat k datům přes HTTP

Dejme tomu, že přes HTTP chceme stáhnout informační zdroj, jako je například Atom feed. Protože jde o publikovaný obsah (feed), nebudeme jej stahovat jen jednou. Budeme jej stahovat opakovaně, pořád dokola. (Většina čteček publikovaného obsahu (feed reader) kontroluje změny každou hodinu.) Nejdříve vyzkoušíme „rychlý a špinavý“ způsob a pak se podíváme, jak bychom to mohli provádět lépe.

```
>>> import urllib.request
>>> a_url = 'http://diveintopython3.org/examples/feed.xml'
>>> data = urllib.request.urlopen(a_url).read()                    [1]
>>> type(data)                                                     [2]
<class 'bytes'>
>>> print(data)
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  …
```

[1] Stažení čehokoliv přes HTTP je v Pythonu neuvěřitelně jednoduché. Dá se to ve skutečnosti napsat na jeden řádek. Modul `urllib.request` nabízí šikovnou funkci `urlopen()`, která přebírá adresu požadované stránky a vrací objekt typu stream, ze kterého získáme celý obsah stránky prostým zavoláním metody `read()`. Už to asi nemůže být jednodušší.

[2] Metoda `urlopen().read()` vrací vždy objekt typu `bytes` a ne řetězec. Vzpomeňte si — bajty jsou bajty, znaky jsou abstrakce. HTTP servery nepracují s abstrakcemi. Kdykoliv požádáme o nějaký zdroj (resource), dostaneme bajty. Pokud z toho chceme udělat řetězec, musíme zjistit znakové kódování a provést explicitní převod na řetězec.

A co na tom je špatného? Při rychlém, jednorázovém přístupu během ladění a vývoje na tom není špatného nic. Dělám to takhle pořád. Chtěl jsem publikovaný obsah (feed), dostal jsem publikovaný obsah. Stejná technika funguje pro libovolné webové stránky. Ale jakmile o tom začneme uvažovat z pohledu webové služby, která se má využívat pravidelně (tj. požadavek na získání publikovaného obsahu každou hodinu), pak by to bylo neefektivní a my bychom byli nezdvořilí.

14.4. Co že to máme na drátě?

Abychom viděli, proč je to neefektivní a nezdvořilé, obrátíme se na ladicí prostředky pythonovské knihovny pro HTTP a uvidíme, co běhá „po drátech“ (tj. co se přenáší v síti).

```
>>> from http.client import HTTPConnection
>>> HTTPConnection.debuglevel = 1                                    [1]
>>> from urllib.request import urlopen
>>> response = urlopen('http://diveintopython3.org/examples/feed.xml')  [2]
send: b'GET /examples/feed.xml HTTP/1.1                              [3]
Host: diveintopython3.org                                            [4]
Accept-Encoding: identity                                            [5]
User-Agent: Python-urllib/3.1'                                       [6]
Connection: close
reply: 'HTTP/1.1 200 OK'
…další ladicí informace vypuštěny…
```

[1] Jak už jsem se zmínil na začátku této kapitoly, `urllib.request` spoléhá na další standardní pythonovskou knihovnu, `http.client`. S knihovnou `http.client` za normálních okolností do přímého styku nepřicházíte. (Modul `urllib.request` ji importuje automaticky.) Ale my si ji importujeme ručně, abychom mohli nastavit příznak ladění u třídy `HTTPConnection`, kterou modul `urllib.request` používá pro připojení k HTTP serveru.

[2] Když teď máme ladicí příznak nastaven, budou se informace o HTTP požadavku a o odpovědi na něj tisknout v reálném čase. Když si vyžádáme Atom feed, je vidět, že modul `urllib.request` posílá serveru pět řádků.

[3] První řádek uvádí používané HTTP sloveso (metodu; zde GET) a cestu ke zdroji (bez uvedení jména domény).

[4] Druhý řádek uvádí doménu, ze které byl požadavek na feed vznesen.

[5] Třetí řádek uvádí komprimační algoritmy, které klient podporuje. Jak bylo uvedeno výše, `urllib.request` standardně kompresi nepodporuje.

[6] Čtvrtý řádek uvádí jméno knihovny, jejímž prostřednictvím byl požadavek vznesen. Výchozí hodnotou je `Python-urllib` a číslo verze. Jak `urllib.request`, tak `httplib2` podporují změnu identifikace zprostředkovatele tím, že se do požadavku jednoduše přidá hlavička `User-Agent`, která přepíše výchozí hodnotu.

Teď se podívejme na to, jakou odpověď poslal server zpět.

Stahovali jsme 3070 bajtů, i když bychom mohli stahovat pouhých 941.

```
# pokračování předchozího příkladu
>>> print(response.headers.as_string())                    [1]
Date: Sun, 31 May 2009 19:23:06 GMT                        [2]
Server: Apache
Last-Modified: Sun, 31 May 2009 06:39:55 GMT               [3]
ETag: "bfe-93d9c4c0"                                       [4]
Accept-Ranges: bytes
Content-Length: 3070                                       [5]
Cache-Control: max-age=86400                               [6]
Expires: Mon, 01 Jun 2009 19:23:06 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml
>>> data = response.read()                                 [7]
>>> len(data)
3070
```

[1] Odpověď (response) vrácená funkcí `urllib.request.urlopen()` obsahuje všechny `HTTP` hlavičky, které server poslal zpět. Obsahuje také metody pro stahování skutečných dat. K tomu se dostaneme za minutku.

[2] Server říká, kdy zpracoval náš požadavek.

[3] Odpověď obsahuje i hlavičku `Last-Modified`.

[4] Odpověď obsahuje také hlavičku `ETag`.

[5] Data mají velikost 3070 bajtů. Všimněte si, že zde *není* hlavička `Content-encoding`. V požadavku jsme uvedli, že přijímáme jen nekomprimovaná data (`Accept-encoding: identity`), takže jsme tím pádem dostali nekomprimovaná data.

[6] V odpovědi se nacházejí hlavičky pro mezipaměti, které říkají, že publikovaný obsah (feed) může být brán z mezipaměti po dobu 24 hodin (86 400 sekund).

[7] A nakonec stáhneme skutečná data voláním `response.read()`. Z výsledku funkce `len()` vidíme, že se stáhlo všech 3070 bajtů najednou.

Jak sami vidíte, tento kód je už teď neefektivní. Požadoval (a obdržel) nekomprimovaná data. Určitě vím, že uvedený server podporuje kompresi `gzip`, ale v `HTTP` se komprese zapíná na vyžádání. Nepožádali jsme o ni, tak jsme ji nedostali. To znamená, že jsme stahovali 3070 bajtů v situaci, kdy jsme mohli stahovat pouhých 941. Zlobivý pejsek, žádná sušenka.

Ale moment, začíná to být ještě horší! Abychom viděli, jak neefektivní ten kód je, požádáme o stejný publikovaný obsah (feed) podruhé.

```
# pokračování předchozího příkladu
>>> response2 = urlopen('http://diveintopython3.org/examples/feed.xml')
send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
Accept-Encoding: identity
User-Agent: Python-urllib/3.1'
Connection: close
reply: 'HTTP/1.1 200 OK'
…další ladicí informace vypuštěny…
```

Všimli jste si na tom požadavku něčeho zvláštního? Vůbec se nezměnil! Je naprosto stejný jako ten předchozí. Žádná známka použití hlavičky `If-Modified-Since`. Žádná známka použití hlavičky `If-None-Match`. Žádný respekt k hlavičkám mezipaměti. Ještě pořád žádná komprese.

A co se stane, když uděláme stejnou věc dvakrát? Dostaneme stejnou odpověď. Dvakrát.

```
# pokračování předchozího příkladu
>>> print(response2.headers.as_string())          [1]
Date: Mon, 01 Jun 2009 03:58:00 GMT
Server: Apache
Last-Modified: Sun, 31 May 2009 22:51:11 GMT
ETag: "bfe-255ef5c0"
Accept-Ranges: bytes
Content-Length: 3070
Cache-Control: max-age=86400
Expires: Tue, 02 Jun 2009 03:58:00 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml
>>> data2 = response2.read()
>>> len(data2)                                    [2]
3070
>>> data2 == data                                 [3]
True
```

[1] Server pořád posílá stejné pole „chytrých“ hlaviček: `Cache-Control` a `Expires` pro mezipaměť (cache), `Last-Modified` a `ETag` pro sledování „nezměněného stavu“. A dokonce hlavičku `Vary: Accept-Encoding`, kterou server dává najevo, že by mohl podporovat kompresi, kdybychom si o ni řekli. Ale my jsme to neudělali.

[2] A ještě jednou, při získávání dat se stáhlo všech 3070 bajtů…

[3] …stejných 3070 bajtů, které jsme stáhli už minule.

Protokol `HTTP` je navržen, aby pracoval lepším způsobem. Knihovna `urllib` umí `HTTP` asi tak, jak já umím španělsky — dost na to, abych se dostal z problémů, ale ne dost k vedení konverzace. A `HTTP` se týká konverzace. Je čas přejít ke knihovně, která protokolem `HTTP` mluví plynule.

14.5. Představujeme `httplib2`

Než začneme knihovnu `httplib2` používat, musíme ji nainstalovat. Navštivte stránku `code.google.com/p/httplib2/` a stáhněte poslední verzi. `httplib2` je k dispozici pro Python 2.x a pro Python 3.x. Ujistěte se, že jde o verzi pro Python 3. Jmenuje se podobně jako `httplib2-python3-0.5.0.zip`. (V době překladu už to bylo jinak: `httplib2-0.6.0.zip`; uvnitř jsou obě verze.)

Rozbalte archiv, otevřete terminálové okno a přejděte do nově vytvořeného adresáře `httplib2`. Pod Windows otevřete menu `Start`, vyberte `Run...`, napište `cmd.exe` a stiskněte `ENTER`.

```
c:\Users\pilgrim\Downloads> dir
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\Downloads

07/28/2009  12:36 PM    <DIR>          .
07/28/2009  12:36 PM    <DIR>          ..
07/28/2009  12:36 PM    <DIR>          httplib2-python3-0.5.0
07/28/2009  12:33 PM            18,997 httplib2-python3-0.5.0.zip
               1 File(s)         18,997 bytes
               3 Dir(s)  61,496,684,544 bytes free

c:\Users\pilgrim\Downloads> cd httplib2-python3-0.5.0
c:\Users\pilgrim\Downloads\httplib2-python3-0.5.0> c:\python31\python.exe setup.py install
running install
running build
running build_py
running install_lib
creating c:\python31\Lib\site-packages\httplib2
```

```
copying build\lib\httplib2\iri2uri.py -> c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\__init__.py -> c:\python31\Lib\site-packages\httplib2
byte-compiling c:\python31\Lib\site-packages\httplib2\iri2uri.py to iri2uri.pyc
byte-compiling c:\python31\Lib\site-packages\httplib2\__init__.py to __init__.pyc
running install_egg_info
Writing c:\python31\Lib\site-packages\httplib2-python3_0.5.0-py3.1.egg-info
```

V Mac OS X spusťte aplikaci `Terminal.app`, kterou najdete ve složce `/Applications/Utilities/`. V Linuxuspusťte aplikaci `Terminal`, kterou obvykle najdete v menu `Applications` pod `Accessories` nebo `System`.

```
you@localhost:~/Desktop$ unzip httplib2-python3-0.5.0.zip
Archive:  httplib2-python3-0.5.0.zip
  inflating: httplib2-python3-0.5.0/README
  inflating: httplib2-python3-0.5.0/setup.py
  inflating: httplib2-python3-0.5.0/PKG-INFO
  inflating: httplib2-python3-0.5.0/httplib2/__init__.py
  inflating: httplib2-python3-0.5.0/httplib2/iri2uri.py
you@localhost:~/Desktop$ cd httplib2-python3-0.5.0/
you@localhost:~/Desktop/httplib2-python3-0.5.0$ sudo python3 setup.py install
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-3.1
creating build/lib.linux-x86_64-3.1/httplib2
copying httplib2/iri2uri.py -> build/lib.linux-x86_64-3.1/httplib2
copying httplib2/__init__.py -> build/lib.linux-x86_64-3.1/httplib2
running install_lib
creating /usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/iri2uri.py -> /usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/__init__.py -> /usr/local/lib/python3.1/dist-packages/
httplib2
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/iri2uri.py to iri2uri.pyc
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/__init__.py to __init__.pyc
running install_egg_info
Writing /usr/local/lib/python3.1/dist-packages/httplib2-python3_0.5.0.egg-info
```

Abychom mohli `httplib2` používat, vytvoříme instanci třídy `httplib2.Http`.

```
>>> import httplib2
>>> h = httplib2.Http('.cache')                                                    [1]
>>> response, content = h.request('http://diveintopython3.org/examples/feed.xml')  [2]
>>> response.status                                                                [3]
200
>>> content[:52]                                                                   [4]
b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content)
3070
```

[1] Primárním rozhraním k `httplib2` je objekt třídy `Http`. Z důvodů, které si ukážeme v další podkapitole, bychom při vytváření objektu třídy `Http` měli vždy předávat jméno adresáře. Adresář nemusí existovat. V případě potřeby si jej `httplib2` vytvoří.

[2] Jakmile máme objekt třídy `Http` k dispozici, můžeme data získat jednoduše tím, že zavoláme metodu `request()` a předáme jí adresu dat. Pro dané url se tím vytvoří požadavek `HTTP GET`. (Později v této kapitole si ukážeme, jak můžeme vytvořit jiné `HTTP` požadavky, jako například `POST`.)

[3] Metoda `request()` vrací dvě hodnoty. První hodnotou je objekt třídy `httplib2.Response`, který obsahuje všechny `HTTP` hlavičky vrácené serverem. Například hodnota stavového kódu (`status`) `200` indikuje, že byl dotaz proveden úspěšně.

[4] Proměnná content obsahuje data, která `HTTP` server vrátil. Data se vracejí jako objekt typu `bytes`, nikoliv jako řetězec. Pokud z toho chceme udělat řetězec, musíme zjistit znakové kódování a převést si je sami.

> Pravděpodobně budete potřebovat jen jeden objekt třídy `httplib2.Http`. Existují rozumné důvody pro vytváření více než jednoho objektu, ale měli byste to dělat jen v případě, kdy víte, proč je potřebujete. „Potřebuji získávat data ze dvou různých `URL`“ takovým důvodem není. Použijte objekt třídy `Http` znovu — prostě zavolejte metodu `request()` dvakrát.

14.5.1. Krátká odbočka vysvětlující, proč `httplib2` vrací bajty místo řetězců

Bajty. Řetězce. To je bolest. Proč `httplib2` nemůže „jednoduše“ provést konverzi za nás? No, ono je to komplikované, protože pravidla pro zjištění znakového kódování jsou specifická v závislosti na tom, jaký zdroj (resource) požadujeme. Jak by mohla `httplib2` vědět, jaký druh zdroje požadujeme? Obvykle bývá uveden v `HTTP` hlavičce `Content-Type`, ale tato hlavička je v `HTTP` nepovinná a ne všechny `HTTP` servery ji vkládají. Pokud tato hlavička není součástí `HTTP` odpovědi, ponechává se odhad na klientovi. (Říká se tomu anglicky „content sniffing“ čili „čmuchání v obsahu“. Výsledek není nikdy perfektní.)

Pokud víme, jaký druh dat očekáváme (v našem případě `XML` dokument), mohli bychom „jednoduše“ předat objekt typu `bytes` funkci `xml.etree.ElementTree.parse()`. To by fungovalo, kdyby `XML` dokument obsahoval informaci o svém vlastním kódování znaků (jako je tomu v tomto případě). Ale jde o nepovinný údaj a ne všechny `XML` dokumenty ho používají. Pokud `XML` dokument informaci o kódo-

vání neobsahuje, měl by se klient podívat na transportní obálku — tj. na HTTP hlavičku Content-Type, která by mohla parametr charset obsahovat.

Ale ono je to ještě horší. Teď už může být informace o kódování uvedena na dvou místech: uvnitř samotného XML dokumentu a uvnitř HTTP hlavičky Content-Type. Jenže když je tato informace uvedena na obou místech, které z nich vyhraje? Podle RFC 3023 platí (a přísahám, to jsem si nevymyslel): pokud je v HTTP hlavičce Content-Type uveden typ média application/xml, application/xml-dtd, application/xml-external-parsed-entity nebo libovolný z podtypů application/xml, jako je application/atom+xml nebo application/rss+xml nebo dokonce application/rdf+xml, pak je kódování rovno

1. kódování zadanému parametrem charset v HTTP hlavičce Content-Type nebo
2. kódování zadanému atributem encoding v XML deklaraci uvnitř dokumentu nebo
3. UTF-8

Na druhou stranu, pokud je v HTTP hlavičce Content-Type uveden typ média text/xml, text/xml-external-parsed-entity nebo podtyp jako text/AnythingAtAll+xml, pak se atribut uvádějící kódování v XML deklaraci uvnitř dokumentu zcela ignoruje a kódování je rovno

1. kódování zadanému parametrem charset v HTTP hlavičce Content-Type nebo
2. us-ASCII

A to se bavíme jen o XML dokumentech. Pro HTML dokumenty vytvořily webové prohlížeče taková byzantská pravidla pro zjišťování obsahu (content-sniffing) [PDF], že se stále ještě snažíme všechna zjistit.

„Opravy jsou vítány.“

14.5.2. Jak httplib2 zachází s mezipamětí

Vzpomínáte si, že jsem vás v předchozí podkapitole nabádal, abyste vždy vytvářeli objekt třídy httplib2.Http se zadaným jménem adresáře? Důvod se jmenuje mezipaměť (cache).

```
# pokračování z předchozího příkladu
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml')  [1]
>>> response2.status                                                                  [2]
200
>>> content2[:52]                                                                     [3]
b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content2)
3070
```

[1] Tohle by vás nemělo moc překvapit. Stejnou věc už jsme dělali naposledy s tou výjimkou, že výsledek ukládáme do dvou nových proměnných.
[2] HTTP opět vrací stavový kód (status) 200, jako minule.
[3] Stažený obsah je také stejný jako minule.

Takže... koho to zajímá? Ukončete pythonovský interaktivní shell a spusťte nové sezení. Hned vám to ukážu.

```
# toto NENÍ pokračování z předchozího příkladu!
# Ukončete, prosím, interaktivní shell
# a spusťte nový.
>>> import httplib2
>>> httplib2.debuglevel = 1                                                    [1]
>>> h = httplib2.Http('.cache')                                                [2]
>>> response, content = h.request('http://diveintopython3.org/examples/feed.xml')  [3]
>>> len(content)                                                               [4]
3070
>>> response.status                                                            [5]
200
>>> response.fromcache                                                         [6]
True
```

[1] Zapněme ladění a podívejme se, co nám lítá po drátech. Takto se v httplib2 zapíná ladicí režim (srovnejte se zapínáním v http.client). httplib2 vytiskne všechna data, která se posílají na server, a některé klíčové informace, které se posílají zpět.
[2] Vytvoříme objekt třídy httplib2.Http se stejným jménem adresáře jako minule.
[3] Vyžádáme si stejné URL jako minule. *Zdá se, že se nic nestalo.* Přesněji řečeno, nic se neposílá na server a ze serveru se nic nevrací. Na síti nepozorujeme vůbec žádnou aktivitu.
[4] Přesto jsme nějaká data „přijali“ — ve skutečnosti jsme dostali všechno.
[5] A „přijali“ jsme také stavový kód protokolu HTTP, který říká, že „požadavek“ byl úspěšný.
[6] Tady je důvod: „odpověď“ byla vygenerována z lokální mezipaměti httplib2. Adresář, jehož jméno jsme zadávali při vytváření objektu třídy httplib2.Http, slouží knihovně httplib2 jako mezipaměť (cache) pro všechny operace, které se kdy provedly.

> Pokud chcete v httplib2 zapnout ladicí režim, musíte nastavit konstantu na úrovni modulu (httplib2.debuglevel) a potom vytvořit nový objekt třídy httplib2.Http. Pokud chcete ladicí režim vypnout, musíte změnit tutéž konstantu na úrovni modulu a potom vytvořit nový objekt třídy httplib2.Http.

Minule jsme požadovali data z konkrétního URL. Požadavek byl úspěšný (status: 200). Odpověď zahrnovala nejen data publikovaného obsahu, ale také množinu hlaviček pro mezipaměť (caching headers). Ty každému příjemci říkají, že si tento zdroj může pamatovat po dobu až 24 hodin (Cache-Control: max-age=86400, což je 24 hodin v sekundách). httplib2 hlavičkám pro mezipaměť rozumí a respektuje je.

Co se děje na drátě? Vůbec nic.

Předchozí odpověď byla uložena do adresáře `.cache` (jehož jméno jsme zadali při vytváření objektu třídy `Http`). Platnost obsahu mezipaměti zatím nevypršela, takže když data ze stejného `URL` požadujeme podruhé, `httplib2` jednoduše vrátí zapamatovaný výsledek, aniž by došlo ke komunikaci po síti.

Říkám „jednoduše“, ale za touto jednoduchostí je evidentně skryto hodně složitostí. Knihovna `httplib2` zvládá používání mezipaměti v `HTTP` *automaticky a aniž se o to musíme starat*. Pokud z nějakého důvodu potřebujeme vědět, zda odpověď přichází z mezipaměti, můžeme zkontrolovat `response.fromcache`. Z jiného pohledu… prostě to funguje.

Dejme tomu, že teď máme data v mezipaměti, ale chceme ji obejít a znovu si je vyžádat od vzdáleného serveru. Prohlížeče to někdy dělají, když si to uživatel vyžádá. Například stisk `F5` obnoví aktuální stránku, ale stiskem `Ctrl+F5` se obejde mezipaměť a aktuální stránka se znovu vyžádá ze vzdáleného serveru. Možná si myslíte „aha, prostě smažu data ze své lokální mezipaměti a provedu požadavek znovu“. Tohle byste udělat mohli. Ale vzpomeňte si, že se to může týkat více stran než jen vás a vzdáleného serveru. Což takhle mezilehlé proxy-servery? Ty jsou zcela mimo vaši kontrolu a pořád mohou uchovávat ona data ve své mezipaměti. A s radostí vám je vrátí, protože obsah jejich mezipaměti je (z jejich pohledu) stále platný.

Takže místo toho, abyste manipulovali s lokální mezipamětí a doufali v nejlepší, měli byste využít vlastností `HTTP` k zajištění toho, že se váš požadavek skutečně dostal až ke vzdálenému serveru.

```
# pokračování předchozího příkladu
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml',
...     headers={'cache-control':'no-cache'})            [1]
connect: (diveintopython3.org, 80)                       [2]
send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
user-agent: Python-httplib2/$Rev: 259 $
accept-encoding: deflate, gzip
cache-control: no-cache'
reply: 'HTTP/1.1 200 OK'
…další ladicí informace vypuštěny…
>>> response2.status
200
>>> response2.fromcache                                  [3]
False
>>> print(dict(response2.items()))                       [4]
{'status': '200',
 'content-length': '3070',
```

```
 'content-location': 'http://diveintopython3.org/examples/feed.xml',
 'accept-ranges': 'bytes',
 'expires': 'Wed, 03 Jun 2009 00:40:26 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Sun, 31 May 2009 22:51:11 GMT',
 'connection': 'close',
 '-content-encoding': 'gzip',
 'etag': '"bfe-255ef5c0"',
 'cache-control': 'max-age=86400',
 'date': 'Tue, 02 Jun 2009 00:40:26 GMT',
 'content-type': 'application/xml'}
```

[1] `httplib2` vám umožní přidat k jakémukoliv odcházejícímu požadavku libovolné `HTTP` hlavičky. Abychom obešli *všechny* mezipaměti (nejen lokální diskovou, ale také mezipaměťové proxy-servery mezi námi a vzdáleným serverem), přidáme do slovníku headers hlavičku `no-cache`.

[2] Teď vidíme, že `httplib2` zahajuje síťový požadavek. `httplib2` rozumí hlavičkám pro mezipaměť a respektuje je *v obou směrech* — jako součást přicházející odpovědi *i jako součást odcházejícího požadavku*. Knihovna si všimla, že jsme přidali hlavičku `no-cache`, takže úplně obešla své lokální mezipaměti. Potom ale nemá na výběr a musí odeslat požadavek na data do sítě.

[3] Tato odpověď *nebyla* generovaná z naší lokální mezipaměti. To samozřejmě víme, protože jsme viděli ladicí informaci týkající se odcházejícího požadavku. Ale je dobré, že si to můžeme ověřit v programu.

[4] Požadavek byl úspěšný. Opět jsme ze vzdáleného serveru stáhli celý publikovaný obsah (feed). Server samozřejmě poslal zpět s požadovanými daty (feed) i celou sadu `HTTP` hlaviček. Jsou mezi nimi i hlavičky pro mezipaměť, které `httplib2` použije pro aktualizaci své lokální mezipaměti v naději, že se při *příštím* požadavku na stejná data bude moci vyhnout přístupu na síť. Návrh používání mezipamětí v `HTTP` je zcela podřízen maximalizaci úspěšnosti mezipamětí (cache hit) a minimalizaci přístupu k síti. I když jsme tentokrát mezipaměti obešli, vzdálený server by opravdu ocenil, kdybychom si výsledek do mezipaměti uložili — s ohledem na příští možný dotaz.

14.5.3. Jak `httplib2` zachází s hlavičkami `Last-Modified` a `ETag`

Hlavičky mezipaměti `Cache-Control` a `Expires` se nazývají *indikátory čerstvosti* (freshness indicators). Říkají mezipamětem jasným způsobem, že se do vypršení platnosti obsahu mezipaměti můžeme zcela vyhnout přístupu k síti. Přesně takové chování jsme viděli v předchozí podkapitole: pokud je indikována čerstvost, `httplib2` při vrácení dat z mezipaměti *negeneruje ani bajt síťové aktivity* (pokud ovšem explicitně nepředepíšeme obejití mezipaměti).

Ale jak to bude vypadat v případě, kdy se data *mohla* změnit, ale přitom se nezměnila? Pro tento účel HTTP definuje hlavičky Last-Modified a Etag. Těmto hlavičkám se říká *validátory*. Pokud už lokální mezipaměť není čerstvá, může klient s dalším dotazem zaslat validátory, aby si ověřil, zda se data skutečně změnila. Pokud se data nezměnila, server pošle zpět stavový kód 304 a *žádná data*. Takže tu sice stále dochází ke vzájemné komunikaci po síti, ale výsledkem je stahování menšího množství bajtů.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> response, content = h.request('http://diveintopython3.org/')          [1]
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
>>> print(dict(response.items()))                                          [2]
{'-content-encoding': 'gzip',
 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 'etag': '"7f806d-1a01-9fb97900"',
 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
 'server': 'Apache',
 'status': '200',
 'vary': 'Accept-Encoding,User-Agent'}
>>> len(content)                                                           [3]
6657
```

[1] Místo publikovaného obsahu (feed) budeme tentokrát stahovat domácí stránku webového místa (home page), která je v HTML. Protože tuto stránku požadujeme úplně poprvé, nemůže httplib2 s požadavkem nic moc udělat a odešle s ním minimum hlaviček.

[2] Odpověď obsahuje velké množství HTTP hlaviček… ale žádné informace pro mezipaměť. Ale obsahuje jak hlavičku ETag, tak hlavičku Last-Modified.

[3] V době vytváření příkladu měla stránka 6657 bajtů. Od té doby už se pravděpodobně změnila, ale tím se nebudeme zatěžovat.

```
# pokračování z předchozího příkladu
>>> response, content = h.request('http://diveintopython3.org/')          [1]
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
if-none-match: "7f806d-1a01-9fb97900"                                     [2]
if-modified-since: Tue, 02 Jun 2009 02:51:48 GMT                          [3]
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 304 Not Modified'                                        [4]
>>> response.fromcache                                                    [5]
True
>>> response.status                                                       [6]
200
>>> response.dict['status']                                               [7]
'304'
>>> len(content)                                                          [8]
6657
```

[1] O stejnou stránku jsme požádali znovu, prostřednictvím stejného objektu třídy `Http` (a se stejnou lokální mezipamětí).

[2] `httplib2` pošle serveru zpět validátor `ETag` jako obsah hlavičky `If-None-Match`.

[3] `httplib2` pošle zpět serveru také validátor `Last-Modified` jako hodnotu hlavičky `If-Modified-Since`.

[4] Server se podívá na zaslané validátory, podívá se na požadovanou stránku a zjistí, že se stránka od posledního požadavku nezměnila. Proto pošle zpět stavový kód `304` a *žádná data*.

[4] A zpět ke klientovi. `httplib2` obdrží stavový kód `304` a načte obsah stránky ze své mezipaměti.

[5] Tohle může být trošku matoucí. Ve skutečnosti tu máme dva stavové kódy — `304` (který vrátil server teď a který způsobil, že `httplib2` použije svou mezipaměť) a `200` (který vrátil server minule a který je spolu s daty uložen v mezipaměti pro `httplib2`). `response.status` vrací stavový kód odpovědi z mezipaměti.

[6] Pokud chceme zjistit surový stavový kód vrácený serverem, můžeme jej zjistit nahlédnutím do `response.dict`, což je slovník aktuálních hlaviček vrácených serverem.

[7] Ať je to jakkoliv, data opět získáte v proměnné `content`. Obecně vzato, nepotřebujeme vědět, proč byl požadavek obsloužen z mezipaměti. (Dokonce nás nemusí vůbec zajímat, že byl obsloužen z mezipaměti. To je v pořádku. Knihovna `httplib2` je dost chytrá na to, abychom si mohli hrát na hlupáky.) V tomto okamžiku už metoda `request()` vrátila řízení volajícímu kódu. `httplib2` už aktualizovala svou mezipaměť a vrátila nám data.

14.5.4. Jak http2lib pracuje s kompresí

"We have both kinds of music, country AND western." (Máme oba druhy hudby, country i western.)

HTTP podporuje několik typů komprese. Dva nejpoužívanější typy jsou gzip a deflate. httplib2 podporuje oba.

```
>>> response, content = h.request('http://diveintopython3.org/')
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip                                    [1]
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
>>> print(dict(response.items()))
{'-content-encoding': 'gzip',                                     [2]
 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 'etag': '"7f806d-1a01-9fb97900"',
 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
 'server': 'Apache',
 'status': '304',
 'vary': 'Accept-Encoding,User-Agent'}
```

[1] Pokaždé když httplib2 odešle požadavek, vloží do něj hlavičku Accept-Encoding, kterou serveru oznámí, že zvládá jak kompresi deflate, tak gzip.

[2] V tomto případě server odpověděl daty komprimovanými algoritmem gzip. V tomto okamžiku metoda request() vrací řízení, httplib2 dekomprimovala (rozbalila) tělo odpovědi a umístila je do proměnné content. Pokud jste zvědaví, jestli odpověď přišla komprimovaná, můžete zkontrolovat response['-content-encoding']. Ale jinak si s tím nemusíte dělat starosti.

14.5.5. Jak httplib2 řeší přesměrování

HTTP definuje dva druhy přesměrování: dočasné a trvalé. U dočasných přesměrování se nedělá nic zvláštního až na to, že se mají následovat (follow), což httplib2 provede automaticky.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> response, content = h.request('http://diveintopython3.org/examples/feed-302.xml')  [1]
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-302.xml HTTP/1.1                                            [2]
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 302 Found'                                                            [3]
send: b'GET /examples/feed.xml HTTP/1.1                                                [4]
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
```

[1] Na tomto URL není žádný publikovaný obsah. Nastavil jsem svůj server, aby signalizoval dočasné přesměrování na správnou adresu.

