Mark Summerfield

Python 3

Výukový kurz

Procedurální i objektově orientované programování

Ladění, testování, distribuce zátěže do více vláken

Síťová komunikace, aplikace typu klient-server, programování databází

Využití regulárních výrazů, tvorba grafického uživatelského rozhraní

Ke stažení zdrojové kódy příkladů z knihy

Mark Summerfield

Python 3
Výukový kurz

Computer Press
Brno
2013

Python 3
Výukový kurz

Mark Summerfield

Překlad: Lukáš Krejčí
Obálka: Martin Sodomka
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Authorized translation from the English language edition, entitled PROGRAMMING IN PYTHON 3: A COMPLETE INTRODUCTION TO THE PYTHON 3.1 LANGUAGE, 2nd Edition, 0321680561 by SUMMERFIELD, MARK, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2010 Pearson Education, Inc.
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. CZECH language edition published by COMPUTER PRESS, A.S., Copyright © 2010.
Autorizovaný překlad z originálního anglického vydání PROGRAMMING IN PYTHON 3: A COMPLETE INTRODUCTION TO THE PYTHON 3.1 LANGUAGE. Originální copyright: published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2010.
Překlad: © Computer Press, a.s., 2010.

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-2737-7

Vydalo nakladatelství Computer Press v Brně roku 2013 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 17 938.
© Albatros Media a. s. Všechna práva vyhrazena. Žádná část této publikace nesmí být kopírována a rozmnožována za účelem rozšiřování v jakékoli formě či jakýmkoli způsobem bez písemného souhlasu vydavatele.

Dotisk 1. vydání

Stručný obsah

1. Rychlý úvod do procedurálního programování 19
2. Datové typy 57
3. Datové typy představující kolekce 109
4. Řídicí struktury a funkce 159
5. Moduly 193
6. Objektově orientované programování 229
7. Práce se soubory 279
8. Pokročilé techniky programování 329
9. Ladění, testování a profilování 399
10. Procesy a vlákna 423
11. Propojení v síti 439
12. Programování databází 455
13. Regulární výrazy 469
14. Úvod do syntaktické analýzy 491
15. Seznámení s programováním grafického uživatelského rozhraní 543

Obsah

Úvod ... **13**
Uspořádání knihy ... 15
Získání a instalace Pythonu 3 ... 16
Poděkování ... 17

Lekce 1
Rychlý úvod do procedurálního programování ... **19**
Tvorba a spouštění programů napsaných v jazyku Python ... 20
Nádherné srdce jazyka Python ... 24
Oblast č. 1: Datové typy ... 25
Oblast č. 2: Odkazy na objekty ... 26
Oblast č. 3: Datové typy pro kolekce ... 28
Oblast č. 4: Logické operátory ... 31
Oblast č. 5: Příkazy pro řízení toku programu ... 35
Oblast č. 6: Aritmetické operátory ... 39
Oblast č. 7: Vstup a výstup ... 42
Oblast č. 8: Tvorba a volání funkcí ... 44
Příklady ... 46
Program bigdigits.py ... 46
Program generate_grid.py ... 49
Shrnutí ... 51
Cvičení ... 53

Lekce 2
Datové typy ... **57**
Identifikátory a klíčová slova ... 58
Celočíselné typy ... 60
Celá čísla ... 61
Logické hodnoty ... 64
Typy s pohyblivou řádovou čárkou ... 64
Čísla s pohyblivou řádovou čárkou ... 65
Komplexní čísla ... 68
Desetinná čísla ... 69

Řetězce ... 71
Porovnávání řetězců ... 73
Řezání a krokování řetězců ... 74
Řetězcové operátory a metody ... 76
Formátování řetězců metodou str.format() ... 83
Kódování znaků ... 95
Příklady ... 98
Program quadratic.py ... 98
Program csv2html.py ... 100
Shrnutí ... 105
Cvičení ... 107

Lekce 3
Datové typy představující kolekce ... 109
Typy představující posloupnost ... 110
N-tice ... 110
Pojmenované n-tice ... 113
Seznamy ... 115
Množinové typy ... 122
Množiny ... 123
Zmrazené množiny ... 126
Typy představující mapování ... 127
Slovníky ... 128
Výchozí slovníky ... 135
Uspořádané slovníky ... 136
Procházení a kopírování kolekcí ... 138
Operace a funkce pro iterátory a iterovatelné objekty ... 138
Kopírování kolekcí ... 146
Příklady ... 148
Program generate_usernames.py ... 148
Program statistics.py ... 151
Shrnutí ... 155
Cvičení ... 156

Lekce 4
Řídicí struktury a funkce ... 159
Řídicí struktury ... 160
Podmíněné větvení ... 160
Cykly ... 161

Zpracování výjimek ... 163
Zachytávání a vyvolávání výjimek ... 163
Vlastní výjimky ... 167
Vlastní funkce ... 171
Jména a dokumentační řetězce ... 175
Rozbalení argumentů a parametrů ... 176
Přístup k proměnným v globálním oboru platnosti ... 178
Lambda funkce ... 180
Tvrzení ... 181
Příklad: make_html_skeleton.py ... 183
Shrnutí ... 188
Cvičení ... 189

Lekce 5
Moduly ... 193
Moduly a balíčky ... 194
Balíčky ... 197
Vlastní moduly ... 200
Přehled standardní knihovny Pythonu ... 209
Práce s řetězci ... 210
Programování na příkazovém řádku ... 211
Matematika a čísla ... 212
Datum a čas ... 212
Algoritmy a datové kolekce představující kolekce ... 214
Souborové formáty, kódování a perzistence dat ... 215
Práce se soubory, adresáři a procesy ... 218
Sítě a Internet ... 220
XML ... 222
Další moduly ... 224
Shrnutí ... 225
Cvičení ... 226

Lekce 6
Objektově orientované programování ... 229
Objektově orientovaný přístup ... 230
Objektově orientované principy a terminologie ... 231
Vlastní třídy ... 234
Atributy a metody ... 234
Dědičnost a polymorfismus ... 239

Řízení přístupu k atributům pomocí vlastností ... 241
Tvorba kompletních, plně integrovaných datových typů ... 243
Vlastní třídy představující kolekce ... 255
Tvorba tříd agregujících kolekce ... 256
Tvorba tříd představujících kolekce pomocí agregace ... 262
Tvorba tříd představujících kolekce pomocí dědičnosti ... 269
Shrnutí ... 275
Cvičení ... 277

Lekce 7
Práce se soubory ... 279
Zapisování a čtení binárních dat ... 284
Naložené objekty s volitelnou kompresí ... 285
Holá binární data s volitelnou kompresí ... 288
Zapisování a analyzování textových souborů ... 297
Zapisování textu ... 297
Analyzování textu ... 298
Analyzování textu pomocí regulárních výrazů ... 301
Zapisování a analyzování souborů XML ... 303
Stromy elementů ... 304
Model DOM (Document Object Model) ... 307
Ruční zápis kódu jazyka XML ... 310
Analýza kódu jazyka XML pomocí rozhraní SAX (Simple API for XML) ... 311
Binární soubory s náhodným přístupem ... 314
Generická třída BinaryRecordFile ... 314
Příklad: Třídy modulu BikeStock ... 322
Shrnutí ... 326
Cvičení ... 327

Lekce 8
Pokročilé techniky programování ... 329
Další techniky procedurálního programování ... 330
Větvení pomocí slovníků ... 331
Generátorové výrazy a funkce ... 332
Dynamické provádění kódu a dynamické importy ... 334
Lokální a rekurzivní funkce ... 341
Dekorátory funkcí a metod ... 345
Anotace funkcí ... 349

Další objektově orientované programování ... 351
Řízení přístupu k atributům ... 352
Funktory ... 355
Správce kontextu ... 357
Deskriptory ... 360
Dekorátory tříd ... 365
Abstraktní bázové třídy ... 368
Vícenásobná dědičnost ... 375
Metatřídy ... 377
Funkcionální styl programování ... 381
Částečná aplikace funkce ... 384
Korutiny ... 385
Příklad: Valid.py ... 393
Shrnutí ... 395
Cvičení ... 396

Lekce 9

Ladění, testování a profilování ... 399
Ladění ... 400
Syntaktické chyby ... 401
Chyby za běhu programu ... 402
Vědecké ladění ... 406
Testování jednotek ... 410
Profilování ... 416
Shrnutí ... 420

Lekce 10

Procesy a vlákna ... 423
Modul pro práci s více procesy ... 424
Modul pro práci s vlákny ... 428
Příklad: Vícevláknový program pro hledání slova ... 429
Příklad: Vícevláknový program pro hledání duplicitních souborů ... 432
Shrnutí ... 437
Cvičení ... 438

Lekce 11
Propojení v síti ... 439
Tvorba klienta TCP ... 441
Tvorba serveru TCP ... 446
Shrnutí ... 452
Cvičení ... 453