[2] Tady je náš požadavek.

[3] A tady je odpověď: `302 Found`. I když se to zde nezobrazuje, odpověď obsahuje také hlavičku `Location`, která ukazuje na skutečné URL.

[4] `httplib2` se ihned otočí a „následuje“ přesměrování vydáním dalšího požadavku na URL, které je uvedeno v hlavičce `Location: http://diveintopython3.org/examples/feed.xml`

„Následování“ přesměrování není nic jiného, než co ukazuje tento příklad. `httplib2` pošle požadavek pro URL, které jsme požadovali. Server odvětí odpovědí, která říká: „Ne ne. Místo toho se podívejte támhle.“ `httplib2` odešle další požadavek pro nové URL.

```
# pokračování z předchozího příkladu
>>> response                                                                    [1]
{'status': '200',
 'content-length': '3070',
 'content-location': 'http://diveintopython3.org/examples/feed.xml',          [2]
 'accept-ranges': 'bytes',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Wed, 03 Jun 2009 02:20:15 GMT',
 'connection': 'close',
 '-content-encoding': 'gzip',                                                 [3]
 'etag': '"bfe-4cbbf5c0"',
 'cache-control': 'max-age=86400',                                            [4]
 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'application/xml'}
```

[1] Odpověď (response), kterou jste obdrželi z jediného volání metody `request()`, je odpovědí z konečného URL.
[2] `httplib2` přidá konečné URL do slovníku `response` jako `content-location`. Nejde o hlavičku, která by přišla ze serveru. Je to záležitost specifická pro `httplib2`.
[3] Jen abych nezapomněl, tento feed je komprimovaný.
[4] A uchovatelný v mezipaměti (cacheable). (To je důležité — jak uvidíme za minutku.)

Slovník `response`, který se nám vrátí, poskytuje informace o *konečném* URL. A co když chceme informace o přechodných URL, tedy o těch, která byla přesměrována na konečné URL? `httplib2` nám umožní i to.

```
# pokračování z předchozího příkladu
>>> response.previous                                                    [1]
{'status': '302',
 'content-length': '228',
 'content-location': 'http://diveintopython3.org/examples/feed-302.xml',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'server': 'Apache',
 'connection': 'close',
 'location': 'http://diveintopython3.org/examples/feed.xml',
 'cache-control': 'max-age=86400',
 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'text/html; charset=iso-8859-1'}
>>> type(response)                                                       [2]
<class 'httplib2.Response'>
>>> type(response.previous)
<class 'httplib2.Response'>
>>> response.previous.previous                                           [3]
>>>
```

[1] Atribut `response.previous` uchovává referenci na předchozí objekt odpovědi, který `httplib2` následovala, aby získala současný objekt odpovědi.
[2] Jak response, tak `response.previous` jsou objekty třídy `httplib2.Response`.
[3] To znamená, že můžeme řetězec přesměrování sledovat zpětně ještě dál kontrolou `response.previous.previous`. (Scénář: jedno URL je přesměrováno na druhé URL, které je přesměrováno na třetí URL. To se opravdu může stát!) V tomto případě už jsme dosáhli začátku řetězce přesměrování, takže atribut má hodnotu `None`.

Co se stane, když si vyžádáme stejné URL znovu?

```
# pokračování z předchozího příkladu
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed-302.xml')  [1]
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-302.xml HTTP/1.1                                              [2]
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 302 Found'                                                              [3]
>>> content2 == content                                                                  [4]
True
```

[1] Stejné URL, stejný objekt třídy `httplib2.Http` (a tím pádem stejná mezipaměť).

[2] Odpověď `302` nebyla v mezipaměti, takže `httplib2` pošle pro stejné URL další požadavek.

[3] A ještě jednou, server odpovídá kódem `302`. Ale všimněte si, co se nestalo: chybí druhý dotaz na konečné URL, `http://diveintopython3.org/examples/feed.xml`. Tato odpověď byla v mezipaměti (vzpomeňte si na hlavičku `Cache-Control`, kterou jsme viděli v předchozím příkladu). Jakmile `httplib2` obdržela kód `302 Found`, *zkontrolovala si před vydáním dalšího požadavku obsah mezipaměti*. Mezipaměť obsahovala čerstvou kopii `http://diveintopython3.org/examples/feed.xml`, takže nebylo nutné žádat o data znovu.

[4] V tomto okamžiku dochází k návratu z metody `request()`. Přečetla data publikovaného obsahu (feed) z mezipaměti a vrátila je. Jde samozřejmě o stejná data, která jsme obdrželi minule.

Jinými slovy, při dočasném přesměrování nemusíme dělat nic zvláštního. `httplib2` je bude následovat automaticky. Skutečnost, že je jedno URL přesměrováno na jiné, nemá na `httplib2` žádné dopady z hlediska podpory komprese, použití mezipaměti, ETagů nebo jakýchkoliv jiných rysů HTTP.

Trvalá přesměrování jsou ve své jednoduchosti podobná.

```
# pokračování z předchozího příkladu
>>> response, content = h.request('http://diveintopython3.org/examples/feed-301.xml')  [1]
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-301.xml HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 301 Moved Permanently'                                                  [2]
>>> response.fromcache                                                                  [3]
True
```

[1] Ještě jednou. Toto URL ve skutečnosti neexistuje. Nastavil jsem svůj server, aby produkoval trvalé přesměrování na `http://diveintopython3.org/examples/feed.xml`.

[2] A tady to máme: stavový kód `301`. Ale znovu si všimněte, co se *nestalo*: neobjevil se žádný požadavek na přesměrované `URL`. Proč ne? Protože už se nachází v lokální mezipaměti.

[3] `httplib2` „následovala“ přesměrování přímo do své mezipaměti.

Ale počkejte! Ono je toho ještě víc!

```
# pokračování z předchozího příkladu
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed-301.xml')  [1]
>>> response2.fromcache                                                                  [2]
True
>>> content2 == content                                                                  [3]
True
```

[1] Tady je ten rozdíl mezi dočasným a trvalým přesměrováním: jakmile jednou `httplib2` následuje trvalé přesměrování, všechny další požadavky se stejným `URL` budou transparentně přepsány na cílové `URL` *aniž se kvůli originálnímu* `URL` *komunikuje po síti*. Připomeňme si, že ladicí režim je pořád zapnutý. Přesto nevidíme vůbec žádný výstup síťové aktivity.

[2] Ano, tato odpověď byla vytažena z lokální mezipaměti.

[3] Ano, dostali jsme celý publikovaný obsah (z mezipaměti).

`HTTP`. Funguje.

14.6. Za hranicemi HTTP GET

Webové služby nad `HTTP` se neomezují jen na požadavky typu `GET`. Co kdybychom chtěli vytvořit něco nového? Kdykoliv přidáte komentář do diskusního fóra, aktualizujete weblog, upravujete svůj stav na mikroblogové službě, jakou je **Twitter** nebo **Identi.ca**, používáte pravděpodobně `HTTP POST`.

Jak Twitter, tak Identi.ca nabízejí pro zveřejňování a aktualizaci vašeho stavu, popsaného 140 nebo méně znaky, jednoduché rozhraní založené na `HTTP`. Podívejme se na dokumentaci aplikačního rozhraní pro aktualizaci vašeho stavu v systému Identi.ca.

Jak to funguje? Když chceme na Identi.ca zveřejnit novou zprávu, musíme zaslat požadavek typu `HTTP POST` na `http://identi.ca/api/statuses/update.format`. (Část format nepatří k `URL`. Nahrazuje se datovým formátem, v jakém nám má server vrátit odpověď na náš požadavek. Takže pokud požadujeme odpověď v `XML`, musíme zaslat požadavek na `https://identi.ca/api/statuses/update.xml`.) Požadavek musí obsahovat parametr nazvaný `status`, který obsahuje text pro aktualizaci našeho stavu. A požadavek musí být autentizován.

Identi.ca REST API **Metoda: statuses/update**
Aktualizuje stav autentizovaného uživatele. Vyžaduje parametr status, popsaný níže.
Požadavek musí být typu POST.

URL
    https://identi.ca/api/statuses/update.format
Formáty
    xml, json, rss, atom
HTTP metod(y)
    POST
Vyžaduje autentizaci
    ano
Parametry
    status. Povinný. Text aktualizace vašeho stavu. Kódované URL podle potřeby.

Autentizován? Jistě. Když chceme na Identi.ca aktualizovat svůj stav, musíme prokázat svou totožnost. Identi.ca není jako wiki. Svůj vlastní stav můžeme aktualizovat jen my. Pro účel bezpečné a snadno použitelné autentizace používá Identi.ca HTTP **Basic Authentication** (základní autentizaci; známou také jako **RFC 2617**) přes SSL. `httplib2` podporuje jak SSL, tak HTTP Basic Authentication, takže tahle část bude snadná.

Požadavek POST se od požadavku GET liší, protože nese *náklad*. Nákladem jsou data, která chceme poslat na server. Částí dat, kterou toto aplikační rozhraní metody *vyžaduje*, je `status` (stav) a měl by mít podobu *kódovaného* URL. Je to velmi jednoduchý serializační formát. Vstupem je množina dvojic klíč-hodnota (tj. slovník) a výsledkem je řetězec.

```
>>> from urllib.parse import urlencode                          [1]
>>> data = {'status': 'Test update from Python 3'}              [2]
>>> urlencode(data)                                             [3]
'status=Test+update+from+Python+3'
```

[1] V Pythonu pro zakódování slovníku do podoby URL najdeme pomocnou funkci: `urllib.parse.urlencode()`.
[2] Aplikační rozhraní systému Identi.ca očekává zhruba takovýto slovník. Obsahuje jeden klíč, `status`, jehož hodnotou je text jedné aktualizace stavu.
[3] A takto vypadá řetězec kódovaného URL. To je *náklad*, který bude požadavkem HTTP POST odeslán „po drátě“ na server s aplikačním rozhraním Identi.ca.

```
>>> from urllib.parse import urlencode
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> data = {'status': 'Test update from Python 3'}
>>> h.add_credentials('diveintomark', 'MY_SECRET_PASSWORD', 'identi.ca')          [1]
>>> resp, content = h.request('https://identi.ca/api/statuses/update.xml',
...     'POST',                                                                   [2]
...     urlencode(data),                                                          [3]
...     headers={'Content-Type': 'application/x-www-form-urlencoded'})            [4]
```

[1] Tímto způsobem `httplib2` pracuje s autentizací. Jméno a heslo uložíme metodou `add_credentials()`. Když se `httplib2` pokusí o vydání požadavku, server odpoví stavovým kódem `401 Unauthorized` (neautorizováno) a připojí seznam autentizačních metod, které podporuje (v hlavičce `WWW-Authenticate`). `httplib2` automaticky vytvoří hlavičku `Authorization` a pošle požadavek s `URL` znovu.

[2] Druhý parametr uvádí typ `HTTP` požadavku. V tomto případě je to `POST`.

[3] Třetím parametrem je *náklad*, který se serveru posílá. Posíláme slovník se stavovou zprávou zakódovaný do podoby `URL`.

[4] Nakonec musíme serveru říct, že náklad má podobu dat zakódovaných do podoby `URL`.

> Třetím parametrem metody `add_credentials()` je doména, ve které osobní údaje platí. Měli byste ji vždy uvádět! Pokud doménu vynecháte a později znovu použijete objekt třídy `httplib2.Http` pro jiné autentizované místo, mohla by `httplib2` způsobit únik jména a hesla z jednoho místa na druhé místo (site).

A o čem zpívají dráty:

```
# pokračování z předchozího příkladu
send: b'POST /api/statuses/update.xml HTTP/1.1
Host: identi.ca
Accept-Encoding: identity
Content-Length: 32
content-type: application/x-www-form-urlencoded
user-agent: Python-httplib2/$Rev: 259 $

status=Test+update+from+Python+3'
reply: 'HTTP/1.1 401 Unauthorized'                                  [1]
send: b'POST /api/statuses/update.xml HTTP/1.1                      [2]
Host: identi.ca
Accept-Encoding: identity
Content-Length: 32
content-type: application/x-www-form-urlencoded
authorization: Basic SECRET_HASH_CONSTRUCTED_BY_HTTPLIB2            [3]
user-agent: Python-httplib2/$Rev: 259 $
```

```
status=Test+update+from+Python+3'
reply: 'HTTP/1.1 200 OK'                                    [4]
```

[1] Po prvním požadavku odpoví server stavovým kódem `401 Unauthorized`. `httplib2` nikdy neposílá autentizační hlavičky, pokud si o ně server explicitně neřekne. Server si o ně říká tímto způsobem.

[2] `httplib2` okamžitě zareaguje opakovaným odesláním požadavku se stejným `URL`.

[3] Tentokrát obsahuje jméno a heslo, která jsme přidali metodou `add_credentials()`.

[4] Funguje to!

A co vlastně server posílá po úspěšném požadavku zpět? To zcela závisí na aplikačním rozhraní příslušné webové služby. V některých protokolech (jako například Atom Publishing Protocol) posílá server zpět stavový kód `201 Created` spolu s umístěním nově vytvořeného zdroje (resource) v hlavičce `Location`. Identi.ca posílá zpět `200 OK` a `XML` dokument, který obsahuje informace o nově vytvořeném zdroji.

```
# pokračování z předchozího příkladu
>>> print(content.decode('utf-8'))                           [1]
<?xml version="1.0" encoding="UTF-8"?>
<status>
 <text>Test update from Python 3</text>                     [2]
 <truncated>false</truncated>
 <created_at>Wed Jun 10 03:53:46 +0000 2009</created_at>
 <in_reply_to_status_id></in_reply_to_status_id>
 <source>api</source>
 <id>5131472</id>                                           [3]
 <in_reply_to_user_id></in_reply_to_user_id>
 <in_reply_to_screen_name></in_reply_to_screen_name>
 <favorited>false</favorited>
 <user>
  <id>3212</id>
  <name>Mark Pilgrim</name>
  <screen_name>diveintomark</screen_name>
  <location>27502, US</location>
  <description>tech writer, husband, father</description>
  <profile_image_url>http://avatar.identi.ca/3212-48-20081216000626.png</profile_image_url>
  <url>http://diveintomark.org/</url>
  <protected>false</protected>
  <followers_count>329</followers_count>
  <profile_background_color></profile_background_color>
  <profile_text_color></profile_text_color>
  <profile_link_color></profile_link_color>
  <profile_sidebar_fill_color></profile_sidebar_fill_color>
  <profile_sidebar_border_color></profile_sidebar_border_color>
  <friends_count>2</friends_count>
```

```
  <created_at>Wed Jul 02 22:03:58 +0000 2008</created_at>
  <favourites_count>30768</favourites_count>
  <utc_offset>0</utc_offset>
  <time_zone>UTC</time_zone>
  <profile_background_image_url></profile_background_image_url>
  <profile_background_tile>false</profile_background_tile>
  <statuses_count>122</statuses_count>
  <following>false</following>
  <notifications>false</notifications>
</user>
</status>
```

[1] Připomeňme si, že data vracená `httplib2` jsou vždy bajty a ne řetězce. Abychom je mohli převést na řetězec, musíme je dekódovat s použitím příslušného znakového kódování. Aplikační rozhraní systému Identi.ca vždy vrací výsledky v `UTF-8`. Takže tato část je snadná.

[2] Zde je text stavové zprávy, kterou jsme právě zveřejnili.

[3] Toto je unikátní identifikátor nové stavové zprávy. Identi.ca jej používá pro konstrukci `URL`, které se dá použít pro zobrazení zprávy na webu.

A tady ji máme:

14.7. Za hranicemi HTTP POST

`HTTP` se neomezuje jen na `GET` a `POST`. Nepochybně jde o nejběžnější typy dotazů, obzvlášť ze strany webových prohlížečů. Ale rozhraní webových služeb může jít za hranice `GET` a `POST` — a knihovna `httplib2` je na to připravená.

```
# pokračování z předchozího příkladu
>>> from xml.etree import ElementTree as etree
>>> tree = etree.fromstring(content)                                    [1]
>>> status_id = tree.findtext('id')                                     [2]
>>> status_id
'5131472'
>>> url = 'https://identi.ca/api/statuses/destroy/{0}.xml'.format(status_id)   [3]
>>> resp, deleted_content = h.request(url, 'DELETE')                    [4]
```

[1] Sever vrátil XML, že ano? A my už víme, jak XML zpracovat.

[2] Metoda `findtext()` najde první objekt odpovídající zadanému výrazu a extrahuje jeho textový obsah. V tomto případě hledáme element `<id>`.

[3] Z textového obsahu elementu `<id>` můžeme zkonstruovat URL pro vymazání stavové zprávy, kterou jsme zrovna zveřejnili.

[4] Zprávu vymažeme tím, že pro zmíněné URL vytvoříme požadavek `HTTP DELETE`.

Po drátech běhá následující:

```
send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1            [1]
Host: identi.ca
Accept-Encoding: identity
user-agent: Python-httplib2/$Rev: 259 $

'
reply: 'HTTP/1.1 401 Unauthorized'                                   [2]
send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1            [3]
Host: identi.ca
Accept-Encoding: identity
authorization: Basic SECRET_HASH_CONSTRUCTED_BY_HTTPLIB2             [4]
user-agent: Python-httplib2/$Rev: 259 $

'
reply: 'HTTP/1.1 200 OK'                                             [5]
>>> resp.status
200
```

[1] „Odstraň tuto stavovou zprávu.“

[2] „Je mi líto, Dave [dejve]. Obávám se, že to nemohu udělat.“

[3] „Neautorizováno ❓ Hmmm. Odstraň tu stavovou zprávu, *prosím*…

[4] …a tady je mé jméno a heslo.“

[5] „Považuj to za hotovou věc!“

Puf a je to pryč.

14.8. Přečtěte si

`httplib2`:

- **Stránka projektu `httplib2`** (anglicky)
  (http://code.google.com/p/httplib2/)
- Další příklady kódu využívajícího **`httplib2`** (anglicky)
  (http://code.google.com/p/httplib2/wiki/ExamplesPython3)
- **Doing `HTTP` Caching Right**: **Introducing `httplib2`** (anglický článek)
  (www.xml.com/pub/a/2006/02/01/doing-http-caching-right-introducing-httplib2.html)
- **`httplib2`: `HTTP` Persistence and Authentication** (anglický článek)
  (www.xml.com/pub/a/2006/03/29/httplib2-http-persistence-and-authentication.html)

Práce `HTTP` s mezipamětí:

- **`HTTP` Tutorial** — napsal Mark Nottingham
  (www.mnot.net/cache_docs/)
- **How to control caching with `HTTP` headers** (anglický článek o **Google Doctype**)
  (http://code.google.com/p/doctype/wiki/ArticleHttpCaching)

`RFC`:

- **RFC 2616: `HTTP`**
  (www.ietf.org/rfc/rfc2616.txt)
- **RFC 2617: `HTTP` Basic Authentication**
  (www.ietf.org/rfc/rfc2617.txt)
- **RFC 1951: `DEFLATE` compression**
  (www.ietf.org/rfc/rfc1951.txt)
- **RFC 1952: `GZIP` compression**
  (www.ietf.org/rfc/rfc1952.txt)

“Words, words. They’re all we have to go on.”
(Slova, slova. Jsou vším, čeho se musíme držet.)

— *Rosencrantz a Guildenstern jsou mrtvi*

15. Případová studie: Přepis chardet pro Python 3

15. Případová studie: Přepis chardet pro Python 3 — 329
15.1. Ponořme se — **331**
15.2. Co se rozumí autodetekcí znakového kódování? — **331**
15.2.1. Není to náhodou neproveditelné? — **331**
15.2.2. Existuje takový algoritmus? — **332**
15.3. Úvod do modulu `chardet` — **332**
15.3.1. `UTF-N` s `BOM` — **332**
15.3.2. Kódování escape sekvencemi — **333**
15.3.3. Vícebajtová kódování — **333**
15.3.4. Jednobajtová kódování — **334**
15.3.5. `windows-1252` — **334**
15.4. Spouštíme `2to3` — **335**
15.5. Krátká odbočka k vícesouborovým modulům — **338**
15.6. Opravme, co `2to3` neumí — **340**
15.6.1. False je syntaktická chyba — **340**
15.6.2. Nenalezen modul `constants` — **341**
15.6.3. Jméno `'file'` není definováno — **342**
15.6.4. Řetězcový vzorek nelze použít pro bajtové objekty — **343**
15.6.5. Objekt typu `'bytes'` nelze implicitně převést na `str` — **345**
15.6.6. Nepodporované typy operandů pro `+: 'int'` a `'bytes'` — **348**
15.6.7. Funkce `ord()` očekávala řetězec o délce 1, ale byl nalezen `int` — **350**
15.6.8. Neuspořádatelné datové typy: `int() >= str()` — **352**
15.6.9. Globální jméno `'reduce'` není definováno — **355**
15.7. Shrnutí — **357**

15.1. Ponořme se
15.2. Co se rozumí autodetekcí znakového kódování?

15.1. Ponořme se

Otázka: Co je příčnou č. 1 vedoucí ke zmatenému textu na webu, ve vaší poštovní schránce a ve všech dokumentech, které kdy byly napsány, napříč všemi počítačovými systémy? Je to kódování znaků. V kapitole Řetězce jsme se bavili o historii kódování znaků a o vytvoření Unicode — „jedno kódování vládne všem“. Moc bych si přál, kdybych se na webových stránkách nikdy víc nesetkával se zmatenými znaky, protože by všechny systémy pro vytváření textu ukládaly přesnou informaci o kódování a protože by byly všechny přenosové protokoly připravené na používání Unicode a každý systém pro zpracování textu by při konverzi mezi kódováními zachovával perfektní věrnost.

Rád bych taky poníka.

Unicode poníka.

Kdyby to tak byl Uniponík.

Budu si muset osedlat autodetekci znakového kódování.

15.2. Co se rozumí autodetekcí znakového kódování?

Rozumí se tím to, že vezmeme posloupnost bajtů v neznámém znakovém kódování a pokoušíme se kódování zjistit, abychom si text mohli přečíst. Podobá se to lámání kódu v situaci, kdy nemáme dešifrovací klíč.

15.2.1. Není to náhodou neproveditelné?

Z obecného pohledu to opravdu je nemožné. Ale některá kódování jsou optimalizována pro určité jazyky a jazyky nejsou náhodné. Některé posloupnosti znaků se objevují neustále, zatímco jiné posloupnosti nedávají žádný smysl. Když osoba plynně ovládající angličtinu otevře noviny a najde „txzqJv 2!dasd0a QqdKjvz“, okamžitě pozná, že nejde o angličtinu (i když se text skládá pouze z písmen, která se v angličtině používají). Na základě studia velkého množství „typického“ textu může počítačový algoritmus simulovat zmíněný druh plynné znalosti a může provést kvalifikovaný odhad týkající se jazyka textu.

Jinými slovy, detekce kódování je ve skutečnosti detekcí jazyka, která se kombinuje se znalostí tendence jazyka používat určité znakové kódování.

15.2.2. Existuje takový algoritmus?

Jak se ukazuje, tak ano. Všechny nejpoužívanější prohlížeče mají autodetekci kódování zabudovanou, protože web je plný stránek, které neobsahují vůbec žádnou informaci o kódování. **Mozilla Firefox** obsahuje knihovnu pro autodetekci kódování, která je open source. Knihovnu jsem přenesl do **Pythonu 2** a modul jsem nazval `chardet`. V této kapitole vás krok za krokem provedu procesem přepisování modulu `chardet` z Pythonu 2 pro Python 3.

15.3. Úvod do modulu `chardet`

Detekce kódování je ve skutečnosti v závěsu za detekcí jazyka.

Než se do přepisu kódu pustíme, bylo by dobré, kdybyste rozuměli, jak funguje! Toto je stručná příručka pro usnadnění orientace ve vlastním kódu. Knihovna `chardet` je příliš velká na to, abych její kód vložil do textu této knihy. Ale můžete si ji stáhnout z `chardet.feedparser.org`.

Hlavním vstupním bodem detekčního algoritmu je `universaldetector.py`. Obsahuje jednu třídu, `UniversalDetector`. (Možná jste mysleli, že hlavním vstupním bodem je funkce detect z `chardet/__init__.py`. To je ale jen funkce pro zvýšení pohodlí, která vytvoří objekt třídy `UniversalDetector`, zavolá jej a vrátí jeho výsledek.)

`UniversalDetector` zvládá pět kategorií kódování:

1. `UTF-N` s Byte Order Mark (`BOM`; znak pro určení pořadí bajtů). Zahrnuje `UTF-8`, obě varianty `UTF-16` (Big-Endian a Little-Endian) a všechny 4 varianty pořadí bajtů `UTF-32`.
2. Kódování s únikovými znaky (escape encodings), která jsou zcela kompatibilní se 7bitovým `ASCII`. Znaky spadající mimo `ASCII` začínají únikovými sekvencemi (escape sequence). Příklady: `ISO-2022-JP` (japonština) a `HZ-GB-2312` (čínština).
3. Vícebajtová kódování, ve kterých je každý znak reprezentován proměnným počtem bajtů. Příklady: `BIG5` (čínština), `SHIFT_JIS` (japonština), `EUC-KR` (korejština) a `UTF-8` bez `BOM`.
4. Jednobajtová kódování, ve kterých je každý znak reprezentován jedním bajtem. Příklady: `KOI8-R` (ruština), `WINDOWS-1255` (hebrejština) a `TIS-620` (thajština).
5. `WINDOWS-1252`, která používají především (v Microsoft Windows) střední manažeři, kteří nerozpoznají znakové kódování od díry v zemi.

15.3.1. `UTF-N` s `BOM`

Pokud text začíná značkou `BOM`, můžeme rozumně předpokládat, že je zakódován v `UTF-8`, `UTF-16` nebo `UTF-32`. (Značka bom nám přesně řekne, o které kódování jde. Byla pro tento účel navržena.) To se děje přímo v `UniversalDetectoru`, který vrátí výsledek okamžitě, bez dalšího zpracovávání textu.

15.3.2. Kódování escape sekvencemi

Pokud text obsahuje rozpoznatelné posloupnosti s únikovými znaky (escape sequence), může to být příznakem použití kódování, kterému se v angličtině říká escaped encoding. `UniversalDetector` vytvoří `EscCharSetProber` (je definován v `escprober.py`) a přivede do něj text.

`EscCharSetProber` vytvoří sadu konečných automatů, které vycházejí z modelů pro `HZ-GB-2312`, `ISO-2022-CN`, `ISO-2022-JP` a `ISO-2022-KR` (jsou definovány v `escsm.py`). `EscCharSetProber` přivádí text do každého z těchto konečných automatů — bajt po bajtu. Pokud některý z konečných automatů skončí s jednoznačnou identifikací kódování, vrátí `EscCharSetProber` okamžitě pozitivní výsledek objektu třídy `UniversalDetector`, který jej vrátí volajícímu. Pokud kterýkoliv z konečných automatů narazí na nepřípustnou posloupnost, je vyřazen a další zpracování pokračuje jen s ostatními konečnými automaty.

15.3.3. Vícebajtová kódování

Za předpokladu, že není použita značka `BOM`, `UniversalDetector` zkontroluje, zda text obsahuje nějaké znaky s nastaveným osmým bitem. Pokud tomu tak je, vytvoří sérii „detekčních zařízení“ (prober) pro rozpoznání vícebajtových kódování, jednobajtových kódování a nakonec, jako poslední možnost, pro `WINDOWS-1252`.

Detekční objekt pro vícebajtová kódování, `MBCSGroupProber` (třída je definována v `mbcsgroupprober.py`), je ve skutečnosti jen obálkou. Ovládá ostatní detekční objekty, po jednom pro každé vícebajtové kódování: `BIG5`, `GB2312`, `EUC-TW`, `EUC-KR`, `EUC-JP`, `SHIFT_JIS` a `UTF-8`. `MBCSGroupProber` směřuje text do každého z těchto specializovaných detekčních objektů a kontroluje výsledky. Pokud nějaký detekční objekt hlásí, že nalezl nepřípustnou posloupnost bajtů, je vyřazen z dalšího zpracování (takže například libovolné následné volání metody `UniversalDetector.feed()` vyřazený detekční objekt přeskočí). Pokud detekční objekt hlásí, že si je poměrně jistý rozpoznáním kódování, oznámí `MBCSGroupProber` tento pozitivní výsledek objektu `UniversalDetector`, který oznámí výsledek volajícímu.

Většina z detekčních objektů pro vícebajtová kódování je odvozena z `MultiByteCharSetProber` (definována v `mbcharsetprober.py`) a jednoduše se navěsí na příslušný konečný automat a analyzátor rozložení. Zbytek práce nechá na `MultiByteCharSetProber`. `MultiByteCharSetProber` prohání text přes konečné automaty specializované na jednotlivá kódování — bajt po bajtu. Vyhledává posloupnosti bajtů, které by indikovaly průkazné pozitivní nebo negativní výsledky. `MultiByteCharSetProber` současně posílá text do analyzátoru rozložení, který je specifický pro každé kódování.

Analyzátory rozložení (jsou definovány v `chardistribution.py`) používají jazykově specifické modely nejčastěji se vyskytujících znaků. Jakmile `MultiByteCharSetProber` předá analyzátorům rozložení dostatečný objem textu, vypočítá ohodnocení spolehlivosti, které je založeno na počtu často používaných znaků, na celkovém počtu znaků a na jazykově závislém rozložení. Pokud je spolehlivost dostatečně velká, vrátí `MultiByteCharSetProber` výsledek do `MBCSGroupProber`, který jej vrátí do `UniversalDetectoru`, který jej vrátí volajícímu.

Případ japonštiny je obtížnější. Analýza rozložení podle jednotlivých znaků nevede vždy k rozlišení EUC-JP a SHIFT_JIS, takže SJISProber (definován v sjisprober.py) používá také dvojznakovou analýzu rozložení. SJISContextAnalysis a EUCJPContextAnalysis (definice se v obou případech nacházejí v jpcntx.py a obě třídy dědí ze společné třídy JapaneseContextAnalysis) v textu kontrolují frekvenci výskytů slabičných znaků hiragana. Jakmile bylo zpracováno dostatečné množství textu, vracejí úroveň spolehlivosti do SJISProber, který zkontroluje oba analyzátory a vrátí výsledek s vyšší úrovní spolehlivosti do MBCSGroupProber.

15.3.4. Jednobajtová kódování

Detekční objekt pro jednobajtové kódování, SBCSGroupProber (třída je definována v sbcsgroupprober.py) je rovněž obálkou, která ovládá skupinu jiných detekčních objektů — jeden pro každou kombinaci jednobajtového kódování a jazyka: WINDOWS-1251, KOI8-R, ISO-8859-5, MacCyrillic, IBM855 a IBM866 (ruština); ISO-8859-7 a WINDOWS-1253 (řečtina); ISO-8859-5 a WINDOWS-1251 (bulharština); ISO-8859-2 a WINDOWS-1250 (čeština, maďarština, slovenština a další); TIS-620 (thajština); WINDOWS-1255 a ISO-8859-8 (hebrejština).