Lekce 12
Programování databází ... 455
Databáze DBM ... 456
Databáze SQL ... 460
Shrnutí ... 467
Cvičení ... 468

Lekce 13
Regulární výrazy ... 469
Jazyk Pythonu pro regulární výrazy ... 471
Znaky a třídy znaků ... 471
Kvantifikátory ... 472
Seskupování a zachytávání ... 474
Aserce a příznaky ... 475
Modul pro regulární výrazy ... 479
Shrnutí ... 488
Cvičení ... 489

Lekce 14
Úvod do syntaktické analýzy ... 491
Terminologie formy BNF a syntaktické analýzy ... 493
Ruční tvorba analyzátorů ... 497
Analyzování jednoduchých dat ve tvaru klíč-hodnota ... 497
Analyzování seznamu skladeb ... 500
Analýza bloků jakožto doménově specifického jazyka ... 502
Syntaktická analýza ve stylu jazyka Python pomocí nástroje PyParsing ... 511
Stručné seznámení s nástrojem PyParsing ... 511
Jednoduchá analýza dat ve tvaru klíč-hodnota ... 515
Analyzování seznamu skladeb ... 516

Analýza bloků jakožto doménově specifického jazyka ... 518
Syntaktická analýza logiky prvního řádu ... 523
Syntaktická analýza s nástrojem PLY podle nástrojů Lex a Yacc ... 528
Analyzování jednoduchých dat ve tvaru klíč-hodnota ... 530
Analyzování seznamu skladeb ... 532
Analýza bloků jakožto doménově specifického jazyka ... 534
Syntaktická analýza logiky prvního řádu ... 536
Shrnutí ... 540
Cvičení ... 541

Lekce 15
Seznámení s programováním grafického uživatelského rozhraní ... 543
Programy ve stylu dialogových oken ... 547
Programy s hlavním oknem ... 552
Vytvoření hlavního okna ... 552
Vytvoření vlastního dialogového okna ... 563
Shrnutí ... 565
Cvičení ... 566

Závěrem ... 569

Rejstřík ... 571

Úvod

Python je pravděpodobně nejsnadněji osvojitelný programovací jazyk, který se nejkrásněji používá. Kód jazyka Python je srozumitelný pro čtení i zápis a k tomu je stručný bez jakéhokoli nádechu tajemna. Python je velmi expresivní jazyk, což znamená, že obvykle stačí napsat daleko méně řádků kódu jazyka Python, než kolik by jich bylo zapotřebí pro ekvivalentní aplikace napsanou třeba v jazyku C++ nebo Java.

Python je multiplatformní jazyk. Obecně lze tedy říci, že program napsaný v jazyku Python lze spustit ve Windows i v unixových systémech, jako je Linux, BSD a Mac OS X, pouhým zkopírováním souboru či souborů, které tvoří daný program, na cílový stroj, aniž by jej bylo nutné „sestavovat" nebo kompilovat. Je možné vytvářet programy napsané v Pythonu, které používají funkčnost specifickou pro určitou platformu. To ale jen zřídkakdy nezbytné, protože téměř celá standardní knihovna Pythonu a většina knihoven třetích stran jsou plně a transparentně multiplatformní.

Jednou z opravdu silných stránek Pythonu je, že se dodává se skutečně kompletní standardní knihovnou, díky čemuž můžeme provádět třeba stahování souboru z Internetu, rozbalování zkomprimovaného archivního souboru nebo vytváření webového serveru jen pomocí jediného nebo několika málo řádků kódu. A kromě standardní knihovny je k dispozici tisíce knihoven třetích stran, z nichž některé poskytují ve srovnání se standardní knihovnou výkonnější a sofistikovanější možnosti (např. síťová knihovna Twisted nebo numerická knihovna NumPy), zatímco jiné poskytují funkčnost, která je příliš specializovaná na to, aby byla zahrnuta do standardní knihovny (např. simulační balíček SimPy). Většina knihoven třetích stran je k dispozici v seznamu balíčků pro jazyk Python (*pypi.python.org/pypi*).

V jazyku Python lze programovat v procedurálním, objektově orientovaném a v menší míře též funkcionálním stylu, i když v jádru je Pythonu objektově orientovaným jazykem. V této knize si ukážeme, jak psát procedurální a objektově orientované programy, a osvojíme si též prvky funkcionálního programování v jazyku Python.

Účelem této knihy je prezentovat způsob, jakým psát programy ve správném stylu Pythonu 3, a po přečtení se stát užitečnou příručkou pro jazyk Python 3. Přestože Python 3 je spíše evolučním nežli revolučním pokračováním Pythonu 2, nejsou u něj starší postupy již vhodné nebo nezbytné, přičemž se objevilo několik nových, využívajících přednosti Pythonu 3. Python 3 je lepší jazyk než Python 2 – je totiž postaven na mnohaleté zkušenosti s Pythonem 2 a přidává spoustu nových možností (a současně vypouští ty, které se v Pythonu 2 neosvědčily), díky nimž je programování ještě příjemnější, pohodlnější, snazší a konzistentnější.

Cílem knihy je naučit *jazyk* Python, a přestože se v ní seznámíte s množstvím standardních knihoven Pythonu, nesetkáte se se všemi. To ale není žádný problém, protože po přečtení knihy budete mít o Pythonu dost znalostí na to, abyste použili jakoukoli ze standardních knihoven nebo z knihoven třetích stran, a také na to, abyste byly schopni vytvářet své vlastní knihovní moduly.

Kniha je navržena tak, aby byla užitečná pro různé skupiny čtenářů, mezi něž patří samouci a amatérští programátoři, studenti, vědci, inženýři a všichni ostatní, kteří potřebují v rámci své práce něco naprogramovat, a samozřejmě také profesionální vývojáři a počítačový odborníci. Ovšem k tomu, aby byla kniha použitelná pro tak široké spektrum čtenářů, aniž by přitom znalé nudila nebo méně zkušené ztrácela, musí předpokládat alespoň nějaké zkušenosti s programováním (v libovolném jazyku). Především předpokládá základní znalosti v oblasti datových typů (jako jsou čísla a řetězce), datových typů představujících kolekce (jako jsou množiny a seznamy), řídících struktur (jako jsou příkazy `if` a `while`) a funkcí. Kromě toho některé příklady a cvičení předpokládají základní znalost značkovacího jazyka HTML a některé ze specializovanějších lekcí na konci vyžadují alespoň základní orientaci v probíraném tématu. Například Lekce o databázích předpokládá základní znalost jazyka SQL.

Kniha je uspořádána s ohledem na maximální možnou produktivitu a rychlost. Na konci první lekce budete schopni psát v jazyku Python malé, ale užitečné programy. V každé další lekci se seznámíte s novými tématy a zároveň témata probíraná v předchozích lekcích budete často rozšiřovat a prohlubovat. To znamená, že při postupném pročítání jednotlivých lekcí můžeme kdykoliv přestat– a s dosud získanými znalostmi budete schopni psát ucelené programy. Potom se můžete samozřejmě pustit do dalšího čtení a naučit se pokročilejší a sofistikovanější techniky. Z tohoto důvodu se s některými tématy seznámíte v jedné lekci a pak je blíže prozkoumáte v další či v několika pozdějších lekcích.

Při výuce nového programovacího jazyka se objevují dva hlavní problémy. Prvním je, že někdy, když je nutné se naučit nějaký nový princip, tento princip závisí na jiném, který zase přímo či nepřímo závisí na tom prvním. Druhý problém tkví v tom, že na začátku může čtenář o jazyku vědět jen něco málo neb vůbec nic, takže je velice obtížné prezentovat zajímavé nebo užitečné příklady či cvičení. V této knize se budeme snažit vyřešit oba problémy. První předpokládáním nějakým předchozích zkušeností s programováním a druhý představením „nádherného srdce“ jazyka Python v lekci 1, což je osm klíčových oblastí jazyka Python, které jsou samy o sobě dostatečné pro tvorbu ucházejících programů. Důsledkem tohoto přístupu je, že v prvních lekcích jsou některé příklady v trošičku umělém stylu, poněvadž používají pouze to, co jsme se do místa jejich prezentace naučili. Tento vliv se s každou další lekcí zmenšuje, a to až do konce lekce 7, kde jsou všechny příklady zapsány stylem, který je pro Python 3 naprosto přirozený.

Přístup knihy je veskrze praktický, takže budete vyzýváni, abyste si příklady a cvičení sami vyzkoušeli a získali tak určitou praxi. Kdykoliv to bude možné, použijeme pro příklady kompletní programy a moduly představující realistické případy užití. Příklady, řešení pro cvičení a errata ke knize jsou k dispozici na stránce *http://knihy.cpress.cz/K1747*.

I když je nejlepší používat nejnovější verzi Pythonu 3, nemusí to být vždy možné, pokud uživatelé nemohou nebo nechtějí svoji verzi Pythonu modernizovat. Každý příklad v této knize funguje s Pythonem 3.0, přičemž příklady a funkční prvky specifické pro Python 3.1 jsou výslovně uvedeny.

Přestože je možné tuto knihu použít pro vývoj softwaru, který používá pouze Python 3.0, měli by všichni, kteří chtějí vytvářet software, který se bude používat řadu let a který by měl být kompatibilní s pozdějšími vydáními Pythonu 3.x, používat Python ve verzi 3.1 a podporovat tuto verzi jako nejstarší verzi Pythonu 3. To je dáno zčásti tím, že Python 3.1 nabízí několik velice pěkných nových možností, ale především tím, že vývojáři Pythonu důrazně doporučují používat Python 3.1 (nebo

novější). Vývojáři se rozhodli, že Python 3.0.1 bude posledním vydáním v řadě 3.0.y a že již žádná další vydání v této řadě nebudou, a to ani tehdy, pokud se objeví nějaké chyby či bezpečnostní problémy. Chtějí totiž, aby všichni uživatelé Pythonu 3 přešli k Pythonu 3.1 (nebo k novější verzi), který bude mít běžná vydání s opravami chyb a bezpečnostních problémů.

Uspořádání knihy

Lekce 1 prezentuje osm klíčových oblastí jazyka Python, které jsou dostatečné pro psaní kompletních programů. Dále popisuje některá z dostupných programovacích prostředí Pythonu a prezentuje dva malinké programy sestavené s využitím osmi klíčových oblastí jazyka Python probíraných v dřívější části lekce.

Lekce 2 až 5 představují prvky procedurálního programování jazyka Python, včetně jeho základních datových typů, datových typů představujících kolekce a řady užitečných vestavěných funkcí a řídících struktur společně s velmi jednoduchou prací se soubory. Lekce 5 ukazuje, jak vytvářet vlastní moduly a balíčky, a poskytuje přehled standardní knihovny Pythonu, abyste měli dobrou představu o funkcích, které jsou v Pythonu ihned k dispozici – a díky kterým nemusíte znovu objevovat kolo.

Lekce 6 poskytuje důkladné seznámení s objektově orientovaným programováním v jazyku Python. Veškerá látka týkající se procedurálního programování, kterou jste se naučili v předchozích lekcích, i nadále platí, protože objektově orientované programování je postaveno na procedurálních základech. Využívá tak například stejné datové typy, datové typy představující kolekce a řídící struktury.

Lekce 7 se věnuje zápisu a čtení souborů. V případě binárních souborů se navíc jedná o kompresi a náhodný přístup a u textových souborů o syntaktickou analýzu prováděnou ručně a pomocí regulárních výrazů. Tato Lekce dále ukazuje, jak zapisovat a číst soubory XML, včetně použití stromů elementů, modelu DOM (Document Object Model – objektový model dokumentu) a rozhraní SAX (Simple API for XML – jednoduché aplikační rozhraní pro XML).

Lekce 8 reviduje látku probíranou v několika předchozích lekcích a prozkoumává řadu pokročilejších prvků jazyka Python v oblasti datových typů a datových typů představujících kolekce, řídících struktur, funkcí a objektově orientovaného programování. Tato Lekce dále představuje spoustu nových funkcí, tříd a pokročilých technologií, včetně funkcionálního stylu programování a použití korutin. Probíraná témata jsou sice náročná, ale zato velice užitečná.

Lekce 9 se od všech předchozích lekcí liší v tom, že místo představování nových prvků jazyka Python probírá techniky a knihovny pro ladění, testování a profilování programů.

Zbývající lekce se věnují nejrůznějším pokročilým tématům. Lekce 10 ukazuje techniky pro rozložení pracovní zátěže programu do více procesů nebo vláken. Lekce 11 ukazuje, jak pomocí standardní podpory Pythonu pro komunikace přes síť vytvářet aplikace s architekturou klient-server. Lekce 12 se věnuje databázovému programování (jednoduché soubory DBM s daty ve tvaru klíč-hodnota i databáze SQL).

Lekce 13 vysvětluje a demonstruje minijazyk regulárních výrazů v Pythonu a věnuje se modulu pro regulární výrazy. Lekce 14 pokračuje dále a ukazuje základní techniky syntaktické analýzy pomocí regulárních výrazů a také použití dvou modulů třetích stran, PyParsing a PLY. Nakonec Lekce 15 představuje programování grafického uživatelského rozhraní (Graphical User Interface neboli GUI)

pomocí modulu tkinter, který je součástí standardní knihovny Pythonu. Kniha má dále velmi stručný závěr a samozřejmě rejstřík.

Mnohé lekce jsou pro udržení související látky na jednom místě docela dlouhé. Nicméně lekce jsou rozděleny na části, oddíly a někdy i pododdíly, takže je lze číst takovým tempem, které vám nejlépe vyhovuje – třeba přečtením jedné části nebo jednoho oddílu najednou.

Získání a instalace Pythonu 3

Máte-li moderní a aktualizovaný unixový systém nebo Mac, pak již máte Python 3 nejspíše nainstalovaný, což ověříte zapsáním příkazu `python -V` (jedná se o velké písmeno V) do konzoly (`Terminal.app` v systému Mac OS X). Jedná-li se o verzi 3.x, pak je Python 3 již přítomen, takže nemusíte nic instalovat. Pokud Python nebyl vůbec nalezen, může to být tím, že má název, který obsahuje číslo verze. Zkuste napsat `python3 -V`, a pokud ani to nefunguje, tak `python3.0 -V` nebo `python3.1 -V`. Pokud některá z těchto možností funguje, pak víte, že již máte Python nainstalovaný, a znáte jeho verzi i název. (V této knize používáme název `python3`, můžeme ale používat takový název, který u vás funguje, například `python3.1`.) Pokud nemáte nainstalovanou žádnou verzi Pythonu 3, čtěte dále.

Pro systémy Windows a Mac OS X jsou k dispozici snadno použitelné grafické instalační balíčky, které vás provedou instalačním procesem krok za krokem. Můžete je stáhnout na adrese *www.python.org/download*. Pro Windows stáhněte balíček „Windows x86 MSI Installer", pokud si ovšem nejste jisti, že váš stroj má jiný procesor, pro který je dodáván jiný instalátor. Máte-li například AMD64, sáhněte po balíčku „Windows X86-64 MSI Installer". Jakmile instalační balíček získáte, stačí jej už jen spustit a řídit se pokyny na obrazovce.

Pro Linux, BSD a další unixové systémy (kromě systému Mac OS X, pro nějž je k dispozici instalační soubor `.dmg`) spočívá nejjednodušší způsob instalace Pythonu v použití systému pro správu balíčků vašeho operačním systému. Ve většině případů je Python k dispozici v několika samostatných balíčcích. Například v systému Ubuntu (od verze 8) existuje `python3.0` pro Python, `idle-python3.0` pro editor IDLE (jednoduché vývojové prostředí) a `python3.0-doc` pro dokumentaci – společně se spoustou dalších balíčků, které vedle standardní knihovny poskytují doplňky s dalšími funkčními prvky. (Pro Python ve verzi 3.1 budou názvy balíčků samozřejmě začínat `python-3.1`.)

Pokud na vašem systému nejsou k dispozici žádné balíčky s Pythonem 3, pak musíte stáhnout zdrojový kód z adresy *www.python.org/download* a sestavit Python úplně od začátku. Stáhněte jeden z archivů tarball se zdroji a v případě komprese gzip jej rozbalte příkazem `tar xvfz Python-3.1.tgz` nebo v případě komprese bzip2 příkazem tar `xvfj Python-3.1.tar.bz2`. (Číslo verze se může lišit, například `Python-3.1.1.tgz` nebo `Python-3.1.2.tar.bz2`, ale stačí jednoduše nahradit 3.1 skutečným číslem verze.) Konfigurace sestavení probíhá standardním způsobem. Nejdříve se přesuňte do nově vytvořeného adresáře Python-3.1 a spusťte `./configure`. (Pro lokální instalaci můžete použít volbu `--prefix`.) Dále spusťte `make`.

Je možné, že na konci obdržíte několik zpráv oznamujících, že ne všechny moduly bylo možné sestavit. To obvykle znamená, že na svém počítači nemáte některé z požadovaných knihoven nebo hlaviček. Pokud například nelze sestavit modul `readline`, použijte systém pro správu balíčků pro nainstalování odpovídající vývojové knihovny – například `readline-devel` na systémech na bázi distribuci Fedora nebo `readline-dev` na systémech na bázi distribuce Debian, jako je například Ubuntu. Další

modul, který se nemusí ihned sestavit, je modul `tkinter`, který závisí na vývojových knihovnách Tcl a Tk, což jsou moduly `tcl-devel` a `tk-devel` na systémech na bázi distribuce Fedora a moduly `tcl8.5-dev` a `tk8.5-dev` na systémech na bázi distribuce Debian (s tím, že vedlejší verze nemusí být 5). Naneštěstí nejsou názvy příslušných balíčků na první pohled zřejmé, a proto může být nutné obrátit se s žádostí o pomoc na diskuzní fórum Pythonu. Po nainstalování chybějících balíčků spusťte znovu `./configure` a `make`.