Vážně, kde je můj Unicode poník?

SBCSGroupProber předává text do každého z těchto detekčních objektů a kontroluje výsledky. Všechny tyto detekční objekty jsou implementovány v jedné třídě, SingleByteCharSetProber (definována v sbcharsetprober.py), která prostřednictvím argumentu přebírá jazykový model. Jazykový model definuje, jak často se v typickém textu vyskytují dvojznakové posloupnosti. SingleByteCharSetProber zpracovává text a zjišťuje nejčastěji se vyskytující dvojznakové posloupnosti. Jakmile byl zpracován dostatečný objem textu, vypočítá úroveň spolehlivosti, která je založena na počtu často se vyskytujících posloupností, na celkovém počtu znaků a na jazykově závislém rozložení.

Hebrejština se řeší jako zvláštní případ. Pokud se text na základě analýzy rozložení dvojznakových posloupností jeví jako hebrejština, snaží se HebrewProber (třída definována v hebrewprober.py) rozlišit mezi vizuální hebrejštinou (kdy je text uložen ve skutečnosti „pozpátku“ řádek po řádku a poté je zobrazen „normálně“, takže může být čten zprava doleva) a logickou hebrejštinou (kdy je zdrojový text uložen v pořadí čtení a klientský program ho vykresluje zprava doleva). Protože se některé znaky kódují jinak podle toho, zda se nacházejí uprostřed slova nebo na jeho konci, můžeme rozumně odhadnout směr zdrojového textu a vrátit příslušné kódování (windows-1255 pro logickou hebrejštinu nebo ISO-8859-8 pro vizuální hebrejštinu).

15.3.5. windows-1252

Pokud UniversalDetector v textu detekuje znaky s nastaveným osmým bitem a žádný z vícebajtových nebo jednobajtových detekčních objektů nevrátil spolehlivý výsledek, vytvoří Latin1Prober (třída je definována v latin1prober.py) a snaží se detekovat anglický text v kódování windows-1252.

Tato detekce je ze své podstaty nespolehlivá, protože anglické znaky se kódují stejným způsobem v mnoha různých kódováních. Jediný způsob, jak lze kódování `windows-1252` rozpoznat, je založen na běžně používaných symbolech, jako jsou střídavé uvozovky (smart quotes; knižní, jiný znak na začátku a jiný na konci), kulaté apostrofy, symbol copyright a podobně. `Latin1Prober` automaticky redukuje ohodnocení své spolehlivosti, aby umožnil přesnějším detektorům vyhrát, pokud je to vůbec možné.

15.4. Spouštíme `2to3`

Jsme připraveni k přenesení modulu `chardet` z Pythonu 2 do Pythonu 3. Python 3 se dodává s pomocným skriptem nazvaným `2to3`, který jako vstup přebírá zdrojový kód napsaný pro Python 2 a automaticky převádí vše, co dovede, do podoby pro Python 3. V některých případech je to snadné — funkce se přejmenovala nebo se přesunula do jiného modulu —, ale v ostatních případech to může být docela složité. Abyste získali představu, co vše *umí* převést, podívejte se na přílohu Přepis kódu do Python 3 s využitím `2to3`. V této kapitole začneme spuštěním `2to3` pro balík `chardet`. Ale jak brzy uvidíte, po provedení kouzel automatickými nástroji nám zbude ještě spousta práce.

Hlavní balík `chardet` je rozdělen do několika různých souborů. Všechny se nacházejí ve stejném adresáři. Skript `2to3` převod více souborů najednou usnadňuje. Jako argument na příkazovém řádku stačí předat jméno adresáře a `2to3` převede každý ze souborů, které se v něm nacházejí.

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w chardet\
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- chardet\__init__.py (original)
+++ chardet\__init__.py (refactored)
@@ -18,7 +18,7 @@
 __version__ = "1.0.1"

 def detect(aBuf):
-    import universaldetector
+    from . import universaldetector
     u = universaldetector.UniversalDetector()
     u.reset()
     u.feed(aBuf)
--- chardet\big5prober.py (original)
+++ chardet\big5prober.py (refactored)
@@ -25,10 +25,10 @@
 # 02110-1301  USA
 ######################### END LICENSE BLOCK #########################
```

```
-from mbcharsetprober import MultiByteCharSetProber
-from codingstatemachine import CodingStateMachine
-from chardistribution import Big5DistributionAnalysis
-from mbcssm import Big5SMModel
+from .mbcharsetprober import MultiByteCharSetProber
+from .codingstatemachine import CodingStateMachine
+from .chardistribution import Big5DistributionAnalysis
+from .mbcssm import Big5SMModel

 class Big5Prober(MultiByteCharSetProber):
     def __init__(self):
--- chardet\chardistribution.py (original)
+++ chardet\chardistribution.py (refactored)
@@ -25,12 +25,12 @@
 # 02110-1301  USA
 ######################### END LICENSE BLOCK #########################

-import constants
-from euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE, EUCTW_TYPICAL_DISTRIBUTION_RATIO
-from euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE, EUCKR_TYPICAL_DISTRIBUTION_RATIO
-from gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE, GB2312_TYPICAL_DISTRIBUTION_RATIO
-from big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE, BIG5_TYPICAL_DISTRIBUTION_RATIO
-from jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE, JIS_TYPICAL_DISTRIBUTION_RATIO
+from . import constants
+from .euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE, EUCTW_TYPICAL_DISTRIBUTION_RATIO
+from .euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE, EUCKR_TYPICAL_DISTRIBUTION_RATIO
+from .gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE, GB2312_TYPICAL_DISTRIBUTION_RATIO
+from .big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE, BIG5_TYPICAL_DISTRIBUTION_RATIO
+from .jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE, JIS_TYPICAL_DISTRIBUTION_RATIO

 ENOUGH_DATA_THRESHOLD = 1024
 SURE_YES = 0.99
.
.
. (takhle to chvíli pokračuje)
.
.
RefactoringTool: Files that were modified:
RefactoringTool: chardet\__init__.py
RefactoringTool: chardet\big5prober.py
RefactoringTool: chardet\chardistribution.py
RefactoringTool: chardet\charsetgroupprober.py
RefactoringTool: chardet\codingstatemachine.py
```

```
RefactoringTool: chardet\constants.py
RefactoringTool: chardet\escprober.py
RefactoringTool: chardet\escsm.py
RefactoringTool: chardet\eucjpprober.py
RefactoringTool: chardet\euckrprober.py
RefactoringTool: chardet\euctwprober.py
RefactoringTool: chardet\gb2312prober.py
RefactoringTool: chardet\hebrewprober.py
RefactoringTool: chardet\jpcntx.py
RefactoringTool: chardet\langbulgarianmodel.py
RefactoringTool: chardet\langcyrillicmodel.py
RefactoringTool: chardet\langgreekmodel.py
RefactoringTool: chardet\langhebrewmodel.py
RefactoringTool: chardet\langhungarianmodel.py
RefactoringTool: chardet\langthaimodel.py
RefactoringTool: chardet\latin1prober.py
RefactoringTool: chardet\mbcharsetprober.py
RefactoringTool: chardet\mbcsgroupprober.py
RefactoringTool: chardet\mbcssm.py
RefactoringTool: chardet\sbcharsetprober.py
RefactoringTool: chardet\sbcsgroupprober.py
RefactoringTool: chardet\sjisprober.py
RefactoringTool: chardet\universaldetector.py
RefactoringTool: chardet\utf8prober.py
```

Teď spustíme skript `2to3` na testovací skript `test.py`.

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w test.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- test.py (original)
+++ test.py (refactored)
@@ -4,7 +4,7 @@
 count = 0
 u = UniversalDetector()
 for f in glob.glob(sys.argv[1]):
-    print f.ljust(60),
+    print(f.ljust(60), end=' ')
     u.reset()
     for line in file(f, 'rb'):
         u.feed(line)
```

```
@@ -12,8 +12,8 @@
     u.close()
     result = u.result
     if result['encoding']:
-        print result['encoding'], 'with confidence', result['confidence']
+        print(result['encoding'], 'with confidence', result['confidence'])
     else:
-        print '******** no result'
+        print('******** no result')
     count += 1
-print count, 'tests'
+print(count, 'tests')
RefactoringTool: Files that were modified:
RefactoringTool: test.py
```

No vida. Nebylo to tak hrozné. Konvertovalo se jen pár importů a příkazů print. Když už o tom mluvíme, *jaký byl* problém se všemi těmi příkazy import? Abychom na to mohli odpovědět, musíme rozumět tomu, jak se modul chardet dělí na více souborů.

15.5. Krátká odbočka k vícesouborovým modulům

`chardet` je *vícesouborový modul*. Mohl jsem se rozhodnout, že veškerý kód uložím do jednoho souboru (pojmenovaného `chardet.py`), ale neudělal jsem to. Místo toho jsem vytvořil adresář (pojmenovaný `chardet`) a v něm jsem vytvořil soubor `__init__.py`. *Pokud Python najde v adresáři soubor* `__init__.py`, *předpokládá, že všechny ostatní soubory ve stejném adresáři jsou součástí stejného modulu*. Jméno adresáře je jménem modulu. Soubory v adresáři se mohou odkazovat na ostatní soubory ve stejném adresáři nebo dokonce v jeho podadresářích. (Více si o tom řekneme za minutku.) Ale celá kolekce souborů se okolnímu pythonovskému kódu jeví jako jediný modul — jako kdyby všechny funkce a třídy byly definovány v jediném souboru s příponou `.py`.

A co je vlastně v souboru `__init__.py`? Nic. Všechno. Něco mezi tím. Soubor `__init__.py` nemusí definovat vůbec nic. Může to být doslova prázdný soubor. Nebo jej můžeme použít k definici funkcí, které jsou našimi hlavními vstupními body. Nebo do něj můžeme umístit všechny naše funkce. Podstatná je jediná věc.

> Adresář se souborem `__init__.py` se vždy považuje za vícesouborový modul. Pokud v adresáři není umístěn soubor `__init__.py`, považuje se prostě za adresář, který nemá k souborům s příponou `.py` žádný vztah.

Podívejme se, jak to funguje v praxi.

```
>>> import chardet
>>> dir(chardet)                                          [1]
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__path__', '__version__', 'detect']
>>> chardet                                               [2]
<module 'chardet' from 'C:\Python31\lib\site-packages\chardet\__init__.py'>
```

[1] Pokud neuvažujeme obvyklé atributy tříd, najdeme v modulu `chardet` jedinou věc a tou je funkce `detect()`.

[2] Tady máme první stopu, která říká, že modul `chardet` je víc než jen obyčejným souborem: u slova „module“ se ve výpisu objevuje soubor `__init__.py` umístěný v adresáři `chardet/`.

Nahlédněme do souboru `__init__.py`.

```
def detect(aBuf):                                         [1]
    from . import universaldetector                       [2]
    u = universaldetector.UniversalDetector()
    u.reset()
    u.feed(aBuf)
    u.close()
    return u.result
```

[1] V souboru `__init__.py` je definována funkce `detect()`, která je hlavním bodem knihovny `chardet`.

[2] Ale funkce `detect()` neobsahuje skoro žádný kód! Ve skutečnosti pouze importuje modul `universaldetector` a začíná jej používat. Ale kde je definován `universaldetector`?

Odpověď je skryta v tomto divně vypadajícím příkazu `import`:

```
from . import universaldetector
```

V překladu do češtiny to znamená „importuj modul `universaldetector`, který je umístěn ve stejném adresáři, jako já“. Tím „já“ se myslí soubor `chardet/__init__.py`. Říká se tomu *relativní import*. Představuje způsob, jakým se mohou soubory ve vícesouborovém modulu na sebe vzájemně odkazovat, aniž by se musely starat o konflikty jmen s jinými moduly, které můžeme mít nainstalované v naší vyhledávací cestě pro import. Uvedený příkaz `import` bude modul `universaldetector` hledat pouze uvnitř adresáře `chardet/`.

Zmíněné dva koncepty — `__init__.py` a relativní importy — znamenají, že náš modul můžeme rozbít na tolik kousků, kolik si přejeme. Modul `chardet` se skládá z 36 souborů s příponou `.py` — z 36! A přitom vše, co musíme udělat, když jej chceme začít používat, je `import chardet`. Pak můžeme zavolat hlavní funkci `chardet.detect()`. Aniž o tom náš kód ví, funkce `detect()` je ve skutečnosti definována v souboru `chardet/__init__.py`. A aniž o tom musíme vědět my, funkce `detect()`

používá k odkazu na třídu definovanou uvnitř chardet/universaldetector.py mechanismus relativního importu, který zase používá relativní import pěti dalších souborů, které se rovněž nacházejí v adresáři chardet/.

> Kdykoliv se přistihnete, že v Pythonu píšete rozsáhlou knihovnu (nebo, což je pravděpodobnější, když zjistíte, že se vaše malá knihovna rozrostla ve velkou), udělejte si čas na refaktorizaci a změňte ji na vícesouborový modul. Je to jedna z mnoha věcí, ve kterých je Python dobrý. Takže té výhody využijte.

15.6. Opravme, co 2to3 neumí

15.6.1. False je syntaktická chyba

Teď zkusíme skutečný test. Spustíme testovací sadu (test suite) na zkušební skript (test harness). Protože je testovací sada navržena tak, aby pokryla všechny možné cesty, kudy se běh programu může kódem ubírat, jde o dobrý způsob, jak ověřit, že v našem přeneseném kódu někde nejsou skryté chyby.

Píšete testy, že?

```
C:\home\chardet> python test.py tests\*\*
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    from chardet.universaldetector import UniversalDetector
  File "C:\home\chardet\chardet\universaldetector.py", line 51
    self.done = constants.False
                              ^
SyntaxError: invalid syntax
```

Hmm, to je jen drobnost. V Pythonu 3 je False vyhrazeným slovem, takže je nemůžeme použít jako jméno proměnné. Podíváme se do constants.py na to, kde je proměnná definována. Tady máme původní verzi z constants.py předtím, než ji skript 2to3 změnil:

```
import __builtin__
if not hasattr(__builtin__, 'False'):
    False = 0
    True = 1
else:
    False = __builtin__.False
    True = __builtin__.True
```

Tento kus kódu byl navržen, aby knihovna běžela ve starších verzích Pythonu 2. Před Pythonem 2.3 neexistoval zabudovaný typ `bool`. Uvedený kód detekuje nepřítomnost zabudovaných konstant `True` a `False` a v případě potřeby je definuje.

Ale v Pythonu 3 je typ `bool` přítomen vždy, takže je celý úryvek kódu zbytečný. Nejjednodušší řešení spočívá v nahrazení všech výskytů `constants.True` a `constants.False` hodnotami `True` a `False`. Pak z `constants.py` odstraníme onen mrtvý kód.

Takže následující řádek v `universaldetector.py`

```
self.done = constants.False
```

se změní na

```
self.done = False
```

Ách, nebylo to uspokojující? Kód je teď kratší a čitelnější.

15.6.2. Nenalezen modul `constants`

Nastal čas spustit znovu `test.py`. Uvidíme, jak daleko se dostaneme.

```
C:\home\chardet> python test.py tests\*\*
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    from chardet.universaldetector import UniversalDetector
  File "C:\home\chardet\chardet\universaldetector.py", line 29, in <module>
    import constants, sys
ImportError: No module named constants
```

Co to říká? Jaképak „No module named `constants`“ (doslova „žádný modul jménem `constants`“)? Modul constants tam samozřejmě je! Je přímo tady v `chardet/constants.py`.

Vzpomínáte si, jak skript `2to3` opravil všechny ty příkazy import? Tato knihovna používá množství relativních importů — moduly, které importují jiné moduly nacházející se uvnitř stejné knihovny —, ale *v Pythonu 3 se změnila logika relativních importů*. V Pythonu 2 jsme mohli jednoduše provést `import constants` a Python by nejdříve prohledával adresář `chardet/`. V Pythonu 3 jsou všechny příkazy import absolutní. Pokud chceme v Pythonu 3 provést relativní import, musíme to říct explicitně:

```
from . import constants
```

No moment. Neměl se o tohle postarat skript 2to3 za nás? No, on to udělal. Ale tento konkrétní příkaz import kombinoval dva typy importu na jednom řádku: relativní import modulu constants, který se nachází uvnitř knihovny, a absolutní import modulu sys, který je předinstalován jako součást pythonovské standardní knihovny. V Pythonu 2 jsme je mohli zkombinovat do jednoho řádku příkazu import. V Pythonu 3 to nejde a skript 2to3 není dost chytrý na to, aby příkaz import rozdělil na dva.

Řešení spočívá v ručním rozdělení příkazu import. Takže tento import „dva v jednom"...

```
import constants, sys
```

... musíme změnit na dva oddělené importy:

```
from . import constants
import sys
```

Variace tohoto problému jsou rozesety po celé knihovně chardet. Na některých místech je to „import constants, sys", jinde je to „import constants, re". Oprava je stejná. Ručně rozdělíme příkaz import na dva řádky. Na jednom uvedeme relativní import, na druhém absolutní import.

Kupředu!

15.6.3. Jméno 'file' není definováno

A zase jdeme na to. Spouštíme test.py, abychom provedli naše testovací případy…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    for line in file(f, 'rb'):
NameError: name 'file' is not defined
```

open() je novým file(). PapayaWhip je nová černá.

Tak tohle mě překvapilo, protože tento obrat jsem používal, co mi paměť sahá. V Pythonu 2 byla globální funkce file() jiným jménem (alias) pro funkci open(), která představovala standardní způsob otvírání textových souborů pro čtení. V Pythonu 3 už globální funkce file() neexistuje, ale funkce open() je tu nadále.

Takže nejjednodušší řešení problému chybějící funkce file() spočívá v jejím nahrazení voláním funkce open():

```
for line in open(f, 'rb'):
```

A to je vše, co o tom můžu říct.

15.6.4. Řetězcový vzorek nelze použít pro bajtové objekty

Teď se začnou dít zajímavé věci. Slůvkem „zajímavé“ rozumím „pekelně matoucí“.

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 98, in feed
    if self._highBitDetector.search(aBuf):
TypeError: can't use a string pattern on a bytes-like object
```

Abychom to odladili, podívejme se, co je `self._highBitDetector`. Je to definováno v metodě `__init__` třídy `UniversalDetector`:

```
class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(r'[\x80-\xFF]')
```

Jde o předkompilovaný regulární výraz, který má hledat znaky mimo `ASCII`, tj. v rozsahu 128–255 (0x80–0xFF). Počkat, tohle není úplně správně. Musíme použít přesnější terminologii. Tento vzorek je navržen pro hledání bajtů s hodnotou mimo ascii, tedy v rozsahu 128–255.

A v tom je ten problém.

V Pythonu 2 byl řetězec polem bajtů. Jeho kódování znaků bylo zachyceno odděleně. Pokud jsme po Pythonu 2 chtěli, aby znakové kódování udržoval u řetězce, museli jsme použít Unicode řetězec (`u''`). Ale v Pythonu 3 je řetězec vždy tím, co Python 2 nazýval Unicode řetězec — to znamená polem Unicode znaků (které mohou být vyjádřeny různým počtem bajtů). A protože je tento regulární výraz definován řetězcovým vzorkem, může být použit jen pro prohledávání řetězců, což je pole znaků. Ale my nechceme prohledávat řetězec. Prohledáváme pole bajtů. Pohledem na trasovací výpis zjistíme, že k chybě došlo v `universaldetector.py`:

```
def feed(self, aBuf):
    .
    .
    .
    if self._mInputState == ePureAscii:
        if self._highBitDetector.search(aBuf):
```

A co je to aBuf? Podívejme se ještě o kousek zpět, na místo, kde se volá UniversalDetector.feed(). Jedno z míst, kde se volá, se nachází v testovacím kódu (test harness) test.py.

```
u = UniversalDetector()
.
.
.
for line in open(f, 'rb'):
    u.feed(line)
```

Není to pole znaků, ale pole bajtů.

A tady máme odpověď: aBuf je řádek načítaný v metodě UniversalDetector.feed() ze souboru na disku. Podívejte se pořádně na parametry, které se používají při otvírání souboru: 'rb'. 'r' znamená „read“ (čtení). No dobrá, to je toho. Čteme ze souboru. No jo! 'b' znamená „binárně“. Bez příznaku 'b' by cyklus for četl soubor po řádcích a každý řádek by převáděl na řetězec — tedy na pole Unicode znaků — s využitím systémového výchozího znakového kódování. Ale s příznakem 'b' čte cyklus for ze souboru po řádcích a každý řádek ukládá do pole bajtů přesně v takovém tvaru, v jakém se nachází v souboru. Výsledné pole bajtů se předává do UniversalDetector.feed() a nakonec se dostane až k předkompilovanému regulárnímu výrazu self._highBitDetector, aby se našly osmibitové… znaky. Ale my nemáme znaky. My máme bajty. A do prčic.

Potřebujeme, aby tento regulární výraz nehledal v poli znaků, ale v poli bajtů.

Když už jsme na to přišli, bude náprava jednoduchá. Regulární výrazy definované řetězci mohou hledat v řetězcích. Regulární výrazy definované poli bajtů mohou hledat v polích bajtů. Abychom definovali vzorek polem bajtů, jednoduše změníme typ argumentu, který používáme pro definici regulárního výrazu, na pole bajtů. (Hned na následujícím řádku je další případ téhož problému.)

```
  class UniversalDetector:
      def __init__(self):
-         self._highBitDetector = re.compile(r'[\x80-\xFF]')
-         self._escDetector = re.compile(r'(\033|~{)')
+         self._highBitDetector = re.compile(b'[\x80-\xFF]')
+         self._escDetector = re.compile(b'(\033|~{)')
          self._mEscCharSetProber = None
          self._mCharSetProbers = []
          self.reset()
```

Když necháme ve všech zdrojových textech vyhledat použití modulu `re`, objevíme další dva případy v `charsetprober.py`. Jde opět o případy, kdy jsou regulární výrazy definovány jako řetězce, ale používáme je pro `aBuf`, což je pole bajtů. Řešení je stejné: definujeme vzorky regulárních výrazů jako pole bajtů.

```
  class CharSetProber:
      .
      .
      .
      def filter_high_bit_only(self, aBuf):
-         aBuf = re.sub(r'([\x00-\x7F])+', ' ', aBuf)
+         aBuf = re.sub(b'([\x00-\x7F])+', b' ', aBuf)
          return aBuf

      def filter_without_english_letters(self, aBuf):
-         aBuf = re.sub(r'([A-Za-z])+', ' ', aBuf)
+         aBuf = re.sub(b'([A-Za-z])+', b' ', aBuf)
          return aBuf
```

15.6.5. Objekt typu `'bytes'` nelze implicitně převést na `str`

Divoucnější a divoucnější…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 100, in feed
    elif (self._mInputState == ePureAscii) and self._escDetector.search(self._mLastChar + aBuf):
TypeError: Can't convert 'bytes' object to str implicitly
```

Zde dochází k nešťastné kolizi mezi stylem zápisu zdrojového textu a interpretem Pythonu. Chyba `TypeError` se může vázat na kteroukoliv část řádku, ale trasovací výpis nám neříká, kde přesně je. Může to být v první nebo v druhé části podmínky, ale z trasovacího výpisu se to nepozná. Abychom prostor pro hledání zúžili, měli bychom řádek rozdělit:

```
elif (self._mInputState == ePureAscii) and \
    self._escDetector.search(self._mLastChar + aBuf):
```

A znovu spustíme test:

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: Can't convert 'bytes' object to str implicitly
```

Aha! Problém se nevyskytoval v první části podmínky (`self._mInputState == ePureAscii`), ale v druhé. Takže co zde vlastně způsobuje chybu `TypeError`? Možná si myslíte, že metoda `search()` očekává hodnotu odlišného typu. To by ale nevygenerovalo takový trasovací výpis. Pythonovské funkce mohou přebírat libovolné hodnoty. Pokud předáme správný počet argumentů, funkce se provede. Pokud bychom předali hodnotu jiného typu, než funkce očekává, mohla by *havarovat*. Ale pokud by se tak stalo, trasovací výpis by ukazoval na místo někde uvnitř funkce. Jenže tento trasovací výpis říká, že se nikdy nedošlo tak daleko, aby se metoda `search()` zavolala. Takže problém musí být skryt v operaci `+`, protože ta se snaží o zkonstruování hodnoty, která bude nakonec předána metodě `search()`.

Z předchozího ladění víme, že `aBuf` je polem bajtů. A co je tedy `self._mLastChar`? Jde o členskou proměnnou definovanou v metodě `reset()`, která je ve skutečnosti volána z metody `__init__()`.

```
class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(b'[\x80-\xFF]')
        self._escDetector = re.compile(b'(\033|~{)')
        self._mEscCharSetProber = None
        self._mCharSetProbers = []
        self.reset()

    def reset(self):
        self.result = {'encoding': None, 'confidence': 0.0}
        self.done = False
        self._mStart = True
        self._mGotData = False
        self._mInputState = ePureAscii
        self._mLastChar = ''
```

A tady máme odpověď. Vidíte to? `self._mLastChar` je řetězec, ale `aBuf` je pole bajtů. Konkatenaci (zřetězení, spojení) nelze provádět pro řetězec a pole bajtů — ani když jde o řetězec nulové délky.

No dobrá, ale k čemu je tedy `self._mLastChar`? V metodě `feed()`, jen pár řádků pod místem označeným v trasovacím výpisu, vidíme...

```
if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
            self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

self._mLastChar = aBuf[-1]
```

Volající funkce volá metodu `feed()` pořád dokola s tím, že jí pokaždé předá pár bajtů. Metoda zpracuje zadané bajty (dostává je v `aBuf`) a potom uloží poslední bajt do `self._mLastChar` pro případ, že by jej potřebovala při dalším volání. (Při použití vícebajtového kódování by metoda `feed()` mohla být zavolána pro polovinu znaku a pak by mohla být volána pro jeho druhou polovinu.) Ale protože je teď `aBuf` místo řetězce polem bajtů, musíme udělat pole bajtů i z `self._mLastChar`. Takže:

```
  def reset(self):
      .
      .
      .
-     self._mLastChar = ''
+     self._mLastChar = b''
```

Když ve všech zdrojových souborech vyhledáme „`mLastChar`“, najdeme podobný problém v `mbcharsetprober.py`. Ale místo uchovávání posledního znaku se uchovávají poslední dva znaky. Třída `MultiByteCharSetProber` používá k uchovávání posledních dvou znaků seznam jednoznakových řetězců. V Pythonu 3 musíme použít seznam celých čísel, protože ve skutečnosti neuchováváme znaky, ale bajty. (Bajty jsou prostě celá čísla v intervalu `0–255`.)

```
  class MultiByteCharSetProber(CharSetProber):
      def __init__(self):
          CharSetProber.__init__(self)
          self._mDistributionAnalyzer = None
          self._mCodingSM = None
-         self._mLastChar = ['\x00', '\x00']
+         self._mLastChar = [0, 0]

      def reset(self):
          CharSetProber.reset(self)
          if self._mCodingSM:
              self._mCodingSM.reset()
          if self._mDistributionAnalyzer:
              self._mDistributionAnalyzer.reset()
-         self._mLastChar = ['\x00', '\x00']
+         self._mLastChar = [0, 0]
```

15.6.6. Nepodporované typy operandů pro +: 'int' a 'bytes'

Mám jednu dobrou a jednu špatnou zprávu. Ta dobrá je, že děláme pokroky…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: unsupported operand type(s) for +: 'int' and 'bytes'
```

…Ta špatná je, že to někdy tak nevypadá.

Ale on to je pokrok! Opravdu! I když trasovací výpis označuje stejný řádek kódu, je to jiná chyba, než se hlásila dříve. Pokrok! Takže kdepak máme problém teď? Když jsme to kontrolovali minule, nesnažil se tento řádek řetězit `int` s polem bajtů (`bytes`). Ve skutečnosti jsme strávili dost času tím, abychom zajistili, že `self._mLastChar` bude pole bajtů. Jak se mohlo změnit na `int`?

Odpověď není skrytá v předchozích řádcích kódu, ale v následujících.

```
if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
            self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

self._mLastChar = aBuf[-1]
```

Každý prvek řetězce je řetězcem. Každý prvek z pole bajtů je celé číslo.

Tato chyba se nevyskytne při prvním volání metody `feed()`. Vyskytne se *při druhém volání* poté, co byl proměnné `self._mLastChar` přiřazen poslední bajt `aBuf`. No a v čem je tedy problém? Když z bajtového pole získáme jeden prvek, dostaneme celé číslo a ne bajtové pole. Abychom ten rozdíl viděli, ukážeme si to v interaktivním shellu:

```
>>> aBuf = b'\xEF\xBB\xBF'                          [1]
>>> len(aBuf)
3
>>> mLastChar = aBuf[-1]
>>> mLastChar                                       [2]
191
>>> type(mLastChar)                                 [3]
<class 'int'>
>>> mLastChar + aBuf                                [4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'bytes'
>>> mLastChar = aBuf[-1:]                           [5]
>>> mLastChar
b'\xbf'
>>> mLastChar + aBuf                                [6]
b'\xbf\xef\xbb\xbf'
```

[1] Definujeme pole bajtů o délce 3.

[2] Poslední prvek pole bajtů má hodnotu 191.

[3] Je to celé číslo (integer).

[4] Zřetězení pole bajtů s celým číslem nefunguje. Právě jsme navodili chybu, kterou jsme pozorovali v `universaldetector.py`.

[5] A tady máme nápravu. Místo získávání posledního prvku z pole bajtů použijeme operaci pro získání výřezu (slicing). Vytvoříme jí nové pole bajtů, které obsahuje jen poslední prvek. To znamená, že začneme posledním prvkem a pokračujeme v tvorbě výřezu (slice), dokud nedosáhneme konce pole bajtů. Teď je `mLastChar` polem bajtů o délce 1.

[6] Zřetězením pole bajtů o délce 1 s polem bajtů o délce 3 dostaneme nové pole bajtů o délce 4.

Takže abychom zajistili, že bude metoda `feed()` v `universaldetector.py` pokračovat v činnosti nezávisle na tom, jak často je volána, musíme inicializovat `self._mLastChar` polem bajtů o nulové délce a potom *musíme zajistit, aby tato proměnná zůstala polem bajtů*.