Po úspěšně provedeném příkazu `make` se můžete spuštěním příkazu `make test` přesvědčit, zda je všechno v pořádku. Není to ale nezbytné a navíc může dokončení tohoto příkazu trvat spoustu minut.

Pokud použijete volbu `--prefix` pro lokální instalaci, pak stačí spustit `make install`. Pokud v případě Pythonu 3.1 instalujete třeba do adresáře `~/local/python31`, pak přidáním adresáře `~/local/python31/bin` do své proměnné prostředí `PATH` budete schopni spouštět Python příkazem `python3` a editor IDLE příkazem `idle3`. Pokud již máte lokální adresář pro spustitelné soubory, který se nachází v proměnné prostředí `PATH` (např. `~/bin`), pak můžete místo změny proměnné `PATH` přidat symbolické odkazy. Máte-li spustitelné soubory například v adresáři `~/bin` a Python jste nainstalovali do adresáře `~/local/python31`, pak můžete vytvořit vhodné odkazy spuštěním příkazů `ln -s ~/local/python31/bin/python3 ~/bin/python3` a `~/local/python31/bin/idle3 ~/bin/idle3`. Pro účely této knihy jsme v systémech Linux a Mac OS X přesně takto provedli lokální instalaci a přidali symbolické odkazy, přičemž ve Windows jsme použili binární instalátor.

Pokud nepoužijete volbu `--prefix` a máte přístup uživatele „root", přihlaste se jako „root" a proveďte příkaz `make install`. Na systémech podporujících příkaz `sudo`, jako je například Ubuntu, spusťte příkaz `sudo make install`. Je-li v systému Python 2, adresář `/usr/bin/python` se nezmění a Python 3 bude dostupný jako `python3.0` (nebo `python3.1` podle nainstalované verze) a od verze Python 3.1 také jako `python3`. Editor IDLE pro Python 3.0 se nainstaluje jako `idle`, takže pokud potřebujete i nadále přístup k editoru IDLE pro Python 2, musíte před provedením instalace starý editor IDLE přejmenovat (např. na `/usr/bin/idle2`). Python 3.1 nainstaluje editor IDLE jako `idle3`, takže k žádnému konfliktu s editorem IDLE pro Python 2 nedochází.

Poděkování

Nejdříve bych chtěl poděkovat za odezvu, kterou jsem obdržel od čtenářů první edice, kteří mi poskytli připomínky ohledně oprav, návrhů nebo obojího.

Mé další poděkování míří k odborným recenzentům knihy, počínaje Jasminem Blanchettem, který je počítačovým odborníkem, programátorem a spisovatelem, s nímž jsem spolupracoval na dvou knihách o C++ a knihovně Qt. Jeho zapojení do plánování lekcí, jeho rady, kritika všech příkladů i jeho pečlivé čtení významným způsobem zlepšily kvalitu této knihy.

Georg Brandl je přední vývojář a dokumentátor v oblasti Pythonu odpovědný za vytvoření nové sady dokumentačních nástrojů. Všiml si spousty zákeřných chyb a velice trpělivě a neústupně je vysvětloval, dokud nebyly pochopeny a opraveny. Dále provedl řadu zlepšení v rámci příkladů.

Phil Thompson je expertem na jazyk Python a tvůrcem knihovny PyQt, což je pravděpodobně nejlepší knihovna GUI pro Python. Jeho bystrozraká a podnětná odezva vedla k řadě vyjasnění a korekcí.

Trenton Schulz je hlavní softwarový inženýr ve společnosti Qt Software (před odkoupením společností Nokia známé jako Trolltech), který byl cenným recenzentem všech mých předchozích knih a který mi opět přišel na pomoc. Pozorně přečetl a množství jeho připomínek napomohlo k ujasnění řady problémů a vedlo k značným zlepšením v textu.

Kromě výše zmíněných recenzentů, z nichž každý přečetl celou knihu, nesmím zapomenout na Davida Boddieho, předního autora odborných titulů ve společnosti Qt Software, zkušeného odborníka na jazyk Python a vývojáře softwaru s otevřeným zdrojovým kódem, který přečetl a poskytl cennou odezvu na několik částí této knihy.

Pro tuto druhou edici bych také rád poděkoval Paulu McGuireovi (autorovi modulu PyParsing), který byl tak laskav a zkontroloval příklady využívající modul PyParsing, které se objevily v nové lekci věnované syntaktické analýze, a který mi poskytl spoustu uvážených a užitečných rad. A pro stejnou lekci zkontroloval David Beazley (autor modulu PLY) příklady využívající modul PLY a postaral se o cennou odezvu. Kromě toho Jasmin Blauchebt, Treon Schulz, Georg Braudla Phil Thompson přečetli většinu z nového materiálu této druhé edice a poskytli mi velice hodnotnou zpětnou vazbu.

Díky patří také Guidovi van Rossumovi, tvůrci jazyka Python, jakož i širší komunitě kolem Pythonu, která se významným způsobem podílela na tvorbě Pythonu a zvláště jeho knihoven, které jsou nesmírně užitečné a které je radost používat.

A jako vždy děkuji Jeffu Kingstonovi, tvůrci jazyka Lout pro sazbu písma, který používám již více než deset let.

Zvláštní díky patří mé redaktorce Debře Williams Cauley za její podporu a také za to, že se opět postarala, aby měl celý proces co nejhladší průběh. Děkuji též Anně Popick, která se tak dobře starala o produkční proces, a korektorovi Audrey Doyle, který opět odvedl naprosto skvělou práci. A v souvislosti s touto druhou edicí chci též poděkovat Jennifer Lindnerové za pomoc při udržování nového materiálu na srozumitelné úrovni a japonskému překladateli první edice Takahiro Nagaovi za odhalení zákeřných chyb, které jsem měl možnost v této edici opravit.

V neposlední řadě bych chtěl poděkovat své ženě Andree za to, že zvládla mé buzení ve čtyři hodiny ráno, kdy často přicházely nápady a opravy kódu, které se tu a tam dožadovaly poznamenání nebo otestování, a za její lásku, věrnost a podporu.

LEKCE 1

Rychlý úvod do procedurálního programování

V této lekci:

- Tvorba a spouštění programů napsaných v jazyku Python
- Nádherné srdce jazyka Python

Tato Lekce vás vybaví všemi informacemi, které jsou nezbytné k tomu, abyste mohli v jazyku Python začít psát své programy. Důrazně doporučujeme nainstalovat Python, pokud jste tak již neučinili, abyste si mohli vše, co se zde naučíte, ihned vyzkoušet (vysvětlení způsobu získání a instalace Pythonu na všechny přední platformy najdete v Úvodu).

V první části této lekce si ukážeme, jak vytvářet a spouštět programy napsané v jazyku Python. K psaní kódu jazyka Python můžete používat svůj oblíbený textový editor. Na druhou stranu programovací prostředí IDLE probírané v této části nabízí kromě editoru kódu také doplňkové funkce, mezi něž patří prvky pro experimentování s kódem jazyka Python a pro ladění programů napsaných v jazyku Python.

Ve druhé části se seznámíte s osmi klíčovými částmi jazyka Python, které jsou samy o sobě dostatečné pro vytváření užitečných programů. Všem těmto částem se budeme podrobně věnovat v dalších lekcích, přičemž v průběhu knihy budeme doplňovat zbývající prvky jazyka Python, takže na jejím konci budete znát celý jazyk a budete schopni použít vše, co nabízí, ve svých vlastních programech.

V poslední části této lekce si ukážeme dva krátké programy, které používají jistou podmnožinu prvků jazyka Python, které jsme si představili ve druhé části, takže si ihned vyzkoušíte, jak se v jazyku Python programuje.

Tvorba a spouštění programů napsaných v jazyku Python

Kódování znaků ➤ 95

Kód jazyka Python lze psát pomocí libovolného textového editoru, který dokáže načítat a ukládat text v kódování ASCII nebo UTF-8 znakové sady Unicode. U souborů s kódem jazyka Python se standardně předpokládá, že používají kódování UTF-8, což je nadmnožina kódování ASCII, která dokáže docela dobře reprezentovat libovolný znak libovolného jazyka. Soubory s kódem jazyka Python mají obvykle příponu `.py`, i když na některých systémech na bázi Unixu (např. Linux a Mac OS X) jsou některé aplikace napsané v jazyku Python bez přípony. Programy napsané v jazyku Python využívající grafické uživatelské rozhraní (GUI) mají většinou příponu `.pyw`, především na systémech Windows a Mac OS X. V této knize budeme pro konzolové programy Pythonu a pro moduly Pythonu používat vždy příponu `.py` a pro programy GUI příponu `.pyw`. Všechny příklady uvedené v této knize lze spustit beze změny na všech platformách, na nichž je dostupný Python 3.

Pro jistotu, že je vše správně připraveno, a také pro demonstraci klasického prvního příkladu, vytvořte v obyčejném textovém editoru (např. Poznámkový blok – vzápětí si ukážeme lepší) soubor s názvem `hello.py` a s následujícím obsahem:

```
#!/usr/bin/env python3

print("Ahoj", "světe!")
```

Na prvním řádku je komentář. Komentář v jazyku Python začíná znakem # a pokračuje až na konec řádku (smysl výše uvedeného komentáře si vysvětlíme vzápětí). Druhý řádek je prázdný – Python sice prázdné řádky ignoruje, ale lidskému oku se větší bloky čtou lépe, jsou-li rozdělené. Na třetím řádku je kód jazyka Python. Zde voláme funkci `print()` se dvěma argumenty, z nichž každý je typu

str (řetězec, tj. posloupnost znaků). Všechny příkazy uvedené v souboru .py se provádějí postupně, přičemž se začíná prvním příkazem a pokračuje se po jednotlivých řádcích. To je odlišné od některých jiných jazyků, jako je například C++ nebo Java, které musejí obsahovat určitou zahajovací funkci či metodu se zvláštním názvem. Tok programu lze samozřejmě korigovat, o čemž se přesvědčíme v následující části při probírání řídících struktur jazyka Python.

Budeme předpokládat, že uživatelé systému Windows mají svůj kód jazyka Python v adresáři C:\py3eg a uživatelé systému na bázi Unixu (např. Unix, Linux nebo Mac OS X) jej mají umístěný v adresáři $HOME/py3eg. Soubor hello.py tedy uložte do adresáře py3eg a zavřete textový editor.

Program máme hotový, takže jej můžeme spustit. Programy napsané v Pythonu se spouštějí pomocí interpretu jazyka Python, což se obvykle provádí uvnitř okna příkazového řádku. Tomu se ve Windows říká „Konzola“ nebo „Příkazový řádek“ a většinou je k dispozici v nabídce **Start → Všechny programy → Příslušenství**. V systému Mac OS X nabízí konzoli program Terminal.app (umístěný standardně ve skupině Applications/Utilities), k němuž se dostanete přes Finder, přičemž na ostatních systémech na bázi Unixu můžete použít xterm nebo konzolu poskytovanou okénkovým systémem (např. konsole nebo gnome-terminal).

Spusťte konzolu a ve Windows napište následující povely (u nichž předpokládáme, že je Python nainstalován ve výchozí lokaci) – výstup konzoly je vytištěn normálním písmem, zatímco vámi zadávaný vstup **zvýrazněn**:

```
C:\>cd c:\py3eg
C:\py3eg\>c:\python31\python.exe hello.py
```

Povel cd (změna adresáře) obsahuje absolutní cestu, a proto nezáleží na tom, z jakého adresáře začínáte.

Uživatelé Unixu zadají níže uvedené povely (předpokládáme, že Python 3 je v systémové proměnné PATH):*

```
$ cd $HOME/py3eg
$ python3 hello.py
```

V obou případech by měl být výstup stejný:

```
Ahoj, světe!
```

Všimněte si, že není-li uvedeno jinak, má Python chování v systému Mac OS X úplně stejně jako v kterémkoli jiném systému na bázi Unixu. Kdykoli se tedy budeme odkazovat na Unix, budeme mít na mysli Linux, BSD, Mac OS X a většinu ostatních Unixů a Unixu podobných systémů.

Ačkoliv má náš program jen jeden prováděný příkaz, jeho spuštěním si můžeme odvodit několik informací o funkci print(). Funkce print() je vestavěnou součástí jazyka Python, takže ji nemusíme „importovat“ nebo „začleňovat“ z nějaké knihovny. Dále vidíme, že každý vypisovaný prvek odděluje jednou mezerou a za posledním prvkem vypíše znak nového řádku. Jedná se o výchozí chování, které lze změnit, jak uvidíme později. Další věcí, která stojí za povšimnutí, je skutečnost, že funkce print() může přijímat libovolný počet argumentů.

print() ➤ 179

* Výzva příkazového řádku v systému Unix se může od níže uvedeného znaku $ klidně lišit, což není vůbec podstatné.

Psaní výše uvedených povelů ke spouštění programů napsaných v jazyku Python začne být brzy velmi otravné a pracné. Naštěstí lze ve Windows i v Unixu použít pohodlnější postup. Za předpokladu, že se nacházíme v adresáři `py3eg`, můžeme ve Windows jednoduše napsat:

```
C:\py3eg\>hello.py
```

Jakmile se v konzolu objeví přípona `.py`, zavolá systém Windows pomocí svého registru souborových asociací interpret jazyka Python.

Tento postup však nefunguje za všech okolností, poněvadž některé verze Windows obsahují chybu, která má v některých situacích vliv na provádění interpretovaných programů, které se spouštějí v důsledku asociace s jistou příponou souboru. To se netýká jen Pythonu, ale i další interpretů, a dokonce i některých souborů s příponou `.bat`. Pokud k tomuto problému dojde, tak prostě spusťte Python přímo.

Pokud v systému Windows vypadá váš výstup takto:

```
('Hello', 'World!')
```

znamená to, že se v systému nachází Python 2 a spouští se místo Pythonu 3. Jedno z řešení spočívá ve změně souborové asociace `.py` z Pythonu 2 na Python 3. Dalším řešením (méně pohodlným, ale bezpečnějším) je umístit interpret Pythonu 3 do cesty (předpokládáme, že je nainstalován ve výchozí lokaci) a pokaždé jej explicitně spouštět (tím se vyhnete výše zmíněné chybě systému Windows s asociacemi souborů):

```
C:\py3eg\>path=c:\python31;%path%
C:\py3eg\>python hello.py
```

Mnohem pohodlnější je vytvořit si soubor py3.bat s jediným řádkem `path=c:\python31;%path%` a uložit jej do adresáře `C:\Windows`. Kdykoliv pak spustíte konzolu s úmyslem provádět programy napsané v jazyku Python 3, tak nejdříve spustíte soubor `py3.bat`. Další možností je nechat si soubor `py3.bat` spouštět automaticky. K tomu stačí otevřít vlastnosti konzoly (v nabídce **Start** vyhledejte konzolu, klepněte na ni pravým tlačítkem a zvolte příkaz **Vlastnosti**) a v záložce **Zástupce** (**Shortcut**) přidejte do pole **Cíl** (**Target**) text „`/u /k c:\windows\py3.bat`" (všimněte si mezery před, mezi a za volbami /u a /k a ujistěte se, že je tento text na konci za textem „`cmd.exe`").

V Unixu je nutné nejdříve udělat ze souboru spustitelný soubor, který pak můžeme spustit:

```
$ chmod +x hello.py
$ ./hello.py
```

Příkaz `chmod` stačí pochopitelně spustit pouze jednou. Pak již můžeme napsat `./hello.py` a program se spustí.

Když se v Unixu spustí z konzoly nějaký program, nejdříve se přečtou první dva bajty souboru.* Jsou-li těmito bajty ASCII znaky `#!`, tak shell předpokládá, že soubor spustí interpret, který je specifiko-

* O interakci mezi uživatelem a konzolou se stará program označovaný jako „shell". Rozdíl mezi konzolou a programem shell pro nás není podstatný, takže budeme používat oba výrazy pro označení téhož prostředí.

ván na prvním řádku. Tento řádek se označuje jako *shebang* (shell execute), a pokud je uveden, musí být vždy na prvním řádku souboru.

Řádek shebang má obvykle jednu z následujících podob:

```
#!/usr/bin/python3
```

nebo:

```
#!/usr/bin/env python3
```

Při použití prvního způsobu se použije uvedený interpret. Tento způsob může být nezbytný pro programy napsané v Pythonu, které budou spouštěny webovým serverem, ačkoliv zadaná cesta se samozřejmě může lišit. Při použití druhého způsobu se použije první interpret `python3` nalezený v aktuální prostředí shellu. Druhý způsob je všestrannější, protože interpret Pythonu 3 nemusí být umístěn v adresáři `/usr/bin` (může být například v adresáři `/usr/local/bin` nebo `$HOME`). Řádek shebang není ve Windows nutný (je však naprosto neškodný). Všechny příklady v této knize mají řádek shebang ve druhé z uvedených podob, ačkoliv si jej zde ukazovat nebudeme.