```
            self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

- self._mLastChar = aBuf[-1]
+ self._mLastChar = aBuf[-1:]
```

15.6.7. Funkce ord() očekávala řetězec o délce 1, ale byl nalezen in

Jste už unaveni? Už to máme skoro hotové…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                       ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\utf8prober.py", line 53, in feed
    codingState = self._mCodingSM.next_state(c)
  File "C:\home\chardet\chardet\codingstatemachine.py", line 43, in next_state
    byteCls = self._mModel['classTable'][ord(c)]
TypeError: ord() expected string of length 1, but int found
```

OK, takže c je typu int, ale funkce ord() očekávala jednoznakový řetězec. No dobrá. Kde je definována proměnná c?

```
# codingstatemachine.py
def next_state(self, c):
    # for each byte we get its class
    # if it is first byte, we also get byte length
    byteCls = self._mModel['classTable'][ord(c)]
```

To nám nepomůže. Tady se jen předává funkci. Podívejme se hlouběji do zásobníku.

```
# utf8prober.py
def feed(self, aBuf):
    for c in aBuf:
        codingState = self._mCodingSM.next_state(c)
```

Vidíte to? V Pythonu 2 byla proměnná aBuf řetězcem, takže proměnná c byla jednoznakovým řetězcem. (Ten dostáváme, když iterujeme přes řetězec — všechny znaky, jeden po druhém.) Ale teď je aBuf polem bajtů, takže c je typu int a ne jednoznakový řetězec. Jinými slovy, už nepotřebujeme volat funkci ord(), protože c už je typu int!

Takže:

```
  def next_state(self, c):
      # for each byte we get its class
      # if it is first byte, we also get byte length
-     byteCls = self._mModel['classTable'][ord(c)]
+     byteCls = self._mModel['classTable'][c]
```

Vyhledáním „ord(c)“ ve všech zdrojových textech odhalíme podobné problémy v sbcharsetprober.py…

```
# sbcharsetprober.py
def feed(self, aBuf):
    if not self._mModel['keepEnglishLetter']:
        aBuf = self.filter_without_english_letters(aBuf)
    aLen = len(aBuf)
    if not aLen:
        return self.get_state()
    for c in aBuf:
        order = self._mModel['charToOrderMap'][ord(c)]
```

… a v latin1prober.py…

```
# latin1prober.py
def feed(self, aBuf):
    aBuf = self.filter_with_english_letters(aBuf)
    for c in aBuf:
        charClass = Latin1_CharToClass[ord(c)]
```

Proměnná c iteruje přes aBuf, což znamená, že v ní bude celé číslo a ne jednoznakový řetězec. Řešení je stejné: ord(c) změníme na prosté c.

```
  # sbcharsetprober.py
  def feed(self, aBuf):
      if not self._mModel['keepEnglishLetter']:
          aBuf = self.filter_without_english_letters(aBuf)
      aLen = len(aBuf)
      if not aLen:
          return self.get_state()
      for c in aBuf:
-         order = self._mModel['charToOrderMap'][ord(c)]
+         order = self._mModel['charToOrderMap'][c]
```

```
  # latin1prober.py
  def feed(self, aBuf):
      aBuf = self.filter_with_english_letters(aBuf)
      for c in aBuf:
-         charClass = Latin1_CharToClass[ord(c)]
+         charClass = Latin1_CharToClass[c]
```

15.6.8. Neuspořádatelné datové typy: `int() >= str()`

A spusťme to znovu.

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                       ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\sjisprober.py", line 68, in feed
    self._mContextAnalyzer.feed(self._mLastChar[2 - charLen :], charLen)
  File "C:\home\chardet\chardet\jpcntx.py", line 145, in feed
    order, charLen = self.get_order(aBuf[i:i+2])
  File "C:\home\chardet\chardet\jpcntx.py", line 176, in get_order
    if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
TypeError: unorderable types: int() >= str()
```

A co se děje zase tady? „Unorderable types“ čili neuspořádatelné typy? (Neuspořádatelné ve smyslu, že mezi těmito hodnotami nelze určit pořadí.) A rozdíl mezi bajty a řetězci znovu vystrkuje svou ošklivou hlavu. Ale podívejte se na kód:

```
class SJISContextAnalysis(JapaneseContextAnalysis):
    def get_order(self, aStr):
        if not aStr: return -1, 1
        # find out current char's byte length
        if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
           ((aStr[0] >= '\xE0') and (aStr[0] <= '\xFC')):
            charLen = 2
        else:
            charLen = 1
```

A odkud se vzala proměnná aStr? Podívejme se hlouběji do zásobníku:

```
def feed(self, aBuf, aLen):
    .
    .
    .
    i = self._mNeedToSkipCharNum
    while i < aLen:
        order, charLen = self.get_order(aBuf[i:i+2])
```

Hele, podívejme. To je náš starý přítel aBuf. Jak už jste mohli odhadnout ze všech předchozích problémů, se kterými jsme se v této kapitole setkali, aBuf je pole bajtů. V tomto místě je metoda feed() nepředává jako celek. Vytváří z něj výřez. Ale jak jsme viděli v této kapitole o něco dříve, výřezem z pole bajtů vznikne pole bajtů. Takže parametr aStr, který přebírá metoda get_order(), je pořád pole bajtů.

A co se tento kód s aStr pokouší dělat? Získává první prvek z pole bajtů a srovnává jej s jednoznakovým řetězcem. V Pythonu 2 to fungovalo, protože aStr a aBuf byly řetězce a aStr[0] by byl taky řetězec. U řetězců můžeme zjišťovat, zda jsou různé. Ale v Pythonu 3 jsou proměnné aStr a aBuf poli bajtů a aStr[0] je celé číslo. Číslo a řetězec nemůžeme porovnávat na neshodu, aniž jednu z hodnot explicitně nepřevedeme na stejný typ.

V tomto případě nemusíme kód komplikovat přidáváním explicitního převodu typu. aStr[0] je celé číslo. Vše, s čím ho srovnáváme, jsou konstanty. Můžeme je změnit z jednoznakových řetězců na čísla. A když už to děláme, změňme také identifikátor aStr na aBuf, protože to ve skutečnosti není řetězec (string).

```
  class SJISContextAnalysis(JapaneseContextAnalysis):
-     def get_order(self, aStr):
-      if not aStr: return -1, 1
+     def get_order(self, aBuf):
+      if not aBuf: return -1, 1
           # find out current char's byte length
-         if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
-            ((aBuf[0] >= '\xE0') and (aBuf[0] <= '\xFC')):
+         if ((aBuf[0] >= 0x81) and (aBuf[0] <= 0x9F)) or \
+            ((aBuf[0] >= 0xE0) and (aBuf[0] <= 0xFC)):
              charLen = 2
          else:
              charLen = 1
```

```
          # return its order if it is hiragana
-      if len(aStr) > 1:
-            if (aStr[0] == '\202') and \
-               (aStr[1] >= '\x9F') and \
-               (aStr[1] <= '\xF1'):
-               return ord(aStr[1]) - 0x9F, charLen
+      if len(aBuf) > 1:
+            if (aBuf[0] == 202) and \
+               (aBuf[1] >= 0x9F) and \
+               (aBuf[1] <= 0xF1):
+               return aBuf[1] - 0x9F, charLen

          return -1, charLen

  class EUCJPContextAnalysis(JapaneseContextAnalysis):
-     def get_order(self, aStr):
-      if not aStr: return -1, 1
+     def get_order(self, aBuf):
+      if not aBuf: return -1, 1
          # find out current char's byte length
-         if (aStr[0] == '\x8E') or \
-           ((aStr[0] >= '\xA1') and (aStr[0] <= '\xFE')):
+         if (aBuf[0] == 0x8E) or \
+           ((aBuf[0] >= 0xA1) and (aBuf[0] <= 0xFE)):
              charLen = 2
-         elif aStr[0] == '\x8F':
+         elif aBuf[0] == 0x8F:
              charLen = 3
          else:
              charLen = 1

        # return its order if it is hiragana
-    if len(aStr) > 1:
-          if (aStr[0] == '\xA4') and \
-             (aStr[1] >= '\xA1') and \
-             (aStr[1] <= '\xF3'):
-                return ord(aStr[1]) - 0xA1, charLen
+    if len(aBuf) > 1:
+          if (aBuf[0] == 0xA4) and \
+             (aBuf[1] >= 0xA1) and \
+             (aBuf[1] <= 0xF3):
+              return aBuf[1] - 0xA1, charLen
        return -1, charLen
```

Hledáním výskytu funkce ord() ve zdrojových textech odkryjeme stejný problém v chardistribution.py (konkrétně ve třídách EUCTWDistributionAnalysis, EUCKRDistributionAnalysis, GB2312DistributionAnalysis, Big5DistributionAnalysis, SJISDistributionAnalysis a EUCJPDistributionAnalysis). Ve všech případech se oprava podobá změnám, které jsme provedli v třídách EUCJPContextAnalysis a SJISContextAnalysis v souboru jpcntx.py.

15.6.9. Globální jméno 'reduce' není definováno

Ještě jedna trhlina…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                       ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    u.close()
  File "C:\home\chardet\chardet\universaldetector.py", line 141, in close
    proberConfidence = prober.get_confidence()
  File "C:\home\chardet\chardet\latin1prober.py", line 126, in get_confidence
    total = reduce(operator.add, self._mFreqCounter)
NameError: global name 'reduce' is not defined
```

Podle oficiálního průvodce **What's New In Python 3.0** byla funkce reduce() vyňata z globálního prostoru jmen a přesunuta do modulu functools. Citujme z průvodce: „Pokud opravdu potřebujete functools.reduce(), použijte ji. Ale v 99 procentech případů je explicitní cyklus for čitelnější.“ O tomto rozhodnutí se dočtete více na weblogu Guida van Rossuma: **The fate of reduce() in Python 3000** (Osud reduce v Pythonu 3000).

```
def get_confidence(self):
    if self.get_state() == constants.eNotMe:
        return 0.01

    total = reduce(operator.add, self._mFreqCounter)
```

Funkce reduce() přebírá dva argumenty — funkci a seznam (přesněji řečeno, může to být libovolný iterovatelný objekt) — a kumulativně aplikuje zadanou funkci na každý z prvků seznamu. Jinými slovy, jde o efektní a nepřímý způsob realizace součtu všech prvků seznamu.

Tato obludnost byla tak běžná, že byla do Pythonu přidána globální funkce sum().

```
  def get_confidence(self):
      if self.get_state() == constants.eNotMe:
          return 0.01

-     total = reduce(operator.add, self._mFreqCounter)
+     total = sum(self._mFreqCounter)
```

Protože jsme přestali používat modul `operator`, můžeme také ze začátku souboru odstranit příslušný příkaz `import`.

```
  from .charsetprober import CharSetProber
  from . import constants
- import operator
```

A tož, možeme to otestovať?

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                      ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml                            Big5 with confidence 0.99
tests\Big5\blog.worren.net.xml                              Big5 with confidence 0.99
tests\Big5\carbonxiv.blogspot.com.xml                       Big5 with confidence 0.99
tests\Big5\catshadow.blogspot.com.xml                       Big5 with confidence 0.99
tests\Big5\coolloud.org.tw.xml                              Big5 with confidence 0.99
tests\Big5\digitalwall.com.xml                              Big5 with confidence 0.99
tests\Big5\ebao.us.xml                                      Big5 with confidence 0.99
tests\Big5\fudesign.blogspot.com.xml                        Big5 with confidence 0.99
tests\Big5\kafkatseng.blogspot.com.xml                      Big5 with confidence 0.99
tests\Big5\ke207.blogspot.com.xml                           Big5 with confidence 0.99
tests\Big5\leavesth.blogspot.com.xml                        Big5 with confidence 0.99
tests\Big5\letterlego.blogspot.com.xml                      Big5 with confidence 0.99
tests\Big5\linyijen.blogspot.com.xml                        Big5 with confidence 0.99
tests\Big5\marilynwu.blogspot.com.xml                       Big5 with confidence 0.99
tests\Big5\myblog.pchome.com.tw.xml                         Big5 with confidence 0.99
tests\Big5\oui-design.com.xml                               Big5 with confidence 0.99
tests\Big5\sanwenji.blogspot.com.xml                        Big5 with confidence 0.99
tests\Big5\sinica.edu.tw.xml                                Big5 with confidence 0.99
tests\Big5\sylvia1976.blogspot.com.xml                      Big5 with confidence 0.99
tests\Big5\tlkkuo.blogspot.com.xml                          Big5 with confidence 0.99
tests\Big5\tw.blog.xubg.com.xml                             Big5 with confidence 0.99
tests\Big5\unoriginalblog.com.xml                           Big5 with confidence 0.99
tests\Big5\upsaid.com.xml                                   Big5 with confidence 0.99
tests\Big5\willythecop.blogspot.com.xml                     Big5 with confidence 0.99
tests\Big5\ytc.blogspot.com.xml                             Big5 with confidence 0.99
```

```
tests\EUC-JP\aivy.co.jp.xml                                    EUC-JP with confidence 0.99
tests\EUC-JP\akaname.main.jp.xml                               EUC-JP with confidence 0.99
tests\EUC-JP\arclamp.jp.xml                                    EUC-JP with confidence 0.99
.
.
.
316 tests
```

No to mě podrž, ono to funguje! */me si trošku zatancuje.*

15.7. Shrnutí

Co jsme se naučili?

1. Přepisování jakéhokoliv netriviálního kódu z Pythonu 2 do Pythonu 3 bude bolestivé. Nedá se to obejít. Je to obtížné.
2. Automatický nástroj `2to3` nám částečně pomůže, ale postará se jen o snadnější části — přejmenování funkcí, přejmenování modulů, úpravy syntaxe. Jde o impozantní kus inženýrské práce, ale koneckonců jde jen o inteligentního robota provádějícího vyhledávání a náhrady.
3. Problémem č. 1 při přepisování této knihovny byl rozdíl mezi řetězci a bajty. V tomto případě se to zdá být zřejmé, protože hlavním účelem knihovny `chardet` je převod proudu bajtů na řetězec. Ale „s proudem bajtů“ se setkáváme častěji, než byste si mysleli. Čtete soubor v „binárním“ režimu? Dostáváte proud bajtů. Získáváte obsah webovské stránky? Voláte webové aplikační rozhraní? Také se vrací proud bajtů.
4. Programu musíte rozumět vy. Skrz naskrz. Především protože jste ho napsali, ale musíte se vyrovnat se všemi jeho triky a zatuchlými kouty. Chyby jsou všude.
5. Testovací případy jsou nepostradatelné. Pokud je nemáte, nic nepřepisujte. *Jediný* důvod, proč věřím tomu, že `chardet` funguje v Pythonu 3, spočívá v tom, že jsem začal s testovací sadou, která prověřovala všechny hlavní cesty, kudy se kód ubírá. Pokud žádné testy nemáte, napište je dříve, než začnete přenos do Pythonu 3 realizovat. Pokud máte jen pár testů, napište jich víc. Pokud máte hodně testů, pak teprve může začít opravdová legrace.

“You’ll find the shame is like the pain; you only feel it once.”
(Zjistíte, že stud je jako bolest; ale cítíte ho jen jednou.)

— Markýza de Merteuil, *Dangerous Liaisons (Nebezpečné známosti)*

16. Balení pythonovských knihoven

16. Balení pythonovských knihoven — 359
16.1. Ponořme se — **361**
16.2. Věci, které za nás Distutils neudělají — **362**
16.3. Struktura adresáře — **363**
16.4. Píšeme svůj instalační skript — **364**
16.5. Přidáváme klasifikaci našeho balíčku — **366**
16.5.1. Příklady dobrých klasifikátorů balíčků — **367**
16.6. Určení dalších souborů prostřednictvím manifestu — **368**
16.7. Kontrola chyb v našem instalačním skriptu — **369**
16.8. Vytvoření distribuce obsahující zdrojové texty — **369**
16.9. Vytvoření grafického instalačního programu — **371**
16.9.1. Tvorba instalačních balíčků pro jiné operační systémy — **373**
16.10. Přidání našeho softwaru do Python Package Index — **373**
16.11. Více možných budoucností balení pythonovských produktů — **375**
16.12. Přečtěte si — **375**

16.1. Ponořme se

Opravdoví umělci prodávají. Alespoň takhle to říká Steve Jobs. Chcete vydat pythonovský skript, knihovnu, rámec (framework) nebo aplikaci? Výborně. Svět potřebuje více pythonovského kódu. Python 3 se dodává s rámcem pro vytváření balíčků zvaným Distutils. Distutils v sobě skrývá mnoho věcí: nástroj pro sestavení (build tool; pro vás), instalační nástroj (pro vaše uživatele), formát metadat balíčků (pro vyhledávače) a další. Tvoří celek s **Python Package Index** („PyPI“), což je centrální archiv pythonovských open-source knihoven.

Všechny uvedené stránky nástroje Distutils jsou soustředěny kolem *instalačního skriptu*, který se tradičně nazývá `setup.py`. Ve skutečnosti už jste v této knize několik instalačních skriptů vytvořených nástrojem Distutils viděli. Distutils jste použili k instalaci `httplib2` v kapitole Webové služby nad HTTP a znovu k instalaci `chardet` v Případové studii: Přepis `chardet` pro Python 3.

Kap.

Kap.

V této kapitole si prostudujeme, jak instalační skripty pro `chardet` a pro `httplib2` pracují, a projdeme si procesem vydání vašeho vlastního pythonovského softwaru.

```
# chardet's setup.py
from distutils.core import setup
setup(
    name = "chardet",
    packages = ["chardet"],
    version = "1.0.2",
    description = "Universal encoding detector",
    author = "Mark Pilgrim",
    author_email = "mark@diveintomark.org",
    url = "http://chardet.feedparser.org/",
    download_url = "http://chardet.feedparser.org/download/python3-chardet-1.0.1.tgz",
    keywords = ["encoding", "i18n", "xml"],
    classifiers = [
        "Programming Language :: Python",
        "Programming Language :: Python :: 3",
        "Development Status :: 4 - Beta",
        "Environment :: Other Environment",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: GNU Library or Lesser General Public License (LGPL)",
        "Operating System :: OS Independent",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Text Processing :: Linguistic",
        ],
    long_description = """\
Universal character encoding detector
```

```
-----------------------------------

Detects
 - ASCII, UTF-8, UTF-16 (2 variants), UTF-32 (4 variants)
 - Big5, GB2312, EUC-TW, HZ-GB-2312, ISO-2022-CN (Traditional and Simplified Chinese)
 - EUC-JP, SHIFT_JIS, ISO-2022-JP (Japanese)
 - EUC-KR, ISO-2022-KR (Korean)
 - KOI8-R, MacCyrillic, IBM855, IBM866, ISO-8859-5, windows-1251 (Cyrillic)
 - ISO-8859-2, windows-1250 (Hungarian)
 - ISO-8859-5, windows-1251 (Bulgarian)
 - windows-1252 (English)
 - ISO-8859-7, windows-1253 (Greek)
 - ISO-8859-8, windows-1255 (Visual and Logical Hebrew)
 - TIS-620 (Thai)

This version requires Python 3 or later; a Python 2 version is available separately.
"""
)
```

> `chardet` a `httplib2` jsou open source, ale neexistuje žádný požadavek na to, abyste své vlastní pythonovské knihovny vydávali pod nějakou konkrétní licencí. Proces popisovaný v této kapitole bude fungovat pro libovolný pythonovský software, nezávisle na licenci.

16.2. Věci, které za nás Distutils neudělají

Vypuštění vašeho prvního pythonovského balíčku je skličující proces. (Uvolnění vašeho druhého balíčku je o něco snazší.) Distutils se celý proces snaží zautomatizovat, jak jen to je možné. Ale některé věci prostě musíte udělat sami.

- **Vybrat licenci.** Tohle je komplikované téma, zatížené politikou a rizikem. Pokud svůj software chcete zveřejnit jako open source, skromně doporučuji pět následujících rad:
  1. Nepište svou vlastní licenci.
  2. Nepište svou vlastní licenci.
  3. Nepište svou vlastní licenci.
  4. Nemusí to být zrovna gpl, ale měla by být s `GPL` slučitelná.
  5. Nepište svou vlastní licenci.
- **Zařaďte svůj software** pomocí klasifikačního systému PyPI. Později v této kapitole vysvětlím, co to znamená.
- **Napište soubor „read me“ (čti mne).** Tohle neodflákněte. Vaši uživatelé by se z něj měli dozvědět přinejmenším to, co váš software dělá a jak se instaluje.

16.3. Struktura adresáře

Vytváření balíčku pro váš pythonovský software začíná tím, že si musíte udělat pořádek v souborech a v adresářích. Adresář `httplib2` vypadá takto:

```
httplib2/                       [1]
|
+--README.txt                   [2]
|
+--setup.py                     [3]
|
+--httplib2/                    [4]
   |
   +--__init__.py
   |
   +--iri2uri.py
```

[1] Vytvořte kořenový adresář, ve kterém bude všechno. Dejte mu stejné jméno, jaké má váš pythonovský modul.

[2] Abyste se přizpůsobili uživatelům Windows, měl by váš soubor „read me“ použít příponu `.txt` a měl by používat windowsovské konce řádků. To, že vy používáte nějaký fantastický editor, který se dá spouštět z příkazového řádku a má i svůj makro jazyk, neznamená, že byste měli ztěžovat život svým uživatelům. (Vaši uživatelé používají „Notepad“, česky „Poznámkový blok“. Je to smutné, ale je to tak.) Váš oblíbený editor má nepochybně volbu pro ukládání souborů s windowsovskými konci řádků — i když pracujete v Linuxu nebo s Mac OS X.

[3] Váš instalační skript využívající Distutils by měl být pojmenován `setup.py` — pokud nemáte dobrý důvod pro to, aby se jmenoval jinak. A vy nemáte dobrý důvod, aby se jmenoval jinak.

[4] Pokud se váš pythonovský software skládá z jediného souboru s příponou `.py`, měli byste jej umístit do kořenového adresáře spolu se svým souborem „read me“ a se svým instalačním skriptem. Ale `httplib2` se neskládá z jediného `.py` souboru. Je to vícesouborový modul. Ale to je v pořádku! Adresář `httplib2` umístěte do kořenového adresáře, takže budete mít soubor `__init__.py` umístěn v adresáři `httplib2/` v kořenovém adresáři `httplib2/`. Nehledejte v tom problém. Ve skutečnosti to zjednoduší proces vytváření balíčku.

Adresář `chardet` vypadá trochu jinak. Stejně jako u `httplib2` jde o vícesouborový modul, takže tu máme adresář `chardet/` uvnitř kořenového adresáře `chardet/`. K souboru `README.txt` má `chardet` navíc `HTML` dokumentaci, umístěnou v adresáři `docs/`. Adresář `docs/` obsahuje několik souborů s příponami `.html` a `.css` a podadresář `images/`, který obsahuje několik souborů s příponami `.png` a `.gif`. (To bude důležité později.) V souladu s konvencemi pro software s licencí `(L)GPL` obsahuje také samostatný soubor zvaný `COPYING.txt`, který obsahuje kompletní text `LGPL`.

```
chardet/
|
+--COPYING.txt
|
+--setup.py
|
+--README.txt
|
+--docs/
|  |
|  +--index.html
|  |
|  +--usage.html
|  |
|  +--images/ ...
|
+--chardet/
   |
   +--__init__.py
   |
   +--big5freq.py
   |
   +--...
```

16.4. Píšeme svůj instalační skript

Instalační skript pro Distutils je pythonovský skript. Teoreticky by mohl dělat vše, co lze dělat v pythonovských skriptech. Prakticky by toho měl dělat co nejméně a co nejstandardnějším způsobem. Instalační skript by měl být nudný. Čím exotičtější bude váš instalační proces, tím exotičtější budou hlášení o chybách.

První řádek každého instalačního skriptu pro Distutils je vždycky stejný:

```
from distutils.core import setup
```

Importujeme funkci `setup()`, která je hlavním vstupním bodem rámce Distutils. 95 % všech distutilsovských skriptů se skládá z jediného volání funkce `setup()` a z ničeho jiného. (Tuhle statistiku jsem si právě vymyslel, ale pokud váš distutilsovský skript dělá něco víc než volání funkce `setup()` z Distutils, měli byste pro to mít dobrý důvod. A máte pro to dobrý důvod? Myslím, že ne.)

Funkce `setup()` přebírá celou řadu parametrů. V zájmu zachování duševního zdraví všech zúčastněných musíte pro každý parametr používat pojmenované argumenty. Není to jen nějaká konvence.

Je to tvrdý požadavek. Pokud se pokusíte o volání funkce `setup()` s nepojmenovanými argumenty, váš instalační skript zhavaruje.

Následující pojmenované argumety jsou povinné:

- **name** — jméno balíčku.
- **version** — verze balíčku.
- **author** — vaše celé jméno.
- **author_email** — vaše e-mailová adresa.
- **url** — domácí stránka vašeho projektu. Pokud pro projekt nemáte vyhrazen zvláštní webový server, můžete zde uvést stránku svého balíčku v PyPI.

Ačkoliv to není povinné, doporučuji, abyste ve svém instalačním skriptu uvedli také následující:

- **description**, jednořádkový popis projektu.
- **long_description**, víceřádkový řetězec ve formátu reStructuredText. PyPI ho převede do `HTML` a zobrazí ho na stránce pro váš balíček.
- **classifiers**, seznam zvláštním způsobem formátovaných řetězců, které si popíšeme v následující podkapitole.

> Metadata instalačního skriptu jsou definována v `PEP 314`.

Teď se podíváme na instalační skript pro `chardet`. Používá všechny zmíněné povinné a doporučené parametry a ještě jeden, o kterém jsem se zatím nezmínil: `packages`.

```
from distutils.core import setup
setup(
    name = 'chardet',
    packages = ['chardet'],
    version = '1.0.2',
    description = 'Universal encoding detector',
    author='Mark Pilgrim',
    ...
)
```

Parametr `packages` zvýrazňuje jedno nešťastné překrývání významů slov během distribučního procesu. O slově „balíček/balík“ (package) jsme se bavili jako o něčem, co vytváříme (a co se potenciálně vypisuje v seznamu PyPI). Jenže to není tím, na co se parametr `packages` odkazuje. Vztahuje se ke skutečnosti, že `chardet` je vícesouborovým modulem, kterému se také někdy říká… „package“ (balíček). Parametr `packages` nástroji Distutils říká, aby do procesu zahrnul adresář `chardet/`, jeho soubor `__init__.py` a všechny ostatní soubory s příponou `.py`, ze kterých se modul `chardet` skládá. To je docela důležité. Veškerá radostná rozprava o dokumentaci a metadatech je k ničemu, pokud zapomenete přibalit skutečný kód!

16.5. Přidáváme klasifikaci našeho balíčku

The Python Package Index („PyPI“) obsahuje tisíce pythonovských knihoven. Ostatní lidé najdou váš balíček snadněji, když použijete správná klasifikační metadata. PyPI vám umožní prohlížet balíčky uspořádané podle klasifikátorů. Pro zúžení nabídky při vyhledávání můžete vybrat dokonce více klasifikátorů. Klasifikátory prostě nejsou jen neviditelná metadata, která byste mohli ignorovat!

Klasifikaci svého softwaru provedete předáním parametru `classifiers` distutilovské funkci `setup()`. Parametr `classifiers` má podobu seznamu řetězců. Ale tyto řetězce *nemají* volný formát. Všechny klasifikační řetězce by měly pocházet z tohoto seznamu na PyPI.

Klasifikátory jsou nepovinné. Můžete napsat distutilovský instalační skript bez jakýchkoliv klasifikátorů. **Nedělejte to**. Měli byste vždy uvést alespoň následující klasifikátory:

- **Programming Language** (programovací jazyk). Konkrétně by měl zahrnovat jak `"Programming Language :: Python"`, tak `"Programming Language :: Python :: 3"`. Pokud je neuvedete, nebude se váš balíček ukazovat v tomto seznamu knihoven kompatibilních s Pythonem 3, na který se dostanete přes odkaz uvedený v bočním sloupku na každé stránce z `pypi.python.org`.
- **License** (licence). Pokud zkouším nějakou knihovnu třetí strany, *je to absolutně první věc, na kterou se dívám*. Nechtějte po mně, abych tuto životně důležitou informaci musel někde hledat. Pokud se váš software nedodává pod více licencemi, neuvádějte víc než jeden licenční klasifikátor. (A pokud k tomu nejste nějak nuceni, nevydávejte svůj software pod více licencemi. A nenuťte k tomu ostatní lidi. Licencování stačí k bolení hlavy už i tak. Nedělejte to ještě horší.)
- **Operating System** (operační systém). Pokud váš software běží pouze pod Windows (nebo jen pod Mac OS X nebo jen pod Linuxem), rád bych se to dozvěděl raději dřív než později. Pokud váš software běží kdekoliv, aniž by potřeboval nějaký platformově závislý kód, použijte klasifikátor `"Operating System :: OS Independent"`. Použití více klasifikátorů `Operating System` je nezbytné pouze v případě, kdy váš software vyžaduje pro každou platformu specifickou podporu. (Běžně tomu tak nebývá.)

Doporučuji, abyste uvedli i následující klasifikátory:

- **Development Status** (stav vývoje). Lze kvalitu vašeho softwaru ohodnotit přívlastkem beta? Alfa? Nebo se nachází ještě v ranějším stadiu (pre-alpha)? Jednu z možností uveďte. Buďte upřímní.
- **Intended Audience** (zamýšlená skupina uživatelů). Kdo by mohl chtít stahovat váš software? Nejpoužívanější volby jsou `Developers` (vývojáři), `End Users/Desktop` (koncoví uživatelé), `Science/Research` (věda a výzkum), and `System Administrators` (správci systémů).
- **Framework** (rámec). Pokud lze váš software považovat za zásuvný modul (plugin) pro větší pythonovské rámce, jako jsou například Django nebo Zope, uveďte příslušný klasifikátor `Framework`. Pokud tomu tak není, neuvádějte jej.
- **Topic** (tematická oblast). Zde naleznete velké množství oblastí, ze kterých si můžete vybrat. Uveďte všechny, které vašemu softwaru odpovídají.

16.5.1. Příklady dobrých klasifikátorů balíčků

Jako příklad uveďme klasifikátory pro Django, což je multiplatformní aplikační rámec (framework), který můžete spouštět na svém webovém serveru. Dodává se pod licencí BSD a je využitelný pro ostrý provoz (production-ready). (Django zatím není kompatibilní s Pythonem 3, takže není uveden klasifikátor `Programming Language :: Python :: 3`.)

```
Programming Language :: Python
License :: OSI Approved :: BSD License
Operating System :: OS Independent
Development Status :: 5 - Production/Stable
Environment :: Web Environment
Framework :: Django
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Internet :: WWW/HTTP :: Dynamic Content
Topic :: Internet :: WWW/HTTP :: WSGI
Topic :: Software Development :: Libraries :: Python Modules
```

Tady jsou klasifikátory pro `chardet`, což je knihovna pro detekci znakového kódování, kterou jsme se
Kap. zabývali v Případové studii: Přepis `chardet` pro Python 3. `chardet` je ve stadiu beta, je multiplatformní, kompatibilní s Pythonem 3, pod licencí LGPL a je určena pro vývojáře, kteří ji mohou začlenit do svých vlastních produktů.

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: GNU Library or Lesser General Public License (LGPL)
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Other Environment
Intended Audience :: Developers
Topic :: Text Processing :: Linguistic
Topic :: Software Development :: Libraries :: Python Modules
```

Kap. A tady jsou klasifikátory pro `httplib2`, což je knihovna, o které jsme se bavili v kapitole Webové služby nad HTTP. `httplib2` je ve stadiu beta, multiplatformní, pod licencí MIT a je učena pro pythonovské vývojáře.