Všimněte si, že u systému na bázi Unixu předpokládáme, že název spustitelného souboru (nebo symbolického odkazu na něj) interpretu jazyka Python 3 v proměnné `PATH` je `python3`. Pokud to neplatí, pak musíte v příkladech upravit řádek shebang tak, aby používal správný název (nebo opravit název a cestu, používáte-li první způsob), nebo vytvořit symbolický odkaz ze spustitelného souboru interpretu Pythonu 3 na název `python3` někde v cestě `PATH`.

Řada výkonných editorů holého textu, jako je Vim nebo Emacs, obsahuje vestavěnou podporu pro úpravu programů psaných v jazyku Python. Tato podpora obvykle zahrnuje barevné zvýrazňování syntaxe a správné odsazovaní řádků. Alternativou je pak programovací prostředí Pythonu s názvem IDLE. V systémech Windows a Mac OS X je toto prostředí standardní součástí instalace Pythonu. V systémech na bázi Unixu se při sestavování z archivu tarball sestavuje prostředí IDLE společně s interpretem jazyka Python, pokud ale používáte správce balíčků, pak můžete prostředí IDLE nainstalovat jako samostatný balíček (viz popis v Úvodu).

Získání a instalace Pythonu 3 ➤ 16

Jak je z obrazovky na obrázku 1.1 patrné, působí IDLE na první pohled poněkud archaickým způsobem připomínajícím doby Motifu na Unixu a Windows 95. To je dáno tím, že namísto moderních knihoven GUI, jako jsou PyGtk, PyQt nebo wxPython, používá knihovnu Tkinter postavenou na bázi Tk (viz Lekce 15). Důvodem pro použití knihovny Tkinter je směsice historie, liberálních licenčních podmínek a skutečnosti, že Tkinter je v porovnání s ostatními knihovnami GUI mnohem menší. Výhodou je navíc to, že prostředí IDLE se standardně dodává s Pythonem a velmi snadno se osvojuje a používá.

Prostředí IDLE poskytuje tři klíčové funkce: možnost zadávat výrazy a kód jazyka Python a sledovat výsledky přímo v okně **Python Shell**, editor kódu, který nabízí barevné zvýrazňování syntaxe a odsazování kódu jazyka Python, a ladicí nástroj, pomocí něhož lze krokovat kód pro snazší identifikaci a odstranění chyb. Okno Python Shell je zvláště užitečné pro zkoušení jednoduchých algoritmů, úryvků kódu a regulárních výrazů a lze jej použít také jako velmi výkonnou a flexibilní kalkulačku.

K dispozici je několik dalších vývojových prostředí pro Python, raději ale používejte IDLE, tedy alespoň zpočátku. Své programy můžete také vytvářet v obyčejném textovém editoru dle vlastního výběru a ladit je pomocí volání funkce `print()`. Interpret jazyka lze vyvolat i bez uvedení programu, který chceme spustit. V takovém případě se interpret spustí v interaktivním režimu, ve kterém je možné zadávat příkazy jazyka Python a sledovat výsledky úplně stejně jako v okně Python Shell v prostředí IDLE, a to včetně téže výzvy příkazového řádku (>>>). Avšak s prostředím IDLE se mnohem snadněji pracuje, a proto byste měli s úryvky kódu experimentovat právě v tomto prostředí. U zde předkládaných krátkých interaktivních příkladů tedy předpokládáme, že se zadávají do interaktivního interpretu jazyka Python nebo do okna Python Shell v prostředí IDLE.

Obrázek 1.1: Okno Python Shell v prostředí IDLE

Nyní již tedy víme, jak se programy napsané v jazyku Python vytvářejí a spouštějí, z jazyka Python jsme se však zatím seznámili pouze s funkcí `print()`. V následující části své znalosti jazyka Python značně rozšíříme, takže budeme schopni vytvářet krátké, ale užitečné programy, což si také v poslední části této lekce vyzkoušíme.

Nádherné srdce jazyka Python

V této části se seznámíme s osmi klíčovými oblastmi jazyka Python a v další části si ukážeme, jak lze pomocí nich napsat několik malých, ale praktických programů. Budete-li mít při pročítání této části pocit, že něco chybí nebo že je výklad příliš rozvláčný, nakoukněte pomocí odkazů, obsahu nebo rejstříku na další stránky knihy. S největší pravděpodobností zjistíte, že prvek, který potřebujete, jazyk Python nejen obsahuje, ale že často nabízí poněkud zhuštěnější formy výrazu, který jsme si zde ukázali, a k tomu ještě mnohem více.

Oblast č. 1: Datové typy

Jednou ze základních věcí, které musí být schopen každý programovací jazyk, je reprezentovat prvky dat. Jazyk Python nabízí hned několik vestavěných datových typů, my se ale prozatím soustředíme pouze na dva z nich. Celočíselné hodnoty (kladná a záporná celá čísla) jsou v Pythonu reprezentovány pomocí typu `int` a řetězce (posloupnosti znaků ze znakové sady Unicode) pomocí typu `str`. Zde je několik příkladů literálů představujících celá čísla a řetězce:

```
-973
210624583337114373395836055367340864637790190801098222508621955072
0
"Někonečně náročné"
'Jakub Krtek'
'suprově αβγ€÷©'
''
```

Druhé výše uvedené číslo je 2^{217}. Velikost celých čísel v jazyku Python totiž není omezena pevně daným počtem bajtů, ale pouze pamětí počítače. Řetězce lze ohraničit dvojitými nebo jednoduchými uvozovkami, podstatné je, aby na obou koncích řetězce byl stejný druh uvozovek. Python používá pro řetězce znakovou sadu Unicode. Řetězce tedy nejsou omezeny jen na znaky ASCII, což je patrné z předposledního řetězce. Prázdný řetězec je prostě takový, který mezi ohraničujícími znaky neobsahuje vůbec nic.

Jazyk Python používá pro přístup k prvkům posloupnosti, jako je například řetězec, hranaté závorky (`[]`). Pokud se například nacházíme v okně Python Shell (ať už v interaktivním interpretu nebo v prostředí IDLE), můžeme zadat následující (výstup v okně Python Shell je vytištěn `normálním písmem`, zatímco vámi zadávaný vstup **`zvýrazněn`**):

```
>>> "Těžké časy"[6]
'č'
>>> "žirafa"[0]
'ž'
```

Okno Python Shell používá standardně pro výzvu svého příkazového řádku znaky `>>>`. Toto nastavení lze však jednoduše změnit. Syntaxi s hranatými závorkami lze použít u datových prvků libovolného datového typu, který představuje nějakou posloupnost, jako jsou řetězce a seznamy. Tato konzistence syntaxe je jedním z důvodů, proč je Python tak výjimečný. Všimněte si, že index v jazyku Python začíná vždy na hodnotě 0.

V jazyku Python jsou typ `str` a základní číselné typy *neměnitelné* (immutable), což znamená, že po nastavení již nelze jejich hodnotu změnit. Na první pohled to může vypadat jako poněkud zvláštní omezení, avšak pro syntaxe jazyka Python to není v praxi žádný problém. Jediným důvodem, proč se o tom zmiňujeme, je fakt, že znak na zadané pozici řetězce sice můžeme získat pomocí hranatých závorek, pro nastavení nového znaku je ale použít nemůžeme. (Je třeba poznamenat, že znak je v jazyku Python jednoduše řetězec s délkou 1.) Pro převod datového prvku jednoho typu na jiný můžeme použít syntaxi `datový_typ(prvek)`:

```
>>> int("45")
45
>>> str(912)
'912'
```

Převod pomocí `int()` je tolerantní vůči úvodnímu a koncovému prázdnému prostoru, takže stejný výsledek získáme také po vyhodnocení výrazu `int(" 45 ")`. Převod pomocí `str()` lze aplikovat na téměř jakýkoliv datový prvek. Podporu převodu pomocí `str()`, `int()` nebo dalších typů lze snadno zařídit také u našich vlastních datových typů, pokud takový převod dává smysl, což si vyzkoušíme v lekci 6. Pokud se převod nezdaří, vyvolá se výjimka. S ošetřováním výjimek se ve stručnosti seznámíme v části „Oblast č. 6", přičemž výjimkám se budeme podrobně věnovat v lekci 4.

Řetězce a celá čísla jsou společně s ostatními vestavěnými datovými typy a některými datovými typy ze standardní knihovny jazyka Python obsahem lekce 2. V této lekci se podíváme také na operace, jež lze aplikovat na neměnitelné posloupnosti, jako jsou řetězce.