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: MIT License
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Web Environment
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Software Development :: Libraries :: Python Modules
```

16.6. Určení dalších souborů prostřednictvím manifestu

Pokud neurčíme jinak, zahrnou Distutils do vašeho instalačního balíčku následující soubory:

- `README.TXT`
- `setup.py`
- Soubory s příponou .py, které se používají ve vícesouborových modulech uvedených v seznamu parametru `packages`.
- Jednotlivé soubory s příponou `.py`, které jsou uvedeny v seznamu parametru `py_modules`.

Tímto způsobem lze pokrýt všechny soubory projektu `httplib2`. Ale u projektu chardet potřebujeme zařadit i licenční soubor `COPYING.txt` a celý adresář `docs/`, který obsahuje obrázky a `HTML` soubory. Pokud chceme Distutils říci, aby byly při tvorbě instalačního balíčku `chardet` zařazeny i tyto dodatečné soubory a adresáře, musíme použít *soubor s manifestem* (manifest file).

Soubor s manifestem je textový soubor s názvem `MANIFEST.in`. Umístíme jej do kořenového adresáře projektu, vedle souborů `README.txt` a `setup.py`. Soubory s manifestem *nejsou* pythonovské skripty. Jsou to textové soubory, které obsahují posloupnosti „příkazů“ ve formátu pro Distutils. Příkazy manifestu nám umožňují zahrnovat nebo vyřazovat konkrétní soubory a adresáře.

Následuje celý obsah souboru manifestu pro projekt `chardet`:

```
include COPYING.txt                                    [1]
recursive-include docs *.html *.css *.png *.gif        [2]
```

[1] První řádek je samovysvětlující: vložit soubor `COPYING.txt` z kořenového adresáře projektu.

[2] Druhý řádek je trochu složitější. Příkaz `recursive-include` přebírá jméno adresáře a jedno nebo víc jmen souborů. Jména souborů nemusí být uvedena explicitně. Mohou být vyjádřena zástupnými znaky (wildcards). Tento řádek znamená: „Vidíš adresář `docs/` v kořenovém adresáři projektu? Najdi v něm (rekurzivně) soubory s příponami `.html`, `.css`, `.png` a `.gif`. Chci, aby byly všechny zařazeny do instalačního balíčku.“

16.7. Kontrola chyb v našem instalačním skriptu
16.8. Vytvoření distribuce obsahující zdrojové texty

Všechny příkazy manifestu zachovávají strukturu adresářů, která je vytvořena v kořenovém adresáři projektu. Uvedený příkaz `recursive-include` nenacpe všechny `.html` a `.png` soubory do kořenového adresáře instalačního balíčku. Dodrží existující strukturu adresáře `docs/`, ale zařadí do ní jen ty soubory, které odpovídají zadaným maskám se zástupnými znaky. (Dříve jsem se o tom nezmiňoval, ale dokumentace `chardet` je ve skutečnosti napsaná v `XML` a do `HTML` je převedena samostatným skriptem. Do instalačního balíčku nechci zařazovat zdrojové `XML` soubory, ale jen výsledné `HTML` soubory a obrázky.)

> Soubory s manifestem mají svůj specifický formát. Detaily hledejte v dokumentech **Specifying the files to distribute** a **The manifest template commands**.

Zopakujme si to: soubor s manifestem musíme vytvářet jen v případě, kdy chceme zahrnout i soubory, které nástroj Distutils nevkládá automaticky. Pokud potřebujeme použít soubor s manifestem, měl by obsahovat jen jména souborů, která by jinak nástroj Distutils nenašel sám.

16.7. Kontrola chyb v našem instalačním skriptu

Musíme myslet na spoustu věcí. Distutils mají zabudovaný validační příkaz, který kontroluje, že náš instalační skript obsahuje všechna povinná metadata. Pokud například zapomeneme uvést parametr `version`, Distutils nám to připomenou.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check
running check
warning: check: missing required meta-data: version
```

Jakmile parametr `version` uvedeme (a všechny ostatní povinné části metadat), příkaz `check` dopadne takto:

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check
running check
```

16.8. Vytvoření distribuce obsahující zdrojové texty

Distutils podporují tvorbu mnoha typů distribučních balíčků. Přinejmenším bychom měli vytvořit „distribuci zdrojů“ (source distribution), která obsahuje naše zdrojové texty s kódem, instalační skript pro Distutils, soubor „read me“ a jakékoliv další soubory, které chceme do distribuce zahrnout. Distribuci zdrojů vytvoříme tím, že instalačnímu skriptu Distutils zadáme příkaz `sdist`.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py sdist
running sdist
running check
reading manifest template 'MANIFEST.in'
```

```
writing manifest file 'MANIFEST'
creating chardet-1.0.2
creating chardet-1.0.2\chardet
creating chardet-1.0.2\docs
creating chardet-1.0.2\docs\images
copying files to chardet-1.0.2...
copying COPYING -> chardet-1.0.2
copying README.txt -> chardet-1.0.2
copying setup.py -> chardet-1.0.2
copying chardet\__init__.py -> chardet-1.0.2\chardet
copying chardet\big5freq.py -> chardet-1.0.2\chardet
...
copying chardet\universaldetector.py -> chardet-1.0.2\chardet
copying chardet\utf8prober.py -> chardet-1.0.2\chardet
copying docs\faq.html -> chardet-1.0.2\docs
copying docs\history.html -> chardet-1.0.2\docs
copying docs\how-it-works.html -> chardet-1.0.2\docs
copying docs\index.html -> chardet-1.0.2\docs
copying docs\license.html -> chardet-1.0.2\docs
copying docs\supported-encodings.html -> chardet-1.0.2\docs
copying docs\usage.html -> chardet-1.0.2\docs
copying docs\images\caution.png -> chardet-1.0.2\docs\images
copying docs\images\important.png -> chardet-1.0.2\docs\images
copying docs\images\note.png -> chardet-1.0.2\docs\images
copying docs\images\permalink.gif -> chardet-1.0.2\docs\images
copying docs\images\tip.png -> chardet-1.0.2\docs\images
copying docs\images\warning.png -> chardet-1.0.2\docs\images
creating dist
creating 'dist\chardet-1.0.2.zip' and adding 'chardet-1.0.2' to it
adding 'chardet-1.0.2\COPYING'
adding 'chardet-1.0.2\PKG-INFO'
adding 'chardet-1.0.2\README.txt'
adding 'chardet-1.0.2\setup.py'
adding 'chardet-1.0.2\chardet\big5freq.py'
adding 'chardet-1.0.2\chardet\big5prober.py'
...
adding 'chardet-1.0.2\chardet\universaldetector.py'
adding 'chardet-1.0.2\chardet\utf8prober.py'
adding 'chardet-1.0.2\chardet\__init__.py'
adding 'chardet-1.0.2\docs\faq.html'
adding 'chardet-1.0.2\docs\history.html'
adding 'chardet-1.0.2\docs\how-it-works.html'
adding 'chardet-1.0.2\docs\index.html'
```

```
adding 'chardet-1.0.2\docs\license.html'
adding 'chardet-1.0.2\docs\supported-encodings.html'
adding 'chardet-1.0.2\docs\usage.html'
adding 'chardet-1.0.2\docs\images\caution.png'
adding 'chardet-1.0.2\docs\images\important.png'
adding 'chardet-1.0.2\docs\images\note.png'
adding 'chardet-1.0.2\docs\images\permalink.gif'
adding 'chardet-1.0.2\docs\images\tip.png'
adding 'chardet-1.0.2\docs\images\warning.png'
removing 'chardet-1.0.2' (and everything under it)
```

Tady bychom se měli zmínit o několika věcech:

- Distutils si všimly souboru s manifestem (`MANIFEST.in`).
- Distutils soubor s manifestem úspěšně zpracovaly a přidaly předepsané soubory — `COPYING.txt` a `HTML` soubory a soubory s obrázky v adresáři `docs/`.
- Pokud se podíváme do adresáře projektu, uvidíme, že Distutils vytvořily adresář `dist/`. V adresáři `dist/` se nachází soubor s příponou `.zip`, který můžeme distribuovat.

```
c:\Users\pilgrim\chardet> dir dist
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\chardet\dist

07/30/2009  06:29 PM    <DIR>          .
07/30/2009  06:29 PM    <DIR>          ..
07/30/2009  06:29 PM           206,440 chardet-1.0.2.zip
               1 File(s)        206,440 bytes
               2 Dir(s)  61,424,635,904 bytes free
```

16.9. Vytvoření grafického instalačního programu

Podle mého názoru si každá pythonovská knihovna zaslouží, aby byl pro uživatele Windows k dispozici grafický instalační program. Dá se udělat snadno (i když sami Windows nepoužíváte) a uživatelé Windows to ocení.

Distutils dovedou vytvořit grafický instalační program pro Windows za nás. Stačí, když instalačnímu skriptu pro Distutils zadáme příkaz `bdist_wininst`.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py bdist_wininst
running bdist_wininst
running build
running build_py
creating build
creating build\lib
creating build\lib\chardet
copying chardet\big5freq.py -> build\lib\chardet
copying chardet\big5prober.py -> build\lib\chardet
...
copying chardet\universaldetector.py -> build\lib\chardet
copying chardet\utf8prober.py -> build\lib\chardet
copying chardet\__init__.py -> build\lib\chardet
installing to build\bdist.win32\wininst
running install_lib
creating build\bdist.win32
creating build\bdist.win32\wininst
creating build\bdist.win32\wininst\PURELIB
creating build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\big5freq.py -> build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\big5prober.py -> build\bdist.win32\wininst\PURELIB\chardet
...
copying build\lib\chardet\universaldetector.py -> build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\utf8prober.py -> build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\__init__.py -> build\bdist.win32\wininst\PURELIB\chardet
running install_egg_info
Writing build\bdist.win32\wininst\PURELIB\chardet-1.0.2-py3.1.egg-info
creating 'c:\users\pilgrim\appdata\local\temp\tmp2f4h7e.zip' and adding '.' to it
adding 'PURELIB\chardet-1.0.2-py3.1.egg-info'
adding 'PURELIB\chardet\big5freq.py'
adding 'PURELIB\chardet\big5prober.py'
...
adding 'PURELIB\chardet\universaldetector.py'
adding 'PURELIB\chardet\utf8prober.py'
adding 'PURELIB\chardet\__init__.py'
removing 'build\bdist.win32\wininst' (and everything under it)
c:\Users\pilgrim\chardet> dir dist
c:\Users\pilgrim\chardet>dir dist
 Volume in drive C has no label.
 Volume Serial Number is AADE-E29F

 Directory of c:\Users\pilgrim\chardet\dist
```

```
07/30/2009  10:14 PM    <DIR>          .
07/30/2009  10:14 PM    <DIR>          ..
07/30/2009  10:14 PM           371,236 chardet-1.0.2.win32.exe
07/30/2009  06:29 PM           206,440 chardet-1.0.2.zip
               2 File(s)        577,676 bytes
               2 Dir(s)  61,424,070,656 bytes free
```

16.9.1. Tvorba instalačních balíčků pro jiné operační systémy

Distutils nám mohou pomoci vytvořit instalační balíčky pro uživatele Linuxu. Ale podle mého názoru to nestojí za tu námahu. Pokud chcete svůj software distribuovat v Linuxu, měli byste svůj čas raději věnovat spolupráci se skupinou lidí, kteří se specializují na vytváření softwarových balíčků pro hlavní distribuce Linuxu.

Například moji knihovnu `chardet` najdete v archivech pro Debian GNU/Linux (a tím pádem i v archivech pro Ubuntu). Nemusel jsem se o to vůbec starat. Balíčky se tam jednoho dne prostě objevily. Komunita kolem distribuce Debian má svá vlastní pravidla pro balení pythonovských knihoven a balíček `python-chardet` pro Debian je navržen tak, aby tyto konvence splňoval. A protože jsou balíčky umístěny v archivech Debianu, získávají uživatelé Debianu bezpečnostní aktualizace a/nebo nové verze podle toho, jaká systémová nastavení si pro údržbu svých počítačů zvolili.

Linuxovské balíčky vytvářené nástrojem Distutils žádnou z těchto výhod nenabízejí. Bude lepší, když svůj čas strávíte jiným způsobem.

16.10. Přidání našeho softwaru do Python Package Index

Nahrání našeho softwaru do Python Package Index představuje proces o třech krocích.

1. Zaregistrujeme se.
2. Zaregistrujeme svůj software.
3. Uložíme (upload) balíčky, které jsme vytvořili příkazy `setup.py sdist` a `setup.py bdist_*`.

Registraci své osoby provedeme prostřednictvím registrační stránky pro uživatele PyPI. Vložíme své uživatelské jméno a heslo, poskytneme platnou e-mailovou adresu a klikneme na tlačítko `Register`. (Pokud máte klíč `PGP` nebo `GPG`, můžete jej uvést také. Pokud jej nemáte nebo nevíte, co to znamená, nedělejte si s tím starosti.) Zkontrolujeme svůj e-mail. Během několika minut bychom měli obdržet zprávu od PyPI s potvrzovacím odkazem. Registrační proces dokončíme tím, že na odkaz klikneme.

Teď zaregistrujeme u PyPI náš software a nahrajeme jej (upload). To vše můžeme provést v jediném kroku.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist bdist_wininst upload [1]
running register
We need to know who you are, so please choose either:
 1. use your existing login,
 2. register as a new user,
 3. have the server generate a new password for you (and email it to you), or
 4. quit
Your selection [default 1]:  1                                                              [2]
Username: MarkPilgrim                                                                       [3]
Password:
Registering chardet to http://pypi.python.org/pypi                                          [4]
Server response (200): OK
running sdist                                                                               [5]
... výstup pro stručnost vypuštěn ...
running bdist_wininst                                                                       [6]
... výstup pro stručnost vypuštěn ...
running upload                                                                              [7]
Submitting dist\chardet-1.0.2.zip to http://pypi.python.org/pypi
Server response (200): OK
Submitting dist\chardet-1.0.2.win32.exe to http://pypi.python.org/pypi
Server response (200): OK
I can store your PyPI login so future submissions will be faster.
(the login will be stored in c:\home\.pypirc)
Save your login (y/N)?n                                                                     [8]
```

[1] Když svůj projekt zveřejníme poprvé, přidají Distutils náš software do Python Package Index a přidělí mu jeho vlastní URL. Při dalších přístupech jednoduše aktualizují metadata projektu podle změn, které uvedeme v parametrech našeho `setup.py`. Poté se vytvoří distribuce zdrojů (source distribution; `sdist`) a instalátor pro Windows (`bdist_wininst`) a nahrají se do PyPI (`upload`).

[2] Vybereme „use your existing login“ (použij svůj existující účet) napsáním `1`, nebo prostě stiskneme ENTER.

[3] Napíšeme uživatelské jméno a heslo, která jsme si zvolili na registrační stránce PyPI. Distutils neopisují zadávané heslo. Místo zadávaných znaků nevypisují ani hvězdičky. Prostě napíšeme heslo a stiskneme ENTER.

[4] Distutils zaregistrují náš balíček v archivu Python Package Index…

[5] …vytvoří distribuci našich zdrojů (source distribution)…

[6] …vytvoří instalátor pro Windows…

[7] …a nahrají (upload) oba do Python Package Index.

[8] Pokud chceme proces zveřejňování nových verzí zautomatizovat, musíme uložit osobní údaje pro PyPI do lokálního souboru. Je to zcela proti zásadám bezpečnosti a zcela nepovinné.

16.11. Více možných budoucností balení pythonovských produktů
16.12. Přečtěte si

Gratuluji. Teď už máte svoji vlastní stránku na Python Package Index!

Její adresa je `http://pypi.python.org/pypi/JMENO`, kde `JMENO` je řetězec, který jste předali parametrem `name` ve svém souboru `setup.py`.

Pokud chceme zveřejnit novou verzi, upravíme ve svém souboru `setup.py` číslo verze a spustíme proces nahrávání (upload) znovu:

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist bdist_wininst upload
```

16.11. Více možných budoucností balení pythonovských produktů

Distutils nejsou jediným nástrojem pro vytváření pythonovských balíčků, ale v době psaní tohoto textu (srpen 2009) to byl jediný rámec pro vytváření instalačních balíčků, který fungoval v Pythonu 3. Pro Python 2 existuje řada dalších rámců. Některé se soustředí na instalaci, jiné na testování a distribuci (deployment). Některé z nich možná budou přepsány pro Python 3.

Následující rámce (frameworks) jsou zaměřeny na instalaci:

- **Setuptools**
- **Pip**
- **Distribute**

Následující se zaměřují na testování a distribuci:

- `virtualenv`
- `zc.buildout`
- **Paver**
- **Fabric**
- `py2exe`

16.12. Přečtěte si

O Distutils:

- **Distributing Python Modules with Distutils** (http://docs.python.org/py3k/distutils/)
- **Core Distutils functionality** uvádí všechny možné argumenty funkce `setup()` (http://docs.python.org/py3k/distutils/apiref.html)
- **Distutils Cookbook** (http://wiki.python.org/moin/Distutils/Cookbook)

- **PEP 370: Per user `site-packages` directory**
  (www.python.org/dev/peps/pep-0370/)
- **PEP 370 and "environment stew"**
  (http://jessenoller.com/2009/07/19/pep-370-per-user-site-packages-and-environment-stew/)

O ostatních rámcích pro vytváření balíčků:

- **The Python packaging ecosystem**
  (http://groups.google.com/group/django-developers/msg/5407cdb400157259)
- **On packaging**
  (www.b-list.org/weblog/2008/dec/14/packaging/)
- **A few corrections to "On packaging"**
  (http://blog.ianbicking.org/2008/12/14/a-few-corrections-to-on-packaging/)
- **Why I like Pip**
  (www.b-list.org/weblog/2008/dec/15/pip/)
- **Python packaging: a few observations**
  (http://cournape.wordpress.com/2009/04/01/python-packaging-a-few-observations-cabal-for-a-solution/)
- **Nobody expects Python packaging!**
  (http://jacobian.org/writing/nobody-expects-python-packaging/)

**“Life is pleasant. Death is peaceful.
It’s the transition that’s troublesome.”**
(Život je zábavný. Smrt je klidná.
Nepříjemný je ten přechod.)

— Isaac Asimov (připsáno)

A. Přepis kódu do Pythonu 3 s využitím 2to3

A. Přepis kódu do Pythonu 3 s využitím 2to3 — 377
A.1. Ponořme se — **379**
A.2. Příkaz `print` — **379**
A.3. Literály Unicode řetězců — **380**
A.4. Globální funkce `unicode()` — **380**
A.5. Datový typ `long` — **380**
A.6. Porovnání <> — **381**
A.7. Slovníková metoda `has_key()` — **381**
A.8. Slovníkové metody, které vracejí seznamy — **382**
A.9. Moduly, které byly přejmenovány nebo reorganizovány — **383**
A.9.1. `http` — **383**
A.9.2. `urllib` — **384**
A.9.3. `dbm` — **385**
A.9.4. `xmlrpc` — **385**
A.9.5. Ostatní moduly — **386**
A.10. Relativní importy uvnitř balíčku — **387**
A.11. Metoda iterátoru `next()` — **388**
A.12. Globální funkce `filter()` — **388**
A.13. Globální funkce `map()` — **389**
A.14. Globální funkce `reduce()` — **390**
A.15. Globální funkce `apply()` — **390**
A.16. Globální funkce `intern()` — **390**
A.17. Příkaz `exec` — **391**
A.18. Příkaz `execfile` — **391**
A.19. `repr` literály (zpětné apostrofy) — **392**
A.20. Příkaz `try...except` — **392**
A.21. Příkaz `raise` — **393**
A.22. Metoda generátorů `throw` — **393**
A.23. Globální funkce `xrange()` — **394**
A.24. Globální funkce `raw_input()` a `input()` — **395**
A.25. Atributy funkcí `func_*` — **395**
A.26. Metoda `xreadlines()` V/V objektů — **396**
A.27. `lambda` funkce, které akceptují n-tici místo více parametrů — **396**
A.28. Atributy speciálních metod — **397**
A.29. Speciální metoda `__nonzero__` — **397**
A.30. Oktalové literály — **398**
A.31. `sys.maxint` — **398**
A.32. Globální funkce `callable()` — **399**
A.33. Globální funkce `zip()` — **399**
A.34. Výjimka `StandardError` — **399**
A.35. Konstanty modulu `types` — **400**
A.36. Globální funkce `isinstance()` — **400**
A.37. Datový typ `basestring` — **401**
A.38. `itertools` module — **401**
A.39. `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback` — **401**
A.40. Generátory seznamů nad n-ticemi — **402**
A.41. Funkce `os.getcwdu()` — **402**
A.42. Metatřídy — **402**
A.43. Věci týkající se stylu — **403**
A.43.1. Množinové literály (`set()`; explicitně) — **403**
A.43.2. Globální funkce `buffer()` (explicitně) — **403**
A.43.3. Bílé znaky kolem čárek (explicitně) — **404**
A.43.4. Běžné obraty (explicitně) — **404**

A.1. Ponořme se
A.2. Příkaz print

A.1. Ponořme se

Mezi Pythonem 2 a Pythonem 3 se toho změnilo tolik, že najdete jen mizivé procento programů, které bez úprav běží v obou verzích. Ale nepropadejte zoufalství! K usnadnění přechodu se Python 3 dodává s pomocným skriptem nazvaným `2to3`. Když mu předáte svůj zdrojový soubor napsaný pro Python 2 jako vstup, převede automaticky do podoby pro Python 3 vše, co dovede. Případová studie: Přepis `chardet` pro Python 3 popisuje, jak se skript `2to3` spouští. Ukazuje také věci, které se automaticky neopraví. V této příloze najdete dokumentaci toho, co dovede opravit automaticky.

Kap.

A.2. Příkaz `print`

V Pythonu 2 byl `print` příkazem. Pokud jsme cokoliv chtěli vytisknout, jednoduše jsme to připsali za klíčové slovo `print`. V Pythonu 3 je `print()` funkcí. Pokud chceme cokoliv vytisknout, předáme to funkci `print()` stejně jako každé jiné funkci.

Poznámky	Python 2	Python 3
[1]	`print print()`	
[2]	`print 1`	`print(1)`
[3]	`print 1, 2`	`print(1, 2)`
[4]	`print 1, 2,`	`print(1, 2, end=' ')`
[5]	`print >>sys.stderr, 1, 2, 3`	`print(1, 2, 3, file=sys.stderr)`

[1] Prázdný řádek vytiskneme voláním `print()` bez zadání argumentů.

[2] Jednu hodnotu vytiskneme voláním `print()` s jedním argumentem.

[3] Dvě hodnoty oddělené mezerou vytiskneme voláním `print()` s dvěma argumenty.

[4] V tomhle je malá finta. Pokud jsme v Pythonu 2 ukončili příkaz `print` čárkou, vytiskly se hodnoty oddělené mezerou, pak se vytiskla ještě jedna koncová mezera a tisk skončil bez generování přechodu na nový řádek. (Z technického hlediska je to o něco komplikovanější. Příkaz `print` v Pythonu 2 používal nyní již nežádoucí (deprecated) atribut zvaný `softspace`. Místo skutečného tisku mezery nastavil Python 2 `sys.stdout.softspace` na 1. Znak mezery ve skutečnosti nebyl vytištěn, dokud se nemělo na stejný řádek tisknout něco dalšího. Pokud další příkaz `print` tiskl přechod na nový řádek, byl atribut `sys.stdout.softspace` nastaven na `0` a mezera se nikdy nevytiskla. Tohoto rozdílu byste si pravděpodobně nikdy nevšimli, pokud by vaše aplikace nebyla citlivá na přítomnost nebo nepřítomnost koncových bílých znaků ve výstupu, který byl vygenerován příkazem `print`.) V Pythonu 3 dosáhneme stejného efektu tím, že funkci `print()` předáme pojmenovaný argument s hodnotou `end=' '`. Výchozí hodnotou argumentu `end` je `'\n'` (přechod na nový řádek), takže po vytisknutí ostatních argumentů jeho přepsáním potlačíme přechod na nový řádek.

[5] V Pythonu 2 jsme mohli výstup přesměrovat do roury (pipe) — například na `sys.stderr` — uvedením zápisu `>>jméno_roury`. V Pythonu 3 dosáhneme stejného efektu předáním odkazu na rouru pojmenovaným argumentem `file`. Výchozí hodnotou argumentu `file` je `sys.stdout` (standardní výstup), takže přepsáním této hodnoty dosáhneme přesměrování do jiné roury.

A.3. Literály Unicode řetězců
A.4. Globální funkce unicode()
A.5. Datový typ long

A.3. Literály Unicode řetězců

Python 2 pracoval s dvěma typy řetězců: s Unicode řetězci a s ne-Unicode řetězci. Python 3 podporuje jediný řetězcový typ: Unicode řetězce.

Poznámky	Python 2	Python 3
[1]	`u'PapayaWhip'`	`'PapayaWhip'`
[2]	`ur'PapayaWhip\foo'`	`r'PapayaWhip\foo'`

[1] Řetězcové literály s prefixem Unicode jsou jednoduše převedeny na obyčejné řetězcové literály, které v Pythonu 3 vždy vyjadřují Unicode řetězce.

[2] Surové Unicode řetězce (raw; ve kterých Python neprovádí interpretaci zpětného lomítka jako zahájení escape posloupnosti) jsou převedeny na surové řetězce. V Pythonu 3 jsou surové řetězce vždy v Unicode.

A.4. Globální funkce `unicode()`

V Pythonu 2 se pro převod objektů na řetězec používaly dvě globální funkce: `unicode()` pro převod na Unicode řetězce a `str()` pro převod na ne-Unicode řetězce. Python 3 má jediný řetězcový typ, Unicode řetězce, takže vše, co potřebujeme, je funkce `str()`. (Funkce `unicode()` už neexistuje.)

Poznámky	Python 2	Python 3
	`unicode(cokoliv)`	`str(cokoliv)`

A.5. Datový typ `long`

Python 2 používal pro celá čísla dva datové typy: `int` a `long`. Hodnota typu `int` nemohla být větší než konstanta `sys.maxint`, která byla závislá na platformě. „Dlouhá“ čísla byla definována přidáním `L` na konec čísla a mohla nabývat větších hodnot než čísla typu `int`. V Pythonu 3 je jen jeden celočíselný typ, který se jmenuje `int` a většinou se chová jako typ `long` v Pythonu 2. Protože už neexistují dva typy, nemusí se používat speciální syntaxe pro jejich rozlišení.

Přečtěte si:
PEP 237: Unifying Long Integers and Integers.
(www.python.org/dev/peps/pep-0237/)

A.6. Porovnání <>
A.7. Slovníková metoda has_key()

Poznámky	Python 2	Python 3
[1]	`x = 1000000000000L`	`x = 1000000000000`
[2]	`x = 0xFFFFFFFFFFFFL`	`x = 0xFFFFFFFFFFFF`
[3]	`long(x)`	`int(x)`
[4]	`type(x) is long`	`type(x) is int`
[5]	`isinstance(x, long)`	`isinstance(x, int)`

[1] Z desítkových číselných literálů pro „dlouhý“ integer (long) se staly desítkové literály pro typ integer.

[2] Z šestnáctkových číselných literálů pro „dlouhý“ integer (long) se staly šestnáctkové literály pro typ integer.

[3] V Pythonu 3 přestala existovat původní funkce `long()`, protože přestal existovat typ long (dlouhý integer). K převodu proměnné na celé číslo použijeme funkci `int()`.

[4] Pokud chceme zkontrolovat, zda je proměnná typu integer, zjistíme její typ a porovnáváme ho s `int` (nikoliv s `long`).

[5] Ke kontrole datového typu můžeme použít i funkci `isinstance()`. Při zjišťování, zda jde o celočíselný typ, se opět odkážeme na `int` a ne na `long`.

A.6. Porovnání <>

Python 2 podporoval operátor <> jako synonymum pro != (porovnání na různost). Python 3 podporuje pouze operátor != a přestal podporovat <>.

Poznámky	Python 2	Python 3
[1]	`if x <> y:`	`if x != y:`
[2]	`if x <> y <> z:`	`if x != y != z:`

[1] Jednoduché porovnání.

[2] Složitější porovnání mezi třemi hodnotami.

A.7. Slovníková metoda has_key()

V Pythonu 2 používaly slovníky metodu `has_key()` (doslova „má klíč“) pro testování, zda se ve slovníku nachází zadaný klíč. V Pythonu 3 tato metoda přestala existovat. Místo ní musíme používat operátor `in`.

Poznámky	Python 2	Python 3
[1]	`a_dictionary.has_key('PapayaWhip')`	`'PapayaWhip' in a_dictionary`
[2]	`a_dictionary.has_key(x) or a_dictionary.has_key(y)`	`x in a_dictionary or y in a_dictionary`
[3]	`a_dictionary.has_key(x or y)`	`(x or y) in a_dictionary`
[4]	`a_dictionary.has_key(x + y)`	`(x + y) in a_dictionary`
[5]	`x + a_dictionary.has_key(y)`	`x + (y in a_dictionary)`

[1] Nejjednodušší forma.

[2] Operátor `in` má vyšší prioritu než operátor `or`, takže podvýrazy `x in a_dictionary` a `y in a_dictionary` nemusíme uzavírat do závorek.

[3] Ale na druhou stranu zde ze stejného důvodu *musíme* uzavřít do závorek `x or y` — `in` má vyšší prioritu než `or`. (Poznámka: Tento kód se od předchozího řádku zcela liší. Python interpretuje nejdříve `x or y`. Výsledkem je buď `x` (pokud se `x` interpretuje v booleovském kontextu jako true), nebo `y`. Potom pro výslednou hodnotu kontroluje, zda se ve slovníku `a_dictionary` vyskytuje jako klíč.)

[4] Operátor `+` má vyšší prioritu než operátor `in`. Z technického hlediska by tento zápis nemusel používat závorky kolem `x + y`, ale `2to3` je stejně přidává.

[5] U tohoto zápisu musí být kolem `y in a_dictionary` závorky určitě uvedeny, protože operátor `+` má vyšší prioritu než operátor `in`.

A.8. Slovníkové metody, které vracejí seznamy

V Pythonu 2 mnohé slovníkové metody vracely seznamy. Mezi nejpoužívanější metody patřily `keys()`, `items()` a `values()`. V Pythonu 3 všechny tyto metody vracejí dynamické pohledy (view). V některých situacích to nečiní žádný problém. Pokud je návratová hodnota těchto metod ihned předána jiné funkci, která iteruje přes celou posloupnost, bude jedno, zda je skutečným typem seznam nebo pohled (view). V jiném kontextu to ale může mít velký vliv. Pokud očekáváme kompletní seznam s jednotlivě adresovatelnými prvky, náš kód se zakucká, protože pohledy nepodporují indexování (tj. zpřístupňování prvku přes index).

Poznámky	Python 2	Python 3
[1]	`a_dictionary.keys()`	`list(a_dictionary.keys())`
[2]	`a_dictionary.items()`	`list(a_dictionary.items())`
[3]	`a_dictionary.iterkeys()`	`iter(a_dictionary.keys())`
[4]	`[i for i in a_dictionary.iterkeys()]`	`[i for i in a_dictionary.keys()]`
[5]	`min(a_dictionary.keys())`	*žádná změna*

[1] Skript `2to3` se přiklání k bezpečnému řešení. Voláním funkce `list()` převádí hodnotu vracenou metodou `keys()` na statický seznam. Bude to fungovat vždycky, ale někdy to bude méně

efektivní než použití pohledu (view). Převedený kód byste si měli prohlédnout a zvážit, zda je statický seznam nezbytně nutný, nebo zda by nestačil pohled.

[2] Další konverze pohledu na seznam — tentokrát u metody `items()`. Stejnou věc provede `2to3` s metodou `values()`.

[3] Python 3 už nepodporuje metodu `iterkeys()`. Použijte `keys()`, a pokud je to nezbytné, udělejte z pohledu iterátor voláním funkce `iter()`.

[4] `2to3` pozná, když je metoda `iterkeys()` použita uvnitř generátorové notace seznamu. Převede ji na metodu `keys()` (neobaluje ji ještě jedním voláním `iter()`). Funguje to, protože přes pohledy (view) lze iterovat.

[5] `2to3` pozná případ, kdy je metoda `keys()` předána funkci, která iteruje celou posloupností. V takovém případě se návratová hodnota nemusí konvertovat na seznam. Funkce `min()` bude vesele iterovat i přes pohled. Týká se to funkcí `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()` a `all()`.

A.9. Moduly, které byly přejmenovány nebo reorganizovány

Několik modulů standardní pythonovské knihovny bylo přejmenováno. Několik vzájemně souvisejících modulů bylo spojeno dohromady nebo bylo reorganizováno tak, aby byly jejich vztahy logičtější.

A.9.1. `http`

V Pythonu 3 bylo několik modulů souvisejících s `HTTP` spojeno do jednoho balíku nazvaného `http`.

Poznámky	Python 2	Python 3
[1]	`import httplib`	`import http.client`
[2]	`import Cookie`	`import http.cookies`
[3]	`import cookielib`	`import http.cookiejar`

[4]	`import BaseHTTPServer`
	`import SimpleHTTPServer`
	`import CGIHttpServer`
		`import http.server`

[1] Modul `http.client` implementuje nízkoúrovňovou knihovnu, která vytváří požadavky na `HTTP` zdroje a interpretuje související `HTTP` odpovědi.

[2] Modul `http.cookies` poskytuje pythonovské rozhraní pro cookies prohlížeče, které se posílají v `HTTP` hlavičce `HTTP` hlavička.

[3] Modul `HTTP.COOKIEJAR` manipuluje se soubory na disku, které oblíbené webové prohlížeče používají k ukládání cookies.

[4] Modul `http.server` implementuje jednoduchý `HTTP` server.

A.9.2. urllib

Python 2 obsahoval změť překrývajících se modulů pro rozklad (parse) a kódování URL a pro získávání příslušného obsahu. V Pythonu 3 byly moduly refaktorizovány a sloučeny do jednoho balíku urllib.

Poznámky	Python 2	Python 3
[1]	`import urllib`	`import urllib.request, urllib.parse, urllib.error`
[2]	`import urllib2`	`import urllib.request, urllib.error`
[3]	`import urlparse`	`import urllib.parse`
[4]	`import robotparser`	`import urllib.robotparser`
[5]	`from urllib import FancyURLopener` `from urllib import urlencode`	`from urllib.request import FancyURLopener` `from urllib.parse import urlencode`
[6]	`from urllib2 import Request` `from urllib2 import HTTPError`	`from urllib.request import Request` `from urllib.error import HTTPError`

[1] Starý modul urllib v Pythonu 2 obsahoval řadu funkcí včetně urlopen() pro načítání dat a splittype(), splithost() a splituser() pro rozklad URL na podstatné části. Uvnitř nového balíku urllib byly tyto funkce logičtěji přeorganizovány. Skript 2to3 také změní všechna volání těchto funkcí, aby zohlednil nové schéma pojmenování.

[2] Původní modul urllib2 z Pythonu 2 byl v Pythonu 3 vložen do balíčku urllib. Všechny oblíbené věci z urllib2 — metoda build_opener(), třídy Request a HTTPBasicAuthHandler a související věci — jsou stále k dispozici.

[3] Modul urllib.parse z Pythonu 3 obsahuje všechny funkce z původního modulu urlparse z Pythonu 2.

[4] Modul urllib.robotparser zpracovává soubory robots.txt.

[5] Třída FancyURLopener, která obsluhuje HTTP přesměrování a další stavové kódy, je v novém modulu urllib.request stále k dispozici. Funkce urlencode() se přesunula do urllib.parse.

[6] Třída Request je v urllib.request stále k dispozici, ale konstanty jako HTTPError byly přesunuty do urllib.error.

Zmínil jsem se o tom, že 2to3 přepíše také volání vašich funkcí? Pokud například v kódu pro Python 2 importujete modul urllib a získáváte data voláním urllib.urlopen(), skript 2to3 opraví jak příkaz import, tak volání funkce.

Python 2	Python 3
`import urllib`	`import urllib.request, urllib.parse, urllib.error`
`print urllib.urlopen ('http://diveintopython3.org/').read()`	`print(urllib.request.urlopen('http://diveintopython3. org/').read())`

A.9.3. dbm

Všechny klony DBM se nyní nacházejí jediném balíku dbm. Pokud potřebujeme použít nějakou specifickou variantu, jako například GNU DBM, můžeme importovat příslušný modul z balíku dbm.

Python 2	Python 3
`import dbm`	`import dbm.ndbm`
`import gdbm`	`import dbm.gnu`
`import dbhash`	`import dbm.bsd`
`import dumbdbm`	`import dbm.dumb`
`import anydbm` `import whichdb`	`import dbm`

A.9.4. xmlrpc

XML-RPC je odlehčená (lightweight) metoda pro provádění RPC (vzdálené volání procedur) přes HTTP. Klientská knihovna pro XML-RPC a několik implementací XML-RPC serveru jsou nyní zkombinovány do jednoho balíčku xmlrpc.

Python 2	Python 3
`import xmlrpclib`	`import xmlrpc.client`
`import DocXMLRPCServer` `import SimpleXMLRPCServer`	`import xmlrpc.server`

A.9.5. Ostatní moduly

Poznámky	Python 2	Python 3

[1]	`try:`
	`  import cStringIO as StringIO`
	`except ImportError:`
	`  import StringIO`
		`import io`

[2]	`try:`
	`  import cPickle as pickle`
	`except ImportError:`
	`  import pickle`
		`import pickle`

[3]	`import __builtin__`	`import builtins`
[4]	`import copy_reg`	`import copyreg`
[5]	`import Queue`	`import queue`
[6]	`import SocketServer`	`import socketserver`
[7]	`import ConfigParser`	`import configparser`
[8]	`import repr`	`import reprlib`
[9]	`import commands`	`import subprocess`

[1] Mezi běžné obraty v Pythonu 2 patřil pokus o `import cStringIO as StringIO`. Pokud operace selhala, provedl se místo toho příkaz `import StringIO`. V Pythonu 3 už to nedělejte. Modul `io` to udělá za vás. Nalezne nejrychlejší dostupnou implementaci a použije ji automaticky.

[2] Podobný obrat se používal pro importování nejrychlejší implementace `pickle`. V Pythonu 3 už to nedělejte. Modul `pickle` to udělá za vás.

[3] Modul `builtins` obsahuje globální funkce, třídy a konstanty, které se používají napříč celým jazykem Python. Redefinicí funkce v modulu `builtins` provedete redefinici globální funkce úplně všude. Je to přesně tak mocné a děsivé, jak to zní.
Modul `copyreg` přidává podporu „piklení“ pro uživatelské typy definované v C.
Modul `queue` implementuje frontu pro více producentů a více konzumentů.
Modul `socketserver` poskytuje obecné (generické) bázové třídy pro implementaci různých druhů soketových serverů.
Modul `configparser` zpracovává konfigurační soubory ve stylu `INI`.
Modul `reprlib` reimplementuje zabudovanou funkci `repr()` s přidaným ovládáním. Lze předepsat, jak dlouhé mohou reprezentace být, než dojde k jejich ořezání.
Modul `subprocess` umožňuje vytvářet procesy, připojovat se k jejich rourám (pipe) a získávat jejich návratové kódy.

A.10. Relativní importy uvnitř balíčku

Balíček je skupina souvisejících modulů, které se používají jako celek. Pokud se v Pythonu 2 moduly uvnitř balíčku potřebovaly odkazovat jeden na druhý, používali jsme příkaz `import foo` nebo `from foo import Bar`. V Pythonu 2 interpret hledal `foo.py` nejdříve uvnitř aktuálního balíčku a teprve potom prohledával ostatní adresáře z pythonovské vyhledávací cesty (`sys.path`). Python 3 funguje trošku jinak. Místo prohledávání aktuálního balíčku začne přímo pythonovskou vyhledávací cestou. Pokud chceme, aby jeden modul uvnitř balíčku importoval jiný modul ze stejného balíčku, musíme explicitně zadat relativní cestu mezi uvedenými moduly.

Dejme tomu, že bychom měli následující balíček s více soubory ve stejném adresáři:

```
chardet/
|
+--__init__.py
|
+--constants.py
|
+--mbcharsetprober.py
|
+--universaldetector.py
```

Teď předpokládejme, že `universaldetector.py` potřebuje importovat celý soubor `constants.py` a jednu třídu z `mbcharsetprober.py`. Jak to vlastně uděláme?

Poznámky	Python 2	Python 3
[1]	`import constants`	`from . import constants`
[2]	`from mbcharsetprober import MultiByteCharSetProber`	`from .mbcharsetprober import MultiByteCharsetProber`

[1] Pokud potřebujeme importovat celý modul odněkud z našeho balíčku, použijeme novou syntaxi `from . import`. Tečka ve skutečnosti označuje relativní cestu od tohoto souboru (`universaldetector.py`) k souboru, který chceme importovat (`constants.py`). V tomto případě se nacházejí ve stejném adresáři, takže použijeme jednu tečku. Importovat můžeme i z rodičovského adresáře (`from .. import jinymodul`) nebo z podadresáře.

[2] Pokud chceme importovat určitou třídu nebo funkci z jiného modulu přímo do prostoru jmen našeho modulu, přidáme k cílovému modulu jako prefix relativní cestu bez koncového lomítka. V tomto případě se `mbcharsetprober.py` nachází ve stejném adresáři jako `universaldetector.py`, takže cestu vyjádříme jednou tečkou. Importovat můžeme i z rodičovského adresáře (`from ..jinymodul import JinaTrida`) nebo z podadresáře.

A.11. Metoda iterátoru next()
A.12. Globální funkce filter()

A.11. Metoda iterátoru `next()`

V Pythonu 2 měly iterátory metodu `next()`, která vracela další položku z posloupnosti. V Pythonu 3 to stále platí, ale máme k dispozici také globální funkci `next()`, která přebírá iterátor jako argument.

Poznámky	Python 2	Python 3
[1]	`anIterator.next()`	`next(anIterator)`
[2]	`funkce_ktera_vraci_iterator().next()`	`next(funkce_ktera_vraci_iterator())`

[3]	`class A:`
	`  def next(self):`
	`  pass`
		`class A:`
		`  def __next__(self):`
		`  pass`

[4]	`class A:`
	`  def next(self, x, y):`
	`  pass`
		žádná změna

[5]	`next = 42`
	`for an_iterator in a_sequence_of_iterators:`
	`  an_iterator.next()`
		`next = 42`
		`for an_iterator in a_sequence_of_iterators:`
		`  an_iterator.__next__()`

[1] V nejjednodušším případě nyní místo volání metody iterátoru `next()` předáváme iterátor globální funkci `next()`.

[2] Pokud máme funkci, která vrací iterátor, zavoláme ji a výsledek předáme funkci `next()`. (Skript `2to3` je dost chytrý na to, aby to převedl správně.)

[3] Pokud definujeme svou vlastní třídu a míníme ji použít jako iterátor, definujeme speciální metodu `__next__()`.

[4] Pokud definujeme svou vlastní třídu a ta shodou okolností obsahuje metodu pojmenovanou `next()`, která přebírá jeden nebo víc argumentů, nechá ji skript `2to3` beze změny. Tato třída nemůže být použita jako iterátor, protože její metoda `next()` vyžaduje argumenty.

[5] Tohle je trošku ošemetné. Pokud máme lokální proměnnou pojmenovanou next, pak bude mít přednost před novou globální funkcí `next()`. V takovém případě budeme muset pro získání dalšího prvku posloupnosti volat speciální metodu iterátoru `__next__()`. (Alternativně bychom mohli refaktorizovat kód tak, že by lokální proměnná nebyla pojmenována `next`, ale to za nás `2to3` automaticky neudělá.)

A.12. Globální funkce `filter()`

V Pythonu 2 vracela funkce `filter()` seznam, který byl výsledkem filtrování posloupnosti přes funkci, která pro každý prvek posloupnosti vracela hodnotu True nebo False. V Pythonu 3 funkce `filter()` nevrací seznam, ale iterátor.

Poznámky	Python 2	Python 3
[1]	`filter(a_function, a_sequence)`	`list(filter(a_function, a_sequence))`
[2]	`list(filter(a_function, a_sequence))`	*žádná změna*
[3]	`filter(None, a_sequence)`	`[i for i in a_sequence if i]`
[4]	`for i in filter(None, a_sequence):`	*žádná změna*
[5]	`[i for i in filter(a_function, a_sequence)]`	*žádná změna*

[1] V nejzákladnějším případě obalí skript `2to3` volání funkce `filter()` voláním funkce `list()`. Tím se provede průchod přes všechny hodnoty a vrátí se skutečný seznam.

[2] Pokud je ale volání funkce `filter()` *už obaleno* v `list()`, nebude `2to3` dělat nic, protože skutečnost, že `filter()` vrací iterátor v takovém případě není důležitá.

[3] Speciální syntaxi `filter(None, ...)` skript `2to3` nahradí použitím sémanticky shodné generátorové notace seznamu.

[4] V kontextu podobajícímu se cyklům `for`, kdy stejně dochází k průchodu celou posloupností, není nutné provádět žádné změny.

[5] Ani zde se nemusí dělat žádné změny, protože generátorová notace seznamu bude iterovat přes všechny prvky posloupnosti, a to může udělat, ať už `filter()` vrací iterátor nebo seznam.

A.13. Globální funkce `map()`

Funkce `map()` nyní vrací iterátor. Jde o stejný případ jako u funkce `filter()`. (V Pythonu 2 se vracel seznam.)

Poznámky	Python 2	Python 3
[1]	`map(a_function, 'PapayaWhip')`	`list(map(a_function, 'PapayaWhip'))`
[2]	`map(None, 'PapayaWhip')`	`list('PapayaWhip')`
[3]	`map(lambda x: x+1, range(42))`	`[x+1 for x in range(42)]`
[4]	`for i in map(a_function, a_sequence):`	*žádná změna*
[5]	`[i for i in map(a_function, a_sequence)]`	*žádná změna*

[1] Stejně jako u `filter()` v nejzákladnějším případě obalí skript `2to3` volání funkce `map()` voláním `list()`.

[2] Speciální syntaxi `map(None, ...)`, vyjadřující funkci identity, převede skript `2to3` na ekvivalentní volání `list()`.

[3] Pokud je prvním argumentem `map()` lambda funkce, převede `2to3` zápis s využitím odpovídající generátorové notace seznamu.

[4] V kontextu jako u cyklů `for`, které stejně procházejí celou posloupností, není nutné provádět změny.

[5] Ani zde se nemusí dělat žádné změny, protože generátorová notace seznamu předepisuje průchod přes všechny prvky posloupnosti, a to může udělat, ať už map() vrací iterátor nebo seznam.

A.14. Globální funkce reduce()
A.15. Globální funkce apply()
A.16. Globální funkce intern()

A.14. Globální funkce reduce()

V Pythonu 3 byla funkce reduce() vyňata z globálního prostoru jmen a umístěna do modulu functools.

Python 2	Python 3
`reduce(a, b, c)`	`from functools import reduce` `reduce(a, b, c)`

A.15. Globální funkce apply()

V Pythonu 2 existovala globální funkce apply(), která přebírala funkci f a seznam [a, b, c] a vrátila f(a, b, c). Stejné věci můžeme dosáhnout tím, že funkci zavoláme přímo a před předávaný seznam argumentů připíšeme hvězdičku. V Pythonu 3 již funkce apply() neexistuje. Musíme použít zápis s hvězdičkou.

Poznámky	Python 2	Python 3
[1]	`apply(a_function, a_list_of_args)`	`a_function(*a_list_of_args)`
[2]	`apply(a_function, a_list_of_args,` `a_dictionary_of_named_args)`	`a_function(*a_list_of_args,` `**a_dictionary_of_named_args)`
[3]	`apply(a_function, a_list_of_args + z)`	`a_function(*a_list_of_args + z)`
[4]	`apply(aModule.a_function, a_list_of_args)`	`aModule.a_function(*a_list_of_args)`

[1] V nejjednodušším případě můžeme funkci při volání předat seznam argumentů (skutečný seznam, jako například [a, b, c]) přidáním hvězdičky před seznam (*). Jde o přesný ekvivalent staré funkce apply() z Pythonu 2.

[2] V Pythonu 2 může funkce apply() ve skutečnosti přebírat tři parametry: funkci, seznam argumentů a slovník s pojmenovanými argumenty. V Pythonu 3 můžeme téhož dosáhnout přidáním hvězdičky před seznam argumentů (*) a přidáním dvou hvězdiček před slovník pojmenovaných argumentů (**).

[3] Zde se operátor + používá pro zřetězení seznamů. Operátor + má vyšší prioritu než operátor *, takže kolem a_list_of_args + z nemusíme přidávat závorky.

[4] Skript 2to3 je dost chytrý na to, aby převedl i složitá volání apply(), včetně volání funkcí z importovaných modulů.

A.16. Globální funkce intern()

V Pythonu 2 bylo možné „internovat“ řetězec voláním funkce intern(), čímž došlo k optimalizaci výkonu při práci s tímto řetězcem. V Pythonu 3 byla funkce intern() přesunuta do modulu sys.

Python 2	Python 3
`intern(aString)`	`sys.intern(aString)`

A.17. Příkaz `exec`

Příkaz `exec` se v Pythonu 3 změnil na funkci stejně, jako se na funkci změnil příkaz `print`. Funkce `exec()` přebírá řetězec, který obsahuje libovolný pythonovský kód, a provede jej, jako kdyby to byl nějaký příkaz nebo výraz. Funkce `exec()` se podobá `eval()`, ale je ještě mocnější a zlověstnější. Funkce `eval()` může vyhodnocovat jediný výraz, ale funkce `exec()` může provést více příkazů, importů, deklarací funkcí — v podstatě celý pythonovský program, předaný jako řetězec.

Poznámky	Python 2	Python 3
[1]	`exec codeString`	`exec(codeString)`
[2]	`exec codeString in a_global_namespace`	`exec(codeString, a_global_namespace)`
[3]	`exec codeString in a_global_namespace, a_local_namespace`	`exec(codeString, a_global_namespace, a_local_namespace)`

[1] V nejjednodušším případě skript `2to3` prostě uzavře kód v podobě řetězce do závorek, protože `exec()` je teď funkce a ne příkaz.

[2] Původní příkaz `exec` mohl přebírat prostor jmen v podobě soukromého prostředí s globálními jmény, ve kterém se měl kód v podobě řetězce provádět. V Pythonu 3 lze dělat totéž. Prostor jmen se funkci `exec()` jednoduše předá jako druhý parametr.

[3] Původní příkaz `exec` umožňoval dokonce přebírat lokální prostor jmen (podobající se prostoru proměnných definovaných uvnitř nějaké funkce). V Pythonu 3 to funkce `exec()` dokáže také.

A.18. Příkaz `execfile`

Původní příkaz `execfile`, podobně jako původní příkaz `exec`, spouštěl řetězce, ve kterých byl uložen pythonovský kód. Tam, kde `exec` přebíral řetězec, `execfile` přebíral jméno souboru. Z Pythonu 3 byl příkaz `execfile` vyřazen. Pokud opravdu chcete použít soubor s pythonovským kódem a spustit jej (ale nechcete jej přitom jednoduše importovat), můžete stejné funkčnosti dosáhnout otevřením souboru, načtením jeho obsahu, zavoláním globální funkce `compile()` (aby byl pythonovský interpret donucen kód přeložit) a nakonec zavoláním nové funkce `exec()`.

Python 2	Python 3
`execfile('a_filename')`	`exec(compile(open('a_filename').read(), 'a_filename', 'exec'))`

A.19 repr-literály (zpětné apostrofy)

V Pythonu 2 bylo možné získat reprezentaci objektu použitím speciální syntaxe, kdy se libovolný objekt obalil zpětnými apostrofy (backticks; jako například `` `x` ``). V Pythonu 3 tato schopnost stále existuje, ale už ji nemůžeme vyvolat použitím zpětných apostrofů. Místo nich musíme použít globální funkci `repr()`.

Poznámky	Python 2	Python 3
[1]	`` `x` ``	`repr(x)`
[2]	`` ``'PapayaWhip' + `2`` ``	`repr('PapayaWhip' + repr(2))`

[1] Připomeňme si, že x může být cokoliv — třída, funkce, modul, primitivní datový typ atd. Funkce `repr()` funguje na všechno.

[2] V Pythonu 2 mohly být zpětné apostrofy zanořeny, což vedlo k tomuto druhu matoucích (ale platných) výrazů. Skript `2to3` je dost chytrý na to, aby zápis převedl na zanořené volání `repr()`.

A.20. Příkaz try...except

Syntaxe pro odchytávání výjimek se mezi verzemi Python 2 a Python 3 mírně změnila.

Poznámky	Python 2	Python 3

[1]	`try:`
	`  import mymodule`
	`except ImportError, e`
	`  pass`
		`try:`
		`  import mymodule`
		`except ImportError as e:`
		`  pass`

[2]	`try:`
	`  import mymodule`
	`except (RuntimeError, ImportError), e`
	`  pass`
		`try:`
		`  import mymodule`
		`except (RuntimeError, ImportError) as e:`
		`  pass`

[3]	`try:`
	`  import mymodule`
	`except ImportError:`
	`  pass`
		žádná změna

[4]	`try:`
	`  import mymodule`
	`except:`
	`  pass`
		žádná změna

[1] Místo čárky se za typem výjimky v Pythonu 3 používá nové klíčové slovo as.
[2] Klíčové slovo as funguje i pro odchytávání více typů výjimek najednou.
[3] Pokud výjimku jen odchytíme, ale ve skutečnosti nás nezajímá možnost přistupování k samotnému objektu výjimky, pak se syntaxe používaná v Pythonu 2 shoduje se syntaxí v Pythonu 3.
[4] Podobně, pokud používáme záchranu v podobě odchytávání *všech* výjimek, je syntaxe identická.

> Nouzové odchytávání všech výjimek byste nikdy neměli používat při importování modulů (ani ve většině ostatních případů). Tímto způsobem odchytíte i věci jako `KeyboardInterrupt` (pokud se uživatel pokoušel o přerušení činnosti programu stiskem `Ctrl-C`) a ztížíte si tím ladění.

A.21. Příkaz `raise`

Syntaxe pro vyvolávání našich vlastních výjimek se mezi verzemi Python 2 a Python 3 mírně změnila.

Poznámky	Python 2	Python 3
[1]	`raise MyException`	*žádná změna*
[2]	`raise MyException, 'error message'`	`raise MyException('error message')`
[3]	`raise MyException, 'error message', a_traceback`	`raise MyException('error message').with_traceback(a_traceback)`
[4]	`raise 'error message'`	*nepodporováno*

[1] Při použití nejjednodušší formy, vyvolání výjimky bez uživatelské chybové zprávy, se syntaxe nezměnila.
[2] Změny si povšimneme, když chceme vyvolat výjimku s uživatelským chybovým hlášením. Python 2 odděloval třídu výjimky a uživatelskou zprávu čárkou. Python 3 předává chybovou zprávu jako parametr.
[3] Python 2 podporoval při složitější syntaxi vyvolání výjimky s uživatelským zpětným trasováním (stack trace). V Pythonu 3 toho můžeme dosáhnout také, ale syntaxe se docela liší.
[4] V Pythonu 2 jsme mohli vyvolat výjimku, aniž jsme zadávali třídu výjimky. Stačilo zadat chybovou zprávu. V Pythonu 3 to již není možné. Skript `2to3` vás bude varovat, že nebyl schopen tuto situaci opravit automaticky.

A.22. Metoda generátorů `throw`

V Pythonu 2 definovaly generátory metodu `throw()`. Volání `a_generator.throw()` vyvolá výjimku v místě, kde se generátor zastavil. Potom se vrací další hodnota, která je vyprodukována (yield) generátorovou funkcí. V Pythonu 3 je uvedená funkčnost stále k dispozici, ale syntaxe se trochu změnila.

Poznámky	Python 2	Python 3
[1]	`a_generator.throw(MyException)`	*žádná změna*
[2]	`a_generator.throw(MyException, 'error message')`	`a_generator.throw(MyException('error message'))`
[3]	`a_generator.throw('error message')`	*nepodporováno*

[1] V nejjednodušším případě generátor vyvolává výjimku bez uživatelské chybové zprávy. V tomto případě se syntaxe v Pythonu 3 vůči Pythonu 2 nezměnila.

[2] Pokud generátor vyvolává výjimku *s uživatelskou chybovou zprávou*, musíme řetězec se zprávou předat vytvářenému objektu výjimky.

[3] Python 2 podporoval vyvolání výjimky, která byla tvořena pouze uživatelským chybovým hlášením. Python 3 toto chování nepodporuje a skript `2to3` zobrazí varování, které říká, že to budete muset opravit ručně.

A.23. Globální funkce `xrange()`

V Pythonu 2 existovaly dva způsoby získávání hodnot intervalu čísel: funkce `range()` vracela seznam a funkce `xrange()`, vracela iterátor. V Pythonu 3 funkce `range()` vrací iterátor a funkce `xrange()` už neexistuje.

Poznámky	Python 2	Python 3
[1]	`xrange(10)`	`range(10)`
[2]	`a_list = range(10)`	`a_list = list(range(10))`
[3]	`[i for i in xrange(10)]`	`[i for i in range(10)]`
[4]	`for i in range(10):`	*žádná změna*
[5]	`sum(range(10))`	*žádná změna*

[1] V nejjednodušším případě skript `2to3` jednoduše změní `xrange()` na `range()`.

[2] Pokud kód pro Python 2 používal `range()`, pak skript `2to3` neví, zda jsme skutečně potřebovali seznam, nebo zda by vyhověl iterátor. V rámci opatrnosti se vracená hodnota převádí na seznam voláním funkce `list()`.

[3] Pokud by byla funkce `xrange()` použita uvnitř generátorového zápisu seznamu, pak je skript `2to3` dost chytrý na to, aby funkci `range()` *neobalil* voláním `list()`. Generátorový zápis seznamu bude bez problémů fungovat s iterátorem, který je funkcí `range()` vrácen.

[4] Bez problémů bude s iterátorem fungovat i cyklus `for`, takže ani zde není nutné nic měnit.

[5] Funkce `sum()` pracuje s iterátorem také, takže `2to3` nemusí nic měnit ani zde. Uvedený přístup se, stejně jako v případě metod slovníku, které vracejí pohledy (view) místo seznamů, aplikuje i u funkcí `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()` a `all()`.

A.24. Globální funkce raw_input() a input()
A.25. Atributy funkcí func_*

A.24. Globální funkce raw_input() a input()

Python 2 poskytoval pro vyžádání si uživatelského vstupu z příkazové řádky dvě globální funkce. První z nich, zvaná input(), očekávala, že uživatel vloží pythonovský výraz (vrací se jeho výsledek). Druhá z nich, zvaná raw_input(), vracela to, co uživatel napsal. Začátečníky to velmi mátlo a považovalo se to za „bradavici“ (wart) na jazyce. Python 3 tuto nepěknost řeší přejmenováním raw_input() na input(), takže to funguje způsobem, který většina naivně očekává.

Poznámky	Python 2	Python 3
[1]	`raw_input()`	`input()`
[2]	`raw_input('prompt')`	`input('prompt')`
[3]	`input()`	`eval(input())`

[1] V nejjednodušším případě se raw_input() mění na input().

[2] V Pythonu 2 mohla funkce raw_input() přebírat vyzývací řetězec jako parametr. Tato možnost je zachována i v Pythonu 3.

[3] Pokud chcete, aby se opravdu vyhodnocoval pythonovský výraz zadaný uživatelem, použijte funkci input() a předejte její výsledek funkci eval().

A.25. Atributy funkcí func_*

V Pythonu 2 může kód uvnitř funkce přistupovat ke speciálním atributům, které se týkají funkce samotné. V Pythonu 3 byly tyto speciální atributy funkcí přejmenovány, aby se dostaly do souladu s ostatními atributy.

Poznámky	Python 2	Python 3
[1]	`a_function.func_name`	`a_function.__name__`
[2]	`a_function.func_doc`	`a_function.__doc__`
[3]	`a_function.func_defaults`	`a_function.__defaults__`
[4]	`a_function.func_dict`	`a_function.__dict__`
[5]	`a_function.func_closure`	`a_function.__closure__`
[6]	`a_function.func_globals`	`a_function.__globals__`
[7]	`a_function.func_code`	`a_function.__code__`

[1] Atribut __name__ (dříve func_name) obsahuje jméno funkce.

[2] Atribut __doc__ (dříve func_doc) obsahuje *dokumentační řetězec*, který byl definován ve zdrojovém textu funkce.

[3] Atribut __defaults__ (dříve func_defaults) je n-tice obsahující výchozí hodnoty argumentů pro ty z argumentů, pro které byly výchozí hodnoty definovány.

A.26. Metoda xreadlines() V/V objektů
A.27. lambda funkce, které akceptují n-tici místo více parametrů

[4] Atribut __dict__ (dříve func_dict) je prostor jmen uchovávající libovolné atributy funkce.
[5] Atribut __closure__ (dříve func_closure) je n-tice buněk, které obsahují vazby (bindings) na volné proměnné, které se ve funkci používají.
[6] Atribut __globals__ (dříve func_globals) je odkaz na globální prostor jmen modulu, ve kterém byla funkce definována.
[7] Atribut __code__ (dříve func_code) je objekt kódu (code object), reprezentující přeložené tělo funkce.

A.26. Metoda xreadlines() V/V objektů

V Pythonu 2 měly souborové objekty metodu xreadlines(), která vracela iterátor procházející souborem po řádcích. Kromě jiného se to hodilo pro cykly for. Ve skutečnosti to byla tak užitečná metoda, že pozdější verze Pythonu 2 přidaly schopnost iterovat samotným souborovým objektům.

V Pythonu 3 přestala metoda xreadlines() existovat. Skript 2to3 je schopen převést jednoduché případy, ale v hraničních situacích po vás bude vyžadovat ruční zásah.

Poznámky	Python 2	Python 3
[1]	`for line in a_file.xreadlines():`	`for line in a_file:`
[2]	`for line in a_file.xreadlines(5):`	*žádná změna (vede k nefunkčnímu kódu)*

[1] Pokud jste byli zvyklí volat xreadlines() bez argumentů, převede toto volání skript 2to3 jen na souborový objekt. V Pythonu 3 zajistí tento zápis stejnou funkčnost: čte se ze souboru řádek po řádku a provádí se tělo cyklu for.
[2] Pokud jste byli zvyklí volat xreadlines() s argumentem (počet řádků, které se mají načíst najednou), pak to skript 2to3 neopraví a váš kód selže s vysvětlením AttributeError: '_io.TextIOWrapper' object has no attribute 'xreadlines'. Opravu pro Python 3 můžete ručně provést změnou xreadlines() na readlines(). (Metoda readlines() teď vrací iterátor, takže je to stejně efektivní, jako bylo xreadlines() v Pythonu 2.)

A.27. lambda funkce, které akceptují n-tici místo více parametrů

V Pythonu 2 jsme mohli definovat anonymní lambda funkci, která přebírá více parametrů, tím, že jsme ji definovali jako funkci, která přebírá n-tici s určeným počtem položek. V důsledku toho Python 2 „rozbalil“ n-tici do pojmenovaných argumentů, na které jsme se pak mohli uvnitř lambda funkce odkazovat jménem. V Pythonu 3 můžeme lambda funkci také předávat n-tici, ale pythonovský interpret ji nerozbalí do pojmenovaných argumentů. Místo toho se budeme muset na jednotlivé argumenty odkazovat pozičním indexem.

Poznámky	Python 2	Python 3
[1]	`lambda (x,): x + f(x)`	`lambda x1: x1[0] + f(x1[0])`
[2]	`lambda (x, y): x + f(y)`	`lambda x_y: x_y[0] + f(x_y[1])`
[3]	`lambda (x, (y, z)): x + y + z`	`lambda x_y_z: x_y_z[0] + x_y_z[1][0] + x_y_z[1][1]`
[4]	`lambda x, y, z: x + y + z`	*žádná změna*

[1] Pokud jsme definovali `lambda` funkci, která přebírá n-tici s jedním prvkem, stane se z ní v Pythonu 3 `lambda` funkce, která se odkazuje na `x1[0]`. Jméno `x1` je generováno skriptem `2to3` automaticky, na základě pojmenovaných argumentů původní n-tice.

[2] lambda funkce s dvouprvkovou n-ticí (`x, y`) bude převedena na `x_y` s pozičními argumenty `x_y[0]` a `x_y[1]`.

[3] Skript `2to3` zvládne dokonce `lambda` funkce s vnořenými n-ticemi pojmenovaných argumentů. Výsledný kód v Pythonu 3 je poněkud nečitelný, ale funguje stejným způsobem, jakým fungoval původní kód v Pythonu 2.

[4] Můžeme definovat `lambda` funkce, které přebírají víc argumentů. Pokud kolem argumentů neuvedeme závorky, chová se Python 2 k zápisu jako k `lambda` funkci s více argumenty. Uvnitř lambda funkce se na pojmenované argumenty odkazujeme jménem jako v každé jiné funkci. V Pythonu 3 tato syntaxe pořád funguje.

A.28. Atributy speciálních metod

V Pythonu 2 se mohly metody tříd odkazovat na objekt třídy, ve které jsou definovány, a také na samotný objekt metody. Reference `im_self` odkazovala na objekt instance třídy, `im_func` na objekt funkce a `im_class` se odkazuje na třídu objektu `im_self`. V Pythonu 3 byly tyto speciální atributy metod přejmenovány, aby se dostaly do souladu s pojmenováním ostatních atributů.

Python 2	Python 3
`aClassInstance.aClassMethod.im_func`	`aClassInstance.aClassMethod.__func__`
`aClassInstance.aClassMethod.im_self`	`aClassInstance.aClassMethod.__self__`
`aClassInstance.aClassMethod.im_class`	`aClassInstance.aClassMethod.__self__.__class__`

A.29. Speciální metoda `__nonzero__`

V Pythonu 2 jsme mohli vytvářet své vlastní třídy, které se daly používat v booleovském kontextu. Mohli jsme například vytvořit instanci takové třídy a pak ji použít v příkazu `if`. Dělalo se to tak, že jsme definovali speciální metodu `__nonzero__()`, která vracela `True` nebo `False`. Ta se volala, kdykoliv byla instance použita v booleovském kontextu. V Pythonu 3 lze dělat totéž, ale jméno metody bylo změněno na `__bool__()`.

Poznámky	Python 2	Python 3

[1]	`class A:`
	`  def __nonzero__(self):`
	`  pass`
		`class A:`
		`  def __bool__(self):`
		`  pass`

[2]	`class A:`
	`  def __nonzero__(self, x, y):`
	`  pass`
		žádná změna

[1] Při vyhodnocování instance v booleovském kontextu se v Pythonu 3 místo `__nonzero__()` volá metoda `__bool__()`.

[2] Pokud ale máme definovánu metodu `__nonzero__()`, která vyžaduje nějaké argumenty, bude nástroj `2to3` předpokládat, že jsme ji používali pro nějaký jiný účel, a neprovede žádné změny.

A.30. Oktalové literály

Syntaxe pro zápis čísel v osmičkové soustavě (tj. oktalových) se mezi Pythonem 2 a Pythonem 3 mírně změnila.

Python 2	Python 3
`x = 0755`	`x = 0o755`

A.31. sys.maxint

V souvislosti se sloučením typů `long` a `int` pozbyla konstanta `sys.maxint` vypovídací přesnost. Tato hodnota může být stále užitečná při zjišťování schopností závislých na platformě. Proto byla v Pythonu ponechána, ale byla přejmenována na `sys.maxsize`.

Poznámky	Python 2	Python 3
[1]	`from sys import maxint`	`from sys import maxsize`
[2]	`a_function(sys.maxint)`	`a_function(sys.maxsize)`

[1] Z maxint se stává `maxsize`.

[2] Jakékoliv použití `sys.maxint` se mění na `sys.maxsize`.

A.32. Globální funkce `callable()`
A.33. Globální funkce `zip()`
A.34. Výjimka `StandardError`

A.32. Globální funkce `callable()`

V Pythonu 2 jsme mohli voláním globální funkce `callable()` zkontrolovat, zda se dá objekt volat (jako funkce). Z Pythonu 3 byla tato globální funkce vyřazena. Pokud chceme zjistit, zda se dá objekt volat, musíme zkontrolovat, zda má speciální metodu `__call__()`.

Python 2	Python 3
`callable(anything)`	`hasattr(anything, '__call__')`

A.33. Globální funkce `zip()`

V Pythonu 2 přebírala globální funkce `zip()` libovolný počet posloupností a vracela seznam n-tic. První n-tice obsahovala první položky ze všech posloupností, druhá n-tice obsahovala druhé položky ze všech posloupností a tak dále. V Pythonu 3 vrací funkce `zip()` místo seznamu iterátor.

Poznámky	Python 2	Python 3
[1]	`zip(a, b, c)`	`list(zip(a, b, c))`
[2]	`d.join(zip(a, b, c))`	*žádná změna*

[1] Nejjednodušší způsob dosažení původního chování funkce `zip()` spočívá v obalení návratové hodnoty voláním `list()`. Tím dojde k průchodu všemi hodnotami iterátoru vraceného funkcí `zip()` a vytvoří se skutečný seznam výsledků.

[2] V kontextu, kde se již využívá iterace přes všechny položky posloupnosti (jako například při volání této metody `join()`), funguje iterátor vracený funkcí `zip()` bez problémů. Skript `2to3` je dost chytrý na to, aby takové případy detekoval a neprováděl ve vašem kódu žádné změny.

A.34. Výjimka `StandardError`

V Pythonu 2 byla `StandardError` bázovou třídou všech zabudovaných výjimek — až na `StopIteration`, `GeneratorExit`, `KeyboardInterrupt` a `SystemExit`. V Pythonu 3 byla třída `StandardError` zrušena. Místo ní se používá třída `Exception`.

Python 2	Python 3
`x = StandardError()`	`x = Exception()`
`x = StandardError(a, b, c)`	`x = Exception(a, b, c)`

A.35. Konstanty modulu types
A.36. Globální funkce isinstance()

A.35. Konstanty modulu `types`

Modul `types` obsahuje širokou paletu konstant, které nám pomáhají určovat typ objektu. V Pythonu 2 obsahoval konstanty pro všechny primitivní typy, jako jsou `dict` a `int`. Z Pythonu 3 byly tyto konstanty odstraněny. Místo nich se používá jméno primitivního typu.

Python 2	Python 3
`types.UnicodeType`	`str`
`types.StringType`	`bytes`
`types.DictType`	`dict`
`types.IntType`	`int`
`types.LongType`	`int`
`types.ListType`	`list`
`types.NoneType`	`type(None)`
`types.BooleanType`	`bool`
`types.BufferType`	`memoryview`
`types.ClassType`	`type`
`types.ComplexType`	`complex`
`types.EllipsisType`	`type(Ellipsis)`
`types.FloatType`	`float`
`types.ObjectType`	`object`
`types.NotImplementedType`	`type(NotImplemented)`
`types.SliceType`	`slice`
`types.TupleType`	`tuple`
`types.TypeType`	`type`
`types.XRangeType`	`range`

> `types.StringType` se převádí na `bytes` a ne na `str`, protože „řetězec“ v Pythonu 2 (ne Unicode řetězec, ale obyčejný řetězec) je ve skutečnosti jen posloupností bajtů odpovídajících určitému znakovému kódování.

A.36. Globální funkce `isinstance()`

Funkce `isinstance()` kontroluje, zda je objekt instancí určité třídy nebo typu. V Pythonu 2 jsme mohli předat n-tici typů a `isinstance()` vrátila `True`, pokud byl objekt jedním z uvedených typů. V Pythonu 3 lze dělat totéž, ale předávání stejného typu dvakrát se považuje za nežádoucí (deprecated).

Python 2	Python 3
`isinstance(x, (int, float, int))`	`isinstance(x, (int, float))`

A.37. Datový typ basestring
A.38. itertools module
A.39. sys.exc_type, sys.exc_value, sys.exc_traceback

A.37. Datový typ basestring

Python 2 pracoval s dvěma typy řetězců: Unicode a ne-Unicode. Ale existoval v něm ještě jeden typ, basestring. Jednalo se o abstraktní typ, nadtřídu jak pro typ str, tak pro typ unicode. Nebylo možné ji volat nebo z ní vytvářet instanci přímo, ale mohli jste ji předat globální funkci isinstance(), když jste chtěli zkontrolovat, zda je objekt buď Unicode, nebo ne-Unicode řetězcem. V Pythonu 3 existuje jediný řetězcový typ, takže důvod k existenci typu basestring pominul.

Python 2	Python 3
`isinstance(x, basestring)`	`isinstance(x, str)`

A.38. itertools module

Python 2.3 zavedl modul itertools, který definoval varianty globálních funkcí zip(), map() a filter(), které místo seznamu vracely iterátory. V Pythonu 3 tyto globální funkce vracejí iterátory, takže uvedené funkce byly z modulu itertools odstraněny. (V modulu itertools je stále mnoho užitečných funkcí, nejen ty právě zmíněné.)

Poznámky	Python 2	Python 3
[1]	`itertools.izip(a, b)`	`zip(a, b)`
[2]	`itertools.imap(a, b)`	`map(a, b)`
[3]	`itertools.ifilter(a, b)`	`filter(a, b)`
[4]	`from itertools import imap, izip, foo`	`from itertools import foo`

[1] Místo itertools.izip() použijte jednoduše globální funkci zip().

[2] Místo itertools.imap() použijte jednoduše map().

[3] Z itertools.ifilter() se stává filter().

[4] Modul itertools v Pythonu 3 pořád existuje. Jen v něm chybí funkce, které byly přesunuty do globálního prostoru jmen. Skript 2to3 je dost chytrý na to, aby odstranil importy, které neexistují, a ponechal ostatní importy nedotčené.

A.39. sys.exc_type, sys.exc_value, sys.exc_traceback

U Pythonu 2 se v modulu sys nacházely tři proměnné, které jsme mohli používat během obsluhy výjimky: sys.exc_type, sys.exc_value, sys.exc_traceback. (Ve skutečnosti mají původ už v Pythonu 1.) Už od Pythonu 1.5 bylo používání těchto proměnných považováno za nežádoucí (deprecated) ve prospěch sys.exc_info(), což je funkce vracející n-tici se všemi třemi hodnotami. V Pythonu 3 byly tyto tři individuální proměnné nakonec odstraněny. Musíme používat funkci sys.exc_info().

A.40. Generátory seznamů nad n-ticemi
A.41. Funkce os.getcwdu()
A.42. Metatřídy

Python 2	Python 3
`sys.exc_type`	`sys.exc_info()[0]`
`sys.exc_value`	`sys.exc_info()[1]`
`sys.exc_traceback`	`sys.exc_info()[2]`

A.40. Generátory seznamů nad n-ticemi

Pokud jsme v Pythonu 2 chtěli použít generátorovou notaci seznamu, která předepisovala iteraci přes n-tici, nemuseli jsme hodnoty n-tice uzavírat do kulatých závorek. V Pythonu 3 se explicitní závorky vyžadují.

Python 2	Python 3
`[i for i in 1, 2]`	`[i for i in (1, 2)]`

A.41. Funkce `os.getcwdu()`

V Pythonu 2 byla k dispozici funkce pojmenovaná `os.getcwd()`, která vracela aktuální pracovní adresář jako (ne-Unicode) řetězec. Protože moderní souborové systémy umí pracovat se jmény adresářů v libovolném znakovém kódování, zavedl Python 2.3 funkci `os.getcwdu()`. Funkce `os.getcwdu()` vracela aktuální pracovní adresář jako Unicode řetězec. V Pythonu 3 existuje jediný řetězcový typ (Unicode), takže `os.getcwd()` je vším, co potřebujeme.

Python 2	Python 3
`os.getcwdu()`	`os.getcwd()`

A.42. Metatřídy

V Pythonu 2 jsme mohli metatřídy vytvářet buď definicí argumentu `metaclass` v deklaraci třídy, nebo definicí speciálního atributu `__metaclass__` na úrovni třídy. V Pythonu 3 byl tento atribut třídy odstraněn.

Poznámky	Python 2	Python 3
[1]	`class C(metaclass=PapayaMeta):` `  pass`	*žádná změna*
[2]	`class Whip:` `  __metaclass__ = PapayaMeta`	`class Whip(metaclass=PapayaMeta):` `  pass`
[3]	`class C(Whipper, Beater):` `  __metaclass__ = PapayaMeta`	`class C(Whipper, Beater, metaclass=PapayaMeta):` `  pass`

[1] Deklarace metatřídy v místě deklarace třídy fungovala v Pythonu 2 a funguje stejně i v Pythonu 3.
[2] Deklarace metatřídy pomocí atributu třídy fungovala v Pythonu 2, ale v Pythonu 3 již ne.
[3] Skript `2to3` je dost chytrý na to, aby zkonstruoval platnou deklaraci třídy dokonce i v případech, kdy třída dědí z jedné nebo více bázových tříd.

A.43. Věci týkající se stylu

Zbytek zde popsaných „oprav“ ve skutečnosti nejsou opravy jako takové. Tyto úpravy nemění podstatu, ale styl. Jde o věci, které fungují jak v Pythonu 2, tak v Pythonu 3. Vývojáři Pythonu ale mají zájem na tom, aby byl pythonovský kód tak jednotný, jak je to jen možné. Z tohoto pohledu existuje oficiální **Python style guide** (Průvodce stylem jazyka Python), který popisuje — až do nesnesitelnosti — všechny možné detaily, které vás téměř určitě nezajímají. A když už `2to3` vytváří tak mohutnou infrastrukturu pro konverzi pythonovského kódu z jedné podoby do druhé, vzali si autoři za své přidat pár nepovinných rysů, které by zlepšily čitelnost vašich pythonovských programů.

A.43.1. Množinové literály (`set()`; explicitně)

V Pythonu 2 bylo jediným možným vyjádřením definice množinového literálu volání `set(posloupnost)`. V Pythonu 3 tato možnost stále funguje, ale čistší způsob spočívá v použití nového zápisu množinového literálu: složené závorky. Funguje to pro všechny množiny s výjimkou prázdné množiny. Je to tím, že slovníky používají složené závorky také a zápis `{}` byl již vyhrazen pro prázdný slovník a ne pro prázdnou množinu.

> Skript `2to3` standardně množinové literály zapsané pomocí `set()` neupravuje. Pokud chceme tuto úpravu povolit, uvedeme při volání `2to3` na příkazovém řádku `-f set_literal` (f jako fix).

Před	**Po**
`set([1, 2, 3])`	`{1, 2, 3}`
`set((1, 2, 3))`	`{1, 2, 3}`
`set([i for i in a_sequence])`	`{i for i in a_sequence}`

A.43.2. Globální funkce `buffer()` (explicitně)

Pythonovské objekty implementované v jazyce C exportují takzvané „rozhraní bloku paměti“ (buffer interface), které umožňuje ostatnímu pythonovskému kódu přímo číst blok paměti a zapisovat do něj. (Je to přesně tak mocné a děsivé, jak to zní.) V Pythonu 3 byla funkce `buffer()` přejmenována na `memoryview()`. (Ve skutečnosti je to sice o něco komplikovanější, ale rozdíly můžete téměř určitě ignorovat.)

> Skript `2to3` standardně funkci `buffer()` neopravuje. Pokud chceme tuto úpravu povolit, uvedeme při volání `2to3` na příkazovém řádku `-f buffer`.

Před	Po
`x = buffer(y)`	`x = memoryview(y)`

A.43.3. Bílé znaky kolem čárek (explicitně)

Navzdory drakonickým pravidlům pro používání bílých znaků (whitespace) při odsazování a předsazování se Python chová docela volně k používání bílých znaků v jiných oblastech. Uvnitř seznamů, n-tic, množin a slovníků se mohou bílé znaky objevit před a za čárkami bez škodlivých účinků. Jenže Průvodce stylem jazyka Python říká, že před čárkami se nemá psát žádná mezera a za čárkou se má psát jedna. Ačkoliv se zde jedná o čistě estetickou záležitost (kód funguje tak jako tak, v Pythonu 2 i v Pythonu 3), skript `2to3` tuto věc může volitelně opravit.

> Skript `2to3` standardně psaní bílých znaků kolem čárek neupravuje. Pokud chceme tuto úpravu povolit, uvedeme při volání `2to3` na příkazovém řádku `-f wscomma`.

Před	Po
`a ,b`	`a, b`
`{a :b}`	`{a: b}`

A.43.3. Běžné obraty (explicitně)

V pythonovské komunitě postupně vznikla celá řada používaných obratů. Některé se datují až k Pythonu 1, jako například cyklus `while 1:`. (Až do verze 2.3 neměl Python opravdový booleovský typ, takže vývojáři místo pravdivostních hodnot používali 1 a 0.) Moderní pythonovští programátoři by své mozky měli natrénovat na modernější podobu takových obratů.

> Skript `2to3` standardně opravu běžných obratů neprovádí. Pokud chceme tuto úpravu povolit, uvedeme při volání `2to3` na příkazovém řádku `-f idioms`.

Před	Po
`while 1:` `  do_stuff()`	`while True:` `  do_stuff()`
`type(x) == T`	`isinstance(x, T)`
`type(x) is T`	`isinstance(x, T)`

`a_list = list(a_sequence)`
`a_list.sort()`
`do_stuff(a_list)`
	`a_list = sorted(a_sequence)`
	`do_stuff(a_list)`

“My specialty is being right when other people are wrong.”
(Mou specialitou je mít pravdu, když se ostatní lidé mýlí.)

— George Bernard Shaw

B. Jména speciálních metod

B. Jména speciálních metod — 405
B.1. Ponořme se — **407**
B.2. Základy — **407**
B.3. Třídy, které se chovají jako iterátory — **407**
B.4. Vypočítávané atributy — **408**
B.5. Třídy, které se chovají jako funkce — **411**
B.6. Třídy, které se chovají jako množiny — **412**
B.7. Třídy, které se chovají jako slovníky — **413**
B.8. Třídy, které se chovají jako čísla — **414**
B.9. Třídy, které se dají porovnávat — **417**
B.10. Třídy, které podporují serializaci — **418**
B.11. Třídy, které mohou být použity v bloku `with` — **418**
B.12. Opravdu esoterické věci — **420**
B.13. Přečtěte si — **420**

B.1. Ponořme se
B.2. Základy
B.3. Třídy, které se chovají jako iterátory

B.1. Ponořme se

V celé knize jsme se setkávali s příklady „speciálních metod“ — v jistém smyslu „magických“ metod, které Python vyvolává, když použijeme určitou syntaxi. Pokud vaše třídy použijí speciální metody, mohou se chovat jako množiny, jako slovníky, jako funkce, jako iterátory nebo dokonce jako čísla. Tato příloha slouží jako referenční příručka ke speciálním metodám, se kterými jsme se už setkali, a jako stručný úvod k některým esoteričtějším speciálním metodám.

B.2. Základy

Pokud jste už četli úvod k třídám, už jste se setkali s nejběžnější speciální metodou, s metodou `__init__()`. Většina tříd, které píšeme, nakonec potřebuje nějakou inicializaci. Existuje několik dalších základních speciálních metod, které jsou zvlášť užitečné při ladění našich uživatelsky definovaných tříd.

Poznámky	To, co chceme...	Takže napíšeme...	A Python zavolá...
[1]	inicializace instance	`x = MyClass()`	`x.__init__()`
[2]	„oficiální“ řetězcová reprezentace	`repr(x)`	`x.__repr__()`
[3]	„neformální“ řetězcová podoba	`str(x)`	`x.__str__()`
[4]	„neformální“ podoba v poli bajtů	`bytes(x)`	`x.__bytes__()`
[5]	hodnota jako naformátovaný řetězec	`format(x, format_spec)`	`x.__format__(format_spec)`

[1] Metoda `__init__()` se volá až poté, co byla instance vytvořena. Pokud chceme ovládat proces skutečného vytváření instance, musíme použít metodu `__new__()`.
[2] Metoda `__repr__()` by podle konvence měla vracet řetězec, který je platným pythonovským výrazem.
[3] Metoda `__str__()` se volá také v případě, kdy použijeme `print(x)`.
[4] *Novinka v Pythonu 3*, která souvisí se zavedením typu `bytes`.
[5] Podle konvence by měl být `format_spec` v souladu s minijazykem pro specifikaci formátu. Modul `decimal.py` z pythonovské standardní knihovny má svou vlastní metodu `__format__()`.

B.3. Třídy, které se chovají jako iterátory

V kapitole o iterátorech jsme si ukázali, jak můžeme vytvořit iterátor od základů s využitím metod `__iter__()` a `__next__()`.

Poznámky	To, co chceme...	Takže napíšeme...	A Python zavolá...
[1]	iterování přes posloupnost	`iter(seq)`	`seq.__iter__()`
[2]	získání další hodnoty iterátoru	`next(seq)`	`seq.__next__()`
[3]	vytvoření iterátoru procházejícího v opačném pořadí	`reversed(seq)`	`seq.__reversed__()`

[1] Metoda `__iter__()` se volá, kdykoliv vytváříme nový iterátor. Je to dobré místo pro nastavení počátečních hodnot iterátoru.
[2] Metoda `__next__()` se volá, kdykoliv se snažíme o získání nové hodnoty iterátoru.
[3] Metoda `__reversed__()` se běžně nepoužívá. Vezme existující posloupnost a vrací iterátor, který produkuje prvky posloupnosti v opačném pořadí, tj. od posledního k prvnímu.

Jak jsme si ukázali v kapitole o iterátorech, cyklus `for` se může chovat jako iterátor. V následujícím cyklu:

```
for x in seq:
    print(x)
```

Python 3 vytvoří iterátor voláním `seq.__iter__()` a potom bude získávat hodnoty `x` voláním jeho metody `__next__()`. Jakmile metoda `__next__()` vyvolá výjimku `StopIteration`, cyklus `for` spořádaně skončí.

B.4. Vypočítávané atributy

Poznámky	To, co chceme...	Takže napíšeme...	A Python zavolá...
1]	získat vypočítaný atribut (nepodmíněně)	`x.my_property`	`x.__getattribute__ ('my_property')`
[2]	získat vypočítaný atribut (fallback)	`x.my_property`	`x.__getattr__('my_property')`
[3]	nastavit hodnotu atributu	`x.my_property = value`	`x.__setattr__('my_property', value)`
[4]	zrušit atribut	`del x.my_property`	`x.__delattr__('my_property')`
[5]	vypsat seznam atributů a metod	`dir(x)`	`x.__dir__()`

[1] Pokud třída definuje metodu `__getattribute__()`, zavolá ji Python *při každém odkazu na libovolný atribut nebo jméno metody* (s výjimkou jmen speciálních metod, protože by tím vznikl nepříjemný nekonečný cyklus).
[2] Pokud třída definuje metodu `__getattr__()`, bude ji Python volat až poté, co atribut nenajde na některém z běžných míst. Pokud instance `x` definuje atribut color, *nepovede* použití `x.color` k volání `x.__getattr__('color')`. Jednoduše se vrátí již definovaná hodnota `x.color`.
[3] Metoda `__setattr__()` se volá, kdykoliv chceme atributu přiřadit nějakou hodnotu.
[4] Metoda `__delattr__()` se volá, kdykoliv chceme atribut zrušit.
[5] Metoda `__dir__()` je užitečná v případech, kdy definujeme metodu `__getattr__()` nebo metodu `__getattribute__()`. Normálně bychom voláním funkce `dir(x)` získali jen seznam běžných atributů a metod. Pokud například metoda `__getattr__()` vytváří atribut `color` dynamicky, nevypisoval by se `color` v seznamu vraceném funkcí `dir(x)` jako jeden z dostupných atributů. Předefinování metody `__dir__()` nám umožní vypsat `color` jako dostupný atribut. Může to být užitečné pro jiné programátory, kteří si přejí používat naši třídu, aniž by museli zkoumat její vnitřní možnosti.

Rozdíl mezi metodami `__getattr__()` a `__getattribute__()` je jemný, ale důležitý. Vysvětlíme si ho na dvou příkladech:

```
class Dynamo:
    def __getattr__(self, key):
        if key == 'color':                [1]
            return 'PapayaWhip'
        else:
            raise AttributeError          [2]

>>> dyn = Dynamo()
>>> dyn.color                             [3]
'PapayaWhip'
>>> dyn.color = 'LemonChiffon'
>>> dyn.color                             [4]
'LemonChiffon'
```

[1] Jméno atributu se předá metodě `__getattr__()` jako řetězec. Pokud je jméno rovno `'color'`, vrátí metoda hodnotu. (V tomto případě se jedná o pevně zadaný řetězec, ale normálně bychom zde provedli nějaký výpočet a vrátili bychom řetězec.)

[2] Pokud jméno atributu neznáme, musí metoda `__getattr__()` vyvolat výjimku `AttributeError`. V opačném případě by náš kód při přístupu k nedefinovanému atributu potichu selhal. (Pokud metoda nevyvolá výjimku nebo explicitně nevrátí nějakou hodnotu, pak — z technického hlediska — vrací `None`, což je pythonovská hodnota null. To znamená, že by *všechny* atributy, které by nebyly explicitně definovány, nabývaly hodnoty `None`. To téměř určitě nechceme.)

[3] Instance `dyn` nemá atribut jménem `color`, takže se zavolá metoda `__getattr__()`, která vrátí vypočítanou hodnotu.

[4] Jakmile explicitně nastavíme `dyn.color`, přestane se metoda `__getattr__()` pro získání hodnoty `dyn.color` volat, protože atribut `dyn.color` už je v instanci definován.

Ve srovnání s tím je metoda `__getattribute__()` absolutní a nepodmíněná.

```
class SuperDynamo:
    def __getattribute__(self, key):
        if key == 'color':
            return 'PapayaWhip'
        else:
            raise AttributeError
```

```
>>> dyn = SuperDynamo()
>>> dyn.color                              [1]
'PapayaWhip'
>>> dyn.color = 'LemonChiffon'
>>> dyn.color                              [2]
'PapayaWhip'
```

[1] Pro získání hodnoty dyn.color se volá metoda `__getattribute__()`.

[2] Dokonce i když explicitně nastavíme `dyn.color`, bude se pro získávání hodnoty `dyn.color` stále volat metoda `__getattribute__()`. Pokud je metoda `__getattribute__()` definována, *volá se nepodmíněně* při hledání každého atributu nebo metody. Platí to i pro atributy, které jsme po vytvoření instance explicitně nastavili (a tím vytvořili).

> Pokud vaše třída definuje metodu `__getattribute__()`, pak pravděpodobně chcete definovat také metodu `__setattr__()`. Pro udržení přehledu o hodnotách atributů musíte mezi těmito metodami zajistit spolupráci. V opačném případě by se atributy nastavené po vytvoření instance ztrácely v černé díře.

U metody `__getattribute__()` musíme být velmi pečliví, protože ji Python používá i při hledání jmen metod třídy.

```
class Rastan:
    def __getattribute__(self, key):
        raise AttributeError            [1]
    def swim(self):
        pass

>>> hero = Rastan()
>>> hero.swim()                         [2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __getattribute__
AttributeError
```

[1] Tato třída definuje metodu `__getattribute__()`, která vždy vyvolá výjimku `AttributeError`. Hledání každého atributu nebo metody skončí neúspěšně.

[2] Pokud zavoláme `hero.swim()`, začne Python v třídě `Rastan` hledat metodu `swim()`. Hledání prochází metodou `__getattribute__()`, *protože hledání všech atributů a metod prochází metodou* `__getattribute__()`. V tomto případě metoda `__getattribute__()` vyvolá výjimku `AttributeError`, takže hledání metody selže a tím pádem selže i její volání.

B.5. Třídy, které se chovají jako funkce

Pokud třída definuje metodu `__call__()`, můžeme instanci třídy volat (callable), jako kdyby to byla funkce.

To, co chceme...	Takže napíšeme...	A Python zavolá...
„volat“ instaci jako funkci	`my_instance()`	`my_instance.__call__()`

Modul `zipfile` tento způsob používá pro definici třídy, která umí zadaným heslem dešifrovat (decrypt) zašifrovaný (encrypted) zip soubor. Dešifrovací algoritmus pro zip vyžaduje, aby se během dešifrování ukládal stav. Pokud dešifrátor (decryptor) definujeme jako třídu, může si stav uchovávat uvnitř instance své třídy. Stav se inicializuje v metodě `__init__()` a aktualizuje se během dešifrování souboru. Ale protože je třída definována jako „volatelná“ (jako funkce), můžeme instanci třídy předat jako první argument funkce `map()` takto:

```
# výňatek ze zipfile.py
class _ZipDecrypter:
.
.
.
    def __init__(self, pwd):
        self.key0 = 305419896                    [1]
        self.key1 = 591751049
        self.key2 = 878082192
        for p in pwd:
            self._UpdateKeys(p)

    def __call__(self, c):                       [2]
        assert isinstance(c, int)
        k = self.key2 | 2
        c = c ^ (((k * (k^1)) >> 8) & 255)
        self._UpdateKeys(c)
        return c
.
.
.
zd = _ZipDecrypter(pwd)                          [3]
bytes = zef_file.read(12)
h = list(map(zd, bytes[0:12]))                   [4]
```

[1] Třída `_ZipDecryptor` udržuje stav v podobě tří rotujících klíčů, které se později aktualizují metodou `_UpdateKeys()` (zde neukázána).

[2] Třída definuje metodu `__call__()`, která způsobuje, že instance třídy můžeme volat, jako kdyby to byly funkce. V tomto případě metoda `__call__()` dešifruje jeden bajt ze zip souboru a potom aktualizuje rotující klíče podle hodnoty dešifrovaného bajtu.

[3] zd je instancí třídy `_ZipDecryptor`. Proměnná `pwd` (password; heslo) je předána metodě `__init__()`, která její obsah uloží a použije jej pro první aktualizaci rotujících klíčů.

[4] Máme prvních 12 bajtů zip souboru. Dešifrujeme je zobrazením bajtů přes `zd`. To znamená, že se 12krát „volá“ `zd`, což znamená, že se 12krát volá metoda `__call__()`, která aktualizuje vnitřní stav instance a 12krát vrací výsledný bajt.

B.6. Třídy, které se chovají jako množiny

Pokud se naše třída chová jako kontejner pro množinu hodnot — tj. pokud má smysl ptát se, zda naše třída „obsahuje“ hodnotu — , pak by pravděpodobně měla definovat následující speciální metody, které způsobí, že se bude chovat jako množina.

To, co chceme...	Takže napíšeme...	A Python zavolá...
počet položek	`len(s)`	`s.__len__()`
test, zda posloupnost obsahuje určitou hodnotu	`x in s`	`s.__contains__(x)`

Modul `cgi` tyto metody používá ve své třídě `FieldStorage`, která reprezentuje všechna pole formuláře nebo parametry dotazu, které byly zaslány na dynamickou webovou stránku.