Oblast č. 2: Odkazy na objekty

Mělké a hloubkové kopírování ➤ 146

Jakmile máme nějaké datové typy, tak další věcí, kterou potřebujeme, jsou proměnné, v nichž je budeme uchovávat. Jazyk Python nezná proměnné jako takové, nabízí však tzv. *odkazy na objekty*. Co se týče neměnitelných objektů, jako jsou `int` a `str`, pak neexistuje žádný rozeznatelný rozdíl mezi proměnnou a odkazem na objekt. U proměnlivých objektů již rozdíl existuje, v praxi však nemá téměř žádný význam. Z tohoto důvodu budou pro nás oba termíny, proměnná a odkaz na objekt, znamenat totéž.

Nyní se podíváme na několik kratičkých příkladů, které si poté podrobně rozebereme.

```
x = "modrá"
y = "zelená"
z = x
```

Syntaxe má prostý tvar *`odkazNaObjekt = hodnota`*. Nic není třeba deklarovat předem, přičemž není nutné uvádět ani typ hodnoty. Jakmile totiž Python začne provádět první příkaz, vytvoří objekt `str` s textem „modrá" a poté vytvoří odkaz na objekt s názvem `x`, který odkazuje na objekt `str`. Z praktického hlediska tedy můžeme říct, že „proměnné x byl přiřazen řetězec 'modrá'". Druhý příkaz je podobný. Třetí příkaz vytváří nový odkaz na objekt s názvem `z` a nastavuje jej tak, aby odkazoval na tentýž objekt, na který odkazuje objekt `x` (v tomto případě se jedná o objekt `str` obsahující text „modrá").

Operátor = není stejný jako operátor přiřazení proměnné v některých jiných jazycích. Operátor = totiž sváže odkaz na objekt s objektem v paměti. Pokud odkaz na objekt již existuje, pak je jednoduše svázán znovu, tentokráte ale s objektem na pravé straně operátoru =. Pokud odkaz na objekt neexistuje, tak jej operátor = vytvoří.

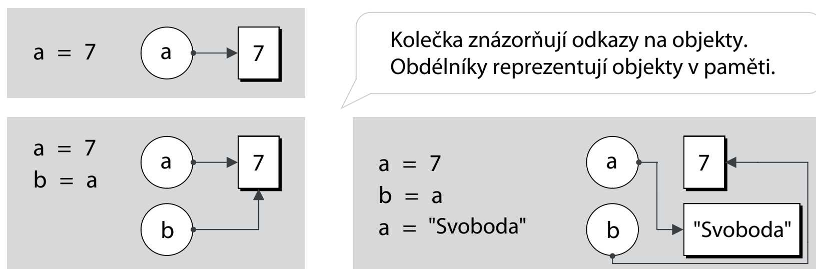
Nyní budeme pokračovat s naším příkladem a provedeme několik opětovných svázání. Jak jsme si řekli již dříve, komentáře začínají znakem # a pokračují až do konce řádku:

```
print(x, y, z) # vypíše: modrá zelená modrá
z = y
```

```
print(x, y, z) # vypíše: modrá zelená zelená
x = z
print(x, y, z) # vypíše: zelená zelená zelená
```

Po čtvrtém příkazu (`x = z`) se všechny tři odkazy na objekt odkazují na tentýž objekt `str`. Vzhledem k tomu, že na řetězec „modrá" se již žádné objekty neodkazují, může jej Python uklidit z paměti.

Obrázek 1.2 schematicky znázorňuje vztah mezi objekty a odkazy na objekty.



Obrázek 1.2: Odkazy na objekty a objekty

Identi-fikátory a klíčová slova ➤ 58

Na názvy používané pro odkazy na objekty (říká se jim *identifikátory*) se vztahuje několik omezení. Nesmějí být totožné s žádným klíčovým slovem jazyka Python a musejí začínat písmenem nebo podtržítkem, za nímž následuje nula nebo více písmen, podtržítek nebo číslic, přičemž nesmí jít o znak představující prázdné místo. Délka není nijak omezena, přičemž písmena a číslice jsou definovány znakovou sadou Unicode, což mimo jiné zahrnuje též znaky a číslice ASCII („a", „b", ..., „z", „A", „B", ..., „Z", „0", „1", ..., „9"). U identifikátorů jazyka Python se rozlišuje velikost písmen, takže například `LIMIT`, `Limit` a `limit` jsou tři různé identifikátory. Další podrobnosti a několik malinko exotických ukázek naleznete v lekci 2.

Jazyk Python používá *dynamickou práci s typy*, což znamená, že odkaz na objekt lze kdykoliv opětovně svázat s jiným objektem (který může být odlišného datového typu). Jazyky, které jsou silně typované (jako například C++ nebo Java), povolují provádění pouze takových operací, které jsou pro dané datové typy definované. Jazyk Python toto omezení také aplikuje, nenazývá se ale silně typovaný jazyk, protože platné operace se mohou změnit. K tomu může dojít třeba v okamžiku, kdy se odkaz na objekt opětovně sváže s objektem jiného datového typu:

```
route = 866
print(route, type(route)) # vypíše: 866 <class 'int'>
route = "Sever"
print(route, type(route)) # vypíše: Sever <class 'str'>
```

Zde vytváříme nový odkaz na objekt s názvem `route` a nastavujeme jej tak, aby odkazoval na nový objekt `int` s hodnotou 866. V tomto okamžiku bychom mohli použít operátor `/`, poněvadž pro celá čísla je dělení platná operace. Odkaz na objekt s názvem `route` pak použijeme znovu tak, aby odkazoval na nový objekt `str` s hodnotou „Server", přičemž objekt `int` je naplánován pro uklizení z paměti, protože se na něj již nic neodkazuje. V tomto okamžiku by použití operátoru `/` způsobilo vyvolání výjimky `TypeError`, neboť operátor `/` není platnou operací pro řetězec.

isinstance() ➢ 238

Funkce `type()` vrací datový typ (označovaný též jako „class“ – „třída“) zadaného datového prvku. Tato funkce je velice užitečná pro testování a ladění, v ostrém kódu by se však již objevit neměla, protože se dá nahradit lepší alternativou, o čemž se přesvědčíme v lekci 6.

Při experimentování s kódem jazyka Python uvnitř interaktivního interpretu nebo v okně Python Shell (např. v prostředí IDLE) pak prostý zápis názvu odkazu na objekt způsobí vypsání jeho hodnoty:

```
>>> x = "modrá"
>>> y = "zelená"
>>> z = x
>>> x
'modrá'
>>> x, y, z
('modrá', 'zelená', 'modrá')
```

To je mnohem pohodlnější řešení než neustálé volání funkce `print()`, funguje ale jen při interaktivní práci s Pythonem. Všechny programy a moduly, které napíšeme, musejí pro výstup nějakých hodnot používat `print()` nebo podobnou funkci. Všimněte si, že Python zobrazil poslední výstup v závorkách a oddělený čárkami. To označuje n-tici, což je uspořádaná, neměnitelná posloupnost objektů. K n-ticím se ještě dostaneme v následujících oblastech.

Oblast č. 3: Datové typy pro kolekce

Často je užitečné uchovávat celou kolekci datových prvků. Jazyk Python nabízí pro kolekce několik datových typů, které mohou uchovávat prvky. Mezi tyto datové typy patří mimo jiné také asociativní pole a množiny. My si zde ale představíme jen dva: `tuple` (n-tice) a `list` (seznam). Pomocí n-tic a seznamů jazyka Python lze uchovávat libovolný počet datových prvků libovolného datového typu. N-tice jsou neměnitelné, takže je po vytvoření již nelze změnit. Seznamy jsou měnitelné, takže lze snadno vkládat a odebírat prvky, kdykoli chceme.

N-tice se vytvářejí pomocí čárek (`,`), jak ukazují tyto příklady (od této chvíle již nebudeme zvýrazňovat vámi zadávanou část kódu):

```
>>> "Dánsko", "Finsko", "Norsko", "Švédsko"
('Dánsko', 'Finsko', 'Norsko', 'Švédsko')
>>> "one",
('one',)
```

Typ tuple ➢ 110

Python vypisuje n-tice uzavřené do závorek. Řada programátorů to napodobuje a při psaní kódu uzavírá n-ticové literály také vždy do závorek. Pokud máme jednoprvkovou n-tici a chceme použít závorky, musíme i tak uvést čárku – například `(1,)`. Prázdnou n-tici vytvoříme pomocí prázdných závorek, tedy `()`. Čárka se používá také k oddělení argumentů při volání funkce, takže pokud chceme jako argument předat n-ticový literál, musíme jej pro jednoznačnost uzavřít do závorek.

Tvorba a volání funkcí ➢ 44

Zde je několik ukázkových seznamů:

```
[1, 4, 9, 16, 25, 36, 49]
['alfa', 'bravo', 'charlie', 'delta', 'echo']
```

```
['zebra', 49, -879, 'hrabáč', 200]
[]
```

Jedna z možností pro vytvoření seznamu, kterou jsme si již ukázali, spočívá v použití hranatých závorek (`[]`). Později se podíváme na další možnosti. Čtvrtý z výše uvedených seznamů je prázdný seznam.