```
# Skript, který reaguje na http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:                                                    [1]
  do_search()

# Výňatek z cgi.py, který vysvětluje, jak to funguje
class FieldStorage:
.
.
.
    def __contains__(self, key):                                 [2]
        if self.list is None:
            raise TypeError('not indexable')
        return any(item.name == key for item in self.list)       [3]

    def __len__(self):                                           [4]
        return len(self.keys())                                  [5]
```

[1] Jakmile vytvoříme instanci třídy `cgi.FieldStorage`, můžeme použít operátor „in“ pro ověření, zda se v řetězci s dotazem nachází určitý parametr.

[2] Kouzlem, které to umožní, je metoda `__contains__()`. Pokud napíšeme `if 'q' in fs`, hledá Python metodu `__contains__()` objektu `fs`, který je definován v `cgi.py`. Hodnota `'q'` je předána metodě `__contains__()` jako argument `key`.

[3] Funkce `any()` přebírá generátorový výraz a vrací `True`, pokud generátor vrátí alespoň jeden prvek. Navíc je dost chytrá, aby skončila s testováním hned, jak se podaří podmínku splnit.

[4] Stejná třída `FieldStorage` podporuje také vracení své délky, takže můžeme napsat `len(fs)` a zavolá se metoda `__len__()` třídy `FieldStorage`, která vrátí počet rozpoznaných parametrů dotazu.

[5] Metoda `self.keys()` kontroluje, zda `self.list is None` (zda seznam vůbec existuje), takže metoda `__len__` nemusí uvedenou kontrolu chyb dublovat.

B.7. Třídy, které se chovají jako slovníky

Když předchozí možnosti trošku rozšíříme, můžeme definovat třídy, které nejenže reagují na operátor „`in`“ a na funkci `len()`, ale které se mohou chovat jako plnohodnotné slovníky vracející hodnoty vázané na klíče.

To, co chceme…	Takže napíšeme…	A Python zavolá…
získat hodnotu podle klíče	`x[key]`	`x.__getitem__(key)`
nastavit hodnotu vázanou na klíč	`x[key] = value`	`x.__setitem__(key, value)`
zrušit dvojici klíč-hodnota	`del x[key]`	`x.__delitem__(key)`
vrátit výchozí hodnotu pro chybějící klíče	`x[nonexistent_key]`	`x.__missing__(nonexistent_key)`

Třída `FieldStorage` z modulu `cgi` definuje rovněž tyto speciální metody, což znamená, že můžeme dělat například následující věci:

```
# Skript, který reaguje na http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:
  do_search(fs['q'])                                          [1]

# Výňatek z cgi.py, který ukazuje, jak to funguje
class FieldStorage:
.
.
.
    def __getitem__(self, key):                               [2]
        if self.list is None:
            raise TypeError('not indexable')
```

```
        found = []
        for item in self.list:
            if item.name == key: found.append(item)
        if not found:
            raise KeyError(key)
        if len(found) == 1:
            return found[0]
        else:
            return found
```

[1] Objekt `fs` je instancí `cgi.FieldStorage`, ale přesto můžeme používat výrazy jako `fs['q']`.

[2] `fs['q']` zavolá metodu `__getitem__()` s parametrem key nastaveným na `'q'`. Potom se ve vnitřním seznamu parametrů dotazu (`self.list`) hledá položka, jejíž atribut `.name` je roven zadanému klíči.

B.8. Třídy, které se chovají jako čísla

Při použití příslušných speciálních metod můžeme definovat své vlastní třídy, které se chovají jako čísla. To znamená, že je můžeme sčítat, odčítat a provádět s nimi další matematické operace. Tímto způsobem jsou implementovány věci v modulu `fractions` — třída Fraction implementuje speciální metody, které nám umožňují provádět takovéto věci:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> x / 3
Fraction(1, 9)
```

Zde je úplný seznam speciálních metod, které musí implementovat třída chovající se jako číslo.

To, co chceme...	Takže napíšeme...	A Python zavolá...
sčítání	`x + y`	`x.__add__(y)`
odčítání	`x - y`	`x.__sub__(y)`
násobení	`x * y`	`x.__mul__(y)`
dělení	`x / y`	`x.__truediv__(y)`
celočíselné dělení (floor division)	`x // y`	`x.__floordiv__(y)`
modulo (zbytek)	`x % y`	`x.__mod__(y)`
celočíselné dělení a zbytek	`divmod(x, y)`	`x.__divmod__(y)`
umocnění na	`x ** y`	`x.__pow__(y)`
bitový posun doleva	`x << y`	`x.__lshift__(y)`
bitový posun doprava	`x >> y`	`x.__rshift__(y)`
logický součin po bitech (and)	`x & y`	`x.__and__(y)`
xor po bitech	`x ^ y`	`x.__xor__(y)`
logický součet po bitech (or)	`x | y`	`x.__or__(y)`

Pokud je x instancí třídy, která tyto metody implementuje, bude to fungovat bez problémů. Ale co když třída některou z těchto metod neimplementuje? Nebo ještě hůř — co když je implementuje, ale neporadí si s některými druhy argumentů? Například:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> 1 / x
Fraction(3, 1)
```

Tohle *není* případ, kdy se vezme `Fraction` a dělí se celým číslem (jako v předchozím příkladu). Minulý příklad byl přímočarý: `x / 3` volá `x.__truediv__(3)` a metoda `__truediv__()` třídy `Fraction` provede matematickou operaci. Ale objekty typu celé číslo (`int`) „neumí“ dělat aritmetické operace se zlomky. Takže jak je možné, že ten příklad funguje?

Existuje druhá sada aritmetických speciálních metod *s obrácenými operandy* (reflected operands). Pokud matematická operace vyžaduje dva operandy (například `x / y`), dá se to řešit dvěma způsoby:

1. Řekneme `x`, aby podělilo samo sebe hodnotou `y`, nebo
2. řekneme `y`, aby se zachovalo jako dělitel hodnoty `x`.

Výše uvedená sada speciálních metod používá první přístup: pokud máme `x / y`, poskytují metody způsob, jak může `x` říci: „Já vím, jak vydělit sebe hodnotou `y`.“ Následující sada speciálních metod se pouští do druhého přístupu — metody poskytují způsob, jakým může y vyjádřit: „Já vím, jak být dělitelem a podělit sebou hodnotu `x`.“

To, co chceme…	Takže napíšeme…	A Python zavolá…
sčítání	`x + y`	`y.__radd__(x)`
odčítání	`x - y`	`y.__rsub__(x)`
násobení	`x * y`	`y.__rmul__(x)`
dělení	`x / y`	`y.__rtruediv__(x)`
celočíselné dělení (floor division)	`x // y`	`y.__rfloordiv__(x)`
modulo (zbytek)	`x % y`	`y.__rmod__(x)`
celočíselné dělení a zbytek	`divmod(x, y)`	`y.__rdivmod__(x)`
umocnění na	`x ** y`	`y.__rpow__(x)`
bitový posun doleva	`x << y`	`y.__rlshift__(x)`
bitový posun doprava	`x >> y`	`y.__rrshift__(x)`
logický součin po bitech (and)	`x & y`	`y.__rand__(x)`
xor po bitech	`x ^ y`	`y.__rxor__(x)`
logický součet po bitech (or)	`x | y`	`y.__ror__(x)`

Ale moment! Ono je toho ještě víc! Pokud provádíme operace „přímo nad proměnnou“ (in-place, in situ, na místě samém), jako například `x/=3`, můžeme definovat ještě další speciální metody.

To, co chceme...	Takže napíšeme...	A Python zavolá...
sčítání nad proměnnou	`x += y`	`x.__iadd__(y)`
odčítání nad proměnnou	`x -= y`	`x.__isub__(y)`
násobení nad proměnnou	`x *= y`	`x.__imul__(y)`
dělení nad proměnnou	`x /= y`	`x.__itruediv__(y)`
celočíselné dělení nad proměnnou (floor division)	`x //= y`	`x.__ifloordiv__(y)`
modulo nad proměnnou	`x %= y`	`x.__imod__(y)`
umocnění nad proměnnou	`x **= y`	`x.__ipow__(y)`
bitový posun doleva nad proměnnou	`x <<= y`	`x.__ilshift__(y)`
bitový posun doprava nad proměnnou	`x >>= y`	`x.__irshift__(y)`
logický součin po bitech	`x &= y`	`x.__iand__(y)`
nad proměnnou (and)		
xor po bitech nad proměnnou	`x ^= y`	`x.__ixor__(y)`
logický součet po bitech nad proměnnou (or)	`x |= y`	`x.__ior__(y)`

Poznámka: Ve většině případů se implementace „in situ“ metod nevyžaduje. Pokud pro určitou operaci příslušnou „in situ“ metodu (tj. nad proměnnou) nedefinujeme, Python se ji pokusí nahradit. Například při provádění výrazu `x /= y` Python...

1. Vyzkouší zavolat `x.__itruediv__(y)`. Pokud je metoda definována a vrátila hodnotu jinou než `NotImplemented`, je to hotové.
2. Vyzkouší zavolat `x.__truediv__(y)`. Pokud je metoda definována a vrátila hodnotu jinou než `NotImplemented`, je původní hodnota `x` zahozena a je nahrazena výslednou hodnotou — jako kdybychom místo toho napsali `x = x / y`.
3. Vyzkouší zavolat `y.__rtruediv__(x)`. Pokud je metoda definována a vrátila hodnotu jinou než `NotImplemented`, je původní hodnota `x` zahozena a je nahrazena výslednou hodnotou.

Takže „in situ“ metodu jako `__itruediv__()` definujeme jen v případech, kdy chceme pro in situ operandy provádět nějakou speciální optimalizaci. V opačném případě Python v podstatě přeformuluje požadavek provedení operandu nad proměnnou na běžnou podobu operandu s přiřazením výsledku do proměnné.

Objekty, které se chovají jako číslo, mohou nad sebou provádět také pár „unárních“ matematických operací.

Poznámky	To, co chceme...	Takže napíšeme...	A Python zavolá...
	unární minus (záporné číslo)	`-x`	`x.__neg__()`
	unární plus (kladné číslo)	`+x`	`x.__pos__()`
	absolutní hodnota	`abs(x)`	`x.__abs__()`
	inverze	`~x`	`x.__invert__()`
	převod na komplexní číslo	`complex(x)`	`x.__complex__()`
	převod na celé číslo	`int(x)`	`x.__int__()`
	převod na reálné číslo	`float(x)`	`x.__float__()`
	převod na nejbližší celé číslo zaokrouhlením	`round(x)`	`x.__round__()`
	převod na nejbližší číslo zaokrouhlením na n desetinných míst	`round(x, n)`	`x.__round__(n)`
	nejmenší celé číslo `>= x`	math.ceil(x)	x.__ceil__()
	největší celé číslo `<= x`	`math.floor(x)`	`x.__floor__()`
	odseknutí `x` na nejbližší celé číslo směrem k `0`	`math.trunc(x)`	`x.__trunc__()`
PEP 357	číslo jako index seznamu	`a_list[x]`	`a_list[x.__index__()]`

B.9. Třídy, které se dají porovnávat

Tuto část jsem od předchozí oddělil, protože porovnání se neomezuje jen na čísla. Porovnávat se dají hodnoty mnoha datových typů — řetězce, seznamy a dokonce i slovníky. Pokud vytváříme svou vlastní třídu a má smysl uvažovat o porovnávání našeho objektu s jinými objekty, můžeme porovnání implementovat následujícími speciálními metodami.

To, co chceme...	Takže napíšeme...	A Python zavolá...
rovnost	`x == y`	`x.__eq__(y)`
různost (nerovnost)	`x != y`	`x.__ne__(y)`
menší než	`x < y`	`x.__lt__(y)`
menší než nebo rovno	`x <= y`	`x.__le__(y)`
větší než	`x > y`	`x.__gt__(y)`
větší než nebo rovno	`x >= y`	`x.__ge__(y)`
pravdivostní hodnota v booleovském kontextu	`if x:`	`x.__bool__()`

> Pokud definujeme metodu `__lt__()`, ale nedefinujeme metodu `__gt__()`, použije Python metodu `__lt__()` s přehozenými operandy. Ale Python neprovádí kombinaci metod. Pokud například definujeme metodu `__lt__()` a metodu `__eq__()` a pokusíme se otestovat, zda je `x <= y`, Python nezavolá postupně `__lt__()` a `__eq__()`. Zavolá pouze metodu `__le__()`.

B.10. Třídy, které podporují serializaci
B.11. Třídy, které mohou být použity v bloku `with`

B.10. Třídy, které podporují serializaci

Python podporuje serializaci a deserializaci libovolných objektů. (Většina pythonovských příruček tento proces nazývá „pickling“ a „unpickling“.) Může to být užitečné pro uložení stavu objektu do souboru a jeho pozdější obnovení. Všechny přirozené datové typy již „piklení“ podporují. Pokud vytvoříte uživatelskou třídu a chcete ji umět serializovat, přečtěte si něco o pickle protokolu, abyste věděli, kdy a jak se volají následující speciální metody.

Poznámky	To, co chceme...	Takže napíšeme...	A Python zavolá...
	uživatelská kopie objektu	`copy.copy(x)`	`x.__copy__()`
	uživatelská kopie objektu do hloubky (deep copy)	`copy.deepcopy(x)`	`x.__deepcopy__()`
*	zjištění stavu objektu před serializací	`pickle.dump(x, file)`	`x.__getstate__()`
*	serializace objektu	`pickle.dump(x, file)`	`x.__reduce__()`
*	serializace objektu (nový serializační protokol)	`pickle.dump(x, file, protocol_version)`	`x.__reduce_ex__(protocol_version)`
*	kontrola nad vytvářením objektu	`x = pickle.load(file)`	`x.__getnewargs__()`
*	během deserializace (unpickling) obnovení stavu objektu po deserializaci	`x = pickle.load(file)`	`x.__setstate__()`

* Při znovuvytváření serializovaného objektu musí Python nejdříve vytvořit nový objekt, který vypadá jako ten serializovaný, a potom musí nastavit hodnoty všech jeho atributů. Metoda `__getnewargs__()` řídí způsob vytváření objektu. Metoda `__setstate__()` poté řídí obnovení hodnot atributů.

B.11. Třídy, které mohou být použity v bloku `with`

Blok `with` definuje operační kontext (runtime context). „Vstupujeme“ do něj (enter) v okamžiku provádění příkazu `with` a „vystupujeme“ z něj (exit) po provedení posledního příkazu v jeho bloku.

To, co chceme...	Takže napíšeme...	A Python zavolá...
udělej něco speciálního při vstupu do bloku with	`with x:`	`x.__enter__()`
udělej něco speciálního při opouštění bloku with	`with x:`	`x.__exit__(exc_type, exc_value, traceback)`

Obrat with soubor funguje následovně:

```
# výňatek z io.py
def _checkClosed(self, msg=None):
    '''Internal: raise an ValueError if file is closed
    '''
    if self.closed:
        raise ValueError('I/O operation on closed file.'
                         if msg is None else msg)

def __enter__(self):
    '''Context management protocol.  Returns self.'''
    self._checkClosed()                                    [1]
    return self                                            [2]

def __exit__(self, *args):
    '''Context management protocol.  Calls close()'''
    self.close()                                           [3]
```

[1] Objekt souboru definuje jak metodu `__enter__()`, tak metodu `__exit__()`. Metoda `__enter__()` kontroluje, zda je soubor otevřen. Pokud ne, vyvolá metoda `_checkClosed()` výjimku.

[2] Metoda `__enter__()` by měla téměř vždy vrátit `self`, což je objekt, který bude v bloku `with` použit pro práci s vlastnostmi (properties) a k volání metod.

[3] Po ukončení bloku `with` se souborový objekt automaticky uzavře. Jak se to udělá? V metodě `__exit__()` se zavolá `self.close()`.

> Metoda __exit__() se zavolá vždy, dokonce i když je uvnitř bloku `with` vyvolána výjimka. Ve skutečnosti je to tak, že při vyvolání výjimky je informace o výjimce předána metodě `__exit__()`. Další detaily naleznete ve standardní dokumentaci: **With Statement Context Managers** (správci kontextu příkazu with).

Kap. O správcích kontextu se dozvíte víc v části Automatické zavírání souborů a Přesměrování standardního výstupu.

B.12. Opravdu esoterické věci

Pokud víme, co děláme, můžeme získat téměř úplnou kontrolu nad tím, jak jsou třídy porovnávány, jak jsou definovány atributy a jaký druh tříd se považuje za podtřídy naší třídy.

Poznámky	To, co chceme...	Takže napíšeme...	A Python zavolá...
	konstruktor třídy	`x = MyClass()`	`x.__new__()`
*	destruktor třídy	`del x`	`x.__del__()`
	definovat jen určité atributy		`x.__slots__()`
	uživatelská heš-hodnota	`hash(x)`	`x.__hash__()`
	získat hodnotu vlastnosti (property)	`x.color`	`type(x).__dict__['color'].__get__(x, type(x))`
	nastavit hodnotu vlastnosti	`x.color = 'PapayaWhip'`	`type(x).__dict__['color'].__set__(x, 'PapayaWhip')`
	zrušit vlastnost	`del x.color`	`type(x).__dict__['color'].__del__(x)`
	zkontrolovat, zda je nějaký objekt instancí naší třídy	`isinstance (x, MyClass)`	`MyClass.__instancecheck__(x)`
	zkontrolovat, zda je nějaká třída podtřídou naší třídy	`issubclass (C, MyClass)`	`MyClass.__subclasscheck__(C)`
	zkontrolovat, zda je nějaká třída podtřídou naší abstraktní bázové třídy	`issubclass(C, MyABC)`	`MyABC.__subclasshook__(C)`

* Okolnosti toho, kdy přesně Python volá speciální metodu `__del__()`, jsou neuvěřitelně komplikované. Abyste tomu porozuměli úplně, musíte vědět, jakým způsobem Python sleduje objekty v paměti. Tady najdete dobrý článek **o mechanismu automatického uvolňování paměti** (garbage collection) a **o destruktorech tříd v jazyce Python** (anglicky). Měli byste si také přečíst o slabých referencích (weak references), o modulu `weakref` a navrch pravděpodobně také o modulu `gc`.

B.13. Přečtěte si

Moduly zmíněné v této příloze (standardní dokumentace):

- **Modul `zipfile`**
  (http://docs.python.org/py3k/library/zipfile.html)
- **Modul `cgi`**
  (http://docs.python.org/py3k/library/cgi.html)
- **Modul `collections`**
  (http://docs.python.org/py3k/library/collections.html)

- **Modul `math`**
  (http://docs.python.org/py3k/library/math.html)
- **Modul `pickle`**
  (http://docs.python.org/py3k/library/pickle.html)
- **Modul `copy`**
  (http://docs.python.org/py3k/library/copy.html)
- **Modul `abc`** („Abstract Base Classes“; abstraktní bázové třídy)
  (http://docs.python.org/py3k/library/abc.html)

Další lehké čtení (standardní dokumentace):

- **Format Specification Mini-Language** (minijazyk pro specifikaci formátu)
  (http://docs.python.org/release/3.1/library/string.html)
- **Python data model** (pythonovský datový model)
  (http://docs.python.org/release/3.1/reference/datamodel.html)
- **Built-in types** (zabudované typy)
  (http://docs.python.org/release/3.1/library/stdtypes.html)
- **`PEP` 357: Allowing Any Object to be Used for Slicing** (jak umožnit každému objektu být použit pro řezy)
  (www.python.org/dev/peps/pep-0357/)
- **`PEP` 3119: Introducing Abstract Base Classes** (úvod do abstraktních bázových tříd)
  (www.python.org/dev/peps/pep-3119/)

“ Go forth on your path, as it exists only through your walking.”
(Jdi dál svou cestou, protože ta existuje jen pod tvými kroky.)

— Sv. Augustin z Hippo (připisováno)

C. Čím pokračovat

C. Čím pokračovat — 423
C.1. Doporučuji k přečtení — **425**
C.2. Kde hledat kód kompatibilní s Pythonem 3 — **426**

C.1. Doporučuji k přečtení

V této knize se bohužel nemůžu zabývat všemi stránkami jazyka Python 3. Naštěstí můžete všude najít mnoho nádherných, volně dostupných učebnic.

Dekorátory:

- **Function Decorators —** Ariel Ortiz (dekorátory funkcí) (http://programmingbits.pythonblogs.com/27_programmingbits/archive/50_function_decorators.html)
- **More on Function Decorators —** Ariel Ortiz (více o dekorátorech funkcí) (http://programmingbits.pythonblogs.com/27_programmingbits/archive/51_more_on_function_decorators.html)
- **Charming Python: Decorators make magic easy —** David Mertz (dekorátory s magickou lehkostí) (www.ibm.com/developerworks/linux/library/l-cpdecor.html)
- **Function Definitions** (definice funkcí) v oficiální pythonovské dokumentaci (http://docs.python.org/reference/compound_stmts.html#function)

Vlastnosti (properties):

- **The Python `property` builtin —** Adam Gomaa (http://adam.gomaa.us/blog/2008/aug/11/the-python-property-builtin/)
- **Getters/Setters/Fuxors —** Ryan Tomayko (http://tomayko.com/writings/getters-setters-fuxors)
- **`property()` function** v oficiální pythonovské dokumentaci (http://docs.python.org/library/functions.html#property)

Deskriptory:

- **How-To Guide For Descriptors —** Raymond Hettinger (http://users.rcn.com/python/download/Descriptor.htm)
- **Charming Python: Python elegance and warts, Part 2 —** David Mertz (www.ibm.com/developerworks/linux/library/l-python-elegance-2.html)
- **Python Descriptors —** Mark Summerfield (www.informit.com/articles/printerfriendly.aspx?p=1309289)
- **Invoking Descriptors** v oficiální pythonovské dokumentaci (http://docs.python.org/py3k/reference/datamodel.html)

Vlákna a multiprocesing:

- **Modul `threading`** (http://docs.python.org/py3k/library/threading.html)
- **`threading` — Manage concurrent threads** (www.doughellmann.com/PyMOTW/threading/)
- **Modul `multiprocessing`** (http://docs.python.org/py3k/library/multiprocessing.html)

- **`multiprocessing` — Manage processes like threads** (http://www.doughellmann.com/PyMOTW/multiprocessing/)
- **Python threads and the Global Interpreter Lock** — Jesse Noller (http://jessenoller.com/2009/02/01/python-threads-and-the-global-interpreter-lock/)
- **Inside the Python `GIL`** (video) — David Beazley (http://blip.tv/file/2232410)

Metatřídy:

- **Metaclass programming in Python** — David Mertz a Michele Simionato (www.ibm.com/developerworks/linux/library/l-pymeta.html)
- **Metaclass programming in Python, Part 2** — David Mertz a Michele Simionato (www.ibm.com/developerworks/linux/library/l-pymeta2/)
- **Metaclass programming in Python, Part 3** — David Mertz a Michele Simionato (www.ibm.com/developerworks/linux/library/l-pymeta3.html)

A navíc fantastický průvodce mnoha moduly ze standardní pythonovské knihovny od Douga Hellmana — **Python Module of the Week** (http://www.doughellmann.com/PyMOTW/contents.html).

C.2. Kde hledat kód kompatibilní s Pythonem 3

Python 3 je relativně nový, takže o kompatibilní knihovny je nouze. Tady jsou nějaká místa, kde můžete hledat kód, který funguje s Pythonem 3 (vše anglicky).

- **Python Package Index:** seznam balíčků pro Python 3 (http://pypi.python.org/pypi?:action=browse&c=533&show=all)
- **Python Cookbook:** seznam receptů označkovaných „python3“ (http://code.activestate.com/recipes/langs/python/tags/python3/)
- **Google Project Hosting:** seznam projektů označkovaných „python3“ (http://code.google.com/hosting/search?q=label:python3)
- **SourceForge:** seznam projektů vyhledaných podle „Python 3“ (http://sourceforge.net/search/?words=%22python+3%22)
- **GitHub:** seznam projektů vyhledaných podle „python3“ (a také podle „python 3“) (https://github.com/search?type=Repositories&language=python&q=python3) (https://github.com/search?type=Repositories&language=python&q=python+3)
- **BitBucket:** seznam projektů vyhledaných podle „python3“ (a podle „python 3“) (http://bitbucket.org/repo/all/?name=python3) (http://bitbucket.org/repo/all/?name=python+3)

“Where’s the ANY key?”
(Kde je LIBOVOLNÁ klávesa?)

— připisováno kdekomu

D. Odstraňování problémů

D. Odstraňování problémů — 427
D.1. Ponořme se — **429**
D.2. Jak se dostat k příkazovému řádku — **429**
D.3. Spuštění Pythonu z příkazového řádku — **429**

D.1. Ponořme se
D.2. Jak se dostat k příkazovému řádku
D.3. Spuštění Pythonu z příkazového řádku

D.1. Ponořme se

D O P S A T

D.2. Jak se dostat k příkazovému řádku

V celé knize se setkáváme s příklady spouštění Pythonu z příkazového řádku. Ale jak se máte k příkazovému řádku dostat?

V Linuxu se podívejte do menu `Applications` a hledejte program zvaný `Terminal`. (Může se nacházet v podmenu jako `Accessories` nebo `System`.)

V Mac OS X naleznete v adresáři `/Application/Utilities/` aplikaci nazvanou `Terminal.app`. Dostanete se tam tak, že kliknete na pracovní plochu, otevřete menu `Go`, vyberete `Go to folder...` (přejít do adresáře) a napíšete `/Applications/Utilities/`. Nakonec poklepete na program `Terminal`.

Ve Windows kliknete na `Start`, vyberete položku `Spustit...`, napíšete `cmd` a stisknete `ENTER`.

D.3. Spuštění Pythonu z příkazového řádku

Jakmile se dostanete na příkazový řádek, měli byste být schopni spustit pythonovský interaktivní shell. V Linuxu nebo v Mac OS X napište na příkazový řádek `python3` a stiskněte `ENTER`. Ve Windows napište na příkazový řádek `c:\python31\python` a stiskněte `ENTER`. Pokud půjde vše dobře, měli byste vidět něco takového:

```
you@localhost:~$ python3
Python 3.1 (r31:73572, Jul 28 2009, 06:52:23)
[GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(Až budete chtít pythonovský interaktivní shell opustit a vrátit se na příkazový řádek, napište `exit()` a stiskněte `ENTER`. Tento obrat funguje na všech platformách.)

Pokud zpozorujete chybu „příkaz nenalezen“ (command not found), znamená to, že pravděpodobně nemáte Python 3 nainstalován.

```
you@localhost:~$ python3
bash: python3: command not found
```

Pokud se do pythonovského interaktivního shellu dostanete, ale číslo verze neodpovídá vašemu očekávání, máte možná nainstalovánu více než jednu verzi Pythonu. Stává se to spíš na systémech Linux a Mac OS X, kde může být starší verze Pythonu předinstalována. Poslední verzi můžete nainstalovat, aniž byste museli starší verzi mazat (mohou být bez problémů instalovány vedle sebe), ale při spouštění Pythonu z příkazového řádku se pak musíte vyjádřit přesněji.

Například na svém domácím linuxovém stroji mám nainstalováno několik verzí Pythonu, abych na nich mohl otestovat software, který vytvářím. Když chci spustit určitou verzi, můžu napsat `python3.0`, `python3.1` nebo `python2.6`.

```
mark@atlantis:~$ python3.0
Python 3.0.1+ (r301:69556, Apr 15 2009, 17:25:52)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
mark@atlantis:~$ python3.1
Python 3.1 (r31:73572, Jul 28 2009, 06:52:23)
[GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
mark@atlantis:~$ python2.6
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Mark Pilgrim
PONOŘME SE DO PYTHON(U) 3

Přeloženo z anglického originálu knihy *Dive Into Python 3*.
1. vydání, 2009
Vydáno nakladatelstvím Apress Media, LLC (*apress.com*), NY.

Vydavatel:
CZ.NIC, z. s. p. o.
Americká 23, 120 00 Praha 2
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2011
Kniha vyšla jako 3. publikace v Edici CZ.NIC.

© 2010 Mark Pilgrim
Uvedené dílo podléhá licenci Creative Commons Uveďte autora-Zachovejte licenci 3.0 Unported.

ISBN 978-80-904248-2-1 (tištěná verze, PDF)
ISBN 978-80-88168-01-0 (ve formátu EPUB)
ISBN 978-80-88168-02-7 (ve formátu MOBI)

CZ.NIC | SPRÁVCE DOMÉNY CZ

Edice CZ.NIC

O autorovi **Mark Pilgrim** se nesmazatelně zapsal do povědomí pythonovské komunity už svojí knihou „Dive Into Python“, ve které originálním a nezapomenutelným způsobem přiblížil čtenářům osobitý styl programování v tomto jazyce, aby se o několik let později připomenul ještě výrazněji s knihou „Dive Into Python 3“, která je stejně originálním a zábavným způsobem věnována jeho nejnovější verzi. S podobným nadšením se však zabývá i dalšími tématy, jeho nejnovější kniha „HTML5: Up & Running“ je čtivým úvodem do problematiky posledního hitu na poli předávání informací na Internetu – standardu HTML5.

O edici Edice CZ.NIC je jedním z osvětových projektů správce české domény nejvyšší úrovně. Cílem tohoto projektu je vydávat odborné, ale i populární publikace spojené s internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz

ISBN 978-80-904248-2-1