Typ `list` ➤ 115

Ve skutečnosti to funguje tak, že seznamy ani n-tice neuchovávají datové prvky, ale odkazy na objekty. Při vytváření seznamů a n-tic (a také při vkládání prvků v případě seznamů) přijímají tyto kolekce kopie zadávaných odkazů na objekty. V případě literálových prvků, jako jsou čísla a řetězce, se v paměti vytvoří a vhodným způsobem inicializuje objekt příslušného datového typu a poté se vytvoří odkaz na objekt odkazující na daný objekt, přičemž do seznamu nebo n-tice se uloží právě tento odkaz na objekt.

Jako cokoliv v jazyku Python také datové typy pro kolekce jsou objekty, takže datové typy pro kolekce lze vnořovat do dalších datových typů pro kolekce, čímž lze například vytvořit seznamy seznamů. V některých situacích je skutečnost, že seznamy, n-tice a většina ostatních datových typů pro kolekce jazyka Python uchovávají místo objektů odkazy na objekty, velice podstatná. Více se tomuto tématu budeme věnovat v lekci 3.

Mělké a hloubkové kopírování ➤ 146

Při procedurálním programování se volají funkce, kterým se často předávají datové prvky jako argumenty. Například s funkcí `print()` jsme se již setkali. Další v Pythonu často používanou funkcí je funkce `len()`, která jako svůj argument přijímá jediný datový prvek a vrací jeho „délku“ jako objekt `int`. Zde je několik příkladů volání funkce `len()`:

```
>>> len(("jedna",))
1
>>> len([3, 5, 1, 2, "pauza", 5])
6
>>> len("automaticky")
11
```

N-tice, seznamy a řetězce jsou datové typy, u nichž má smysl mluvit o velikost (`Sized`), a proto lze datové prvky libovolného z těchto datových typů předat funkci `len()`. (Při předání funkci len() datového prvku, který nezná pojem velikost, dojde k vyvolání výjimky.)

třída `Sized` ➤ 369

Všechny datové prvky jazyka Python jsou *objekty* (nazývané též *instance*) určitého datového typu (označovaného též jako *třída*). Pro nás budou oba termíny *datový typ* a *třída* znamenat totéž. Jediným podstatným rozdílem mezi objektem a holými prvky s daty nabízenými v některých jiných jazycích (např. vestavěné číselné typy jazyka C++ nebo Java) spočívá v tom, že objekt může mít *metody*. Metoda je v podstatě funkce, která se volá pro určitý objekt. Například typ `list` má metodu `append()`, která připojí zadaný objekt k seznamu:

```
>>> x = ["zebra", 49, -879, "hrabáč", 200]
>>> x.append("další")
>>> x
['zebra', 49, -879, 'hrabáč', 200, 'další']
```

Objekt x ví, že je seznamem (všechny objekty jazyka Python znají svůj vlastní datový typ), takže datový typ nemusíme explicitně uvádět. V implementaci metody append() bude prvním argumentem samotný objekt x. O to se stará Python zcela automaticky v rámci své syntaktické podpory pro metody.

Metoda append() mění původní seznam. To je možné díky tomu, že seznamy jsou měnitelné. Je to také potenciálně efektivnější než vytvoření nového seznamu s původními a nově přidaným prvkem, s nímž by se opětovně svázal náš odkaz na objekt. To platí zvláště pro velmi dlouhé seznamy.

V procedurálním jazyku lze téhož dosáhnout pomocí metody append() datového typu list (což je naprosto platná syntaxe jazyka Python):

```
>>> list.append(x, "extra")
>>> x
['zebra', 49, -879, 'hrabáč', 200, 'další', 'extra']
```

Zde specifikujeme datový typ a jeho metodu, které jako první argument předáváme datový prvek tohoto datového typu, na němž chceme metodu zavolat. Dále pokračujeme jakýmikoli dalšími argumenty metody. (S ohledem na dědičnost existuje mezi těmito dvěma syntaxemi drobný rozdíl. V praxi se nejčastěji používá první z uvedených forem. Dědičnost budeme probírat v lekci 6.)

Pokud vám objektově orientované programování nic neříká, pak vám to může na první pohled připadat trošku zvláštní. Prozatím se spokojte s tím, že Python nabízí konvenční funkce volané ve tvaru *názevFunkce*(*argumenty*) a metody, které se volají stylem *názevObjektu*.*názevMetody*(*argumetny*). (Objektově orientovanému programování se budeme věnovat v lekci 6.)

Operátor tečka („přístup k atributům“) se používá pro přístup k atributům objektu. Atributem může být libovolný druh objektu, i když zatím jsme se ukázali jen atributy ve formě metod. Vzhledem k tomu, že atributem může být objekt, který má také atributy, které zase mohou mít svoje atributy (a tak pořád dál), můžeme pro přístup k určitému atributu použít tolik operátorů tečka, kolik jen potřebujeme.

Typ list má mnoho dalších metod, včetně metody insert(), která se používá k vložení prvku na zadanou pozici, a metody remove(), která odstraní prvek na zadané pozici. Jak jsme si řekli již dříve, indexy jazyka Python začínají vždy od nuly.

Viděli jsme, že pomocí operátoru [] můžeme získat znaky z řetězce, a řekli jsme si, že tento operátor lze použít s libovolnou posloupností. Seznamy jsou posloupnosti, a proto můžeme provádět následující:

```
>>> x
['zebra', 49, -879, 'hrabáč', 200, 'další', 'extra']
>>> x[0]
'zebra'
>>> x[4]
200
```

N-tice jsou také posloupnosti, takže pokud by byl objekt `x` n-ticí, mohli bychom získat jeho prvky pomocí hranatých závorek úplně stejně jako v případě seznamu `x`. Avšak vzhledem k tomu, že seznamy jsou měnitelné (na rozdíl od řetězců a n-tic), můžeme pomocí operátoru `[]` prvky seznamu také nastavovat:

```
>>> x[1] = "čtyřicet devět"
>>> x
['zebra', 'čtyřicet devět', -879, 'hrabáč', 200, 'další', 'extra']
```

Pokud použijeme index, který je mimo rozsah, dojde k vyvolání výjimky (s výjimkami se stručně seznámíme v oblasti č. 5 a podrobně se jim budeme věnovat v lekci 4).

Termín posloupnost jsme použili již několikrát, přičemž jsme spoléhali na neformální porozumění jeho významu – v tomto budeme prozatím pokračovat i nadále. Nicméně jazyk Python přesně definuje, jaké funkce musí posloupnost podporovat, a podobně definuje, jaké funkce musí nabízet objekt podporující velikost. Tato „pravidla“ se samozřejmě týkají mnoha další kategorií, do nichž mohou spadat určité datové typy, což si ukážeme v lekci 8.

Seznamy, n-tice a ostatní vestavěné datové typy pro kolekce jazyka Python jsou obsahem lekce 3.

Oblast č. 4: Logické operátory

Jedním ze základních prvků jakéhokoliv programovacího jazyka jsou jeho logické operace. Jazyk Python nabízí čtyři skupiny logických operací a my si zde představíme základy každé z nich.

Operátor identita

Vzhledem k tomu, že všechny proměnné jazyka Python jsou ve skutečnosti odkazy na objekty, může být někdy užitečné ptát se, zda dva či více odkazů na objekty odkazují na tentýž objekt. K tomuto účelu slouží binární operátor `is`, který vrací hodnotu `True`, pokud se odkaz na objekt na levé straně odkazuje na stejný objekt jako odkaz na objekt na pravé straně. Zde je několik příkladů:

```
>>> a = ["retence", 3, None]
>>> b = ["retence", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
True
```

Všimněte si, že obvykle nemá smysl používat operátor `is` k porovnání objektů `int`, `str` a většiny ostatních datových typů, protože téměř vždy chceme porovnat jejich hodnoty. Ve skutečnosti může vést použití operátoru `is` k porovnání datových prvků k neintuitivním výsledům, což je patrné z předchozího příkladu, kde jsme sice na začátku nastavili `a` a `b` na seznam se stejnými hodnotami, avšak samotné seznamy jsou uloženy jako samostatné objekty typu `list`, a proto při prvním použití operátoru `is` obdržíme hodnotu `False`.

Jednou z výhod porovnávání na základě identity je jeho rychlost. Ta je dána tím, že není nutné prozkoumávat samotné odkazované objekty. Operátoru `is` totiž stačí porovnat pouze paměťové adresy objektů – stejná adresa znamená stejný objekt.

Operátor `is` se nejčastěji používá k porovnání datového prvku s vestavěným objektem `None`, který se většinou používá pro označení neznámé nebo neexistující hodnoty:

```
>>> a = "něco"
>>> b = None
>>> a is not None, b is None
(True, True)
```

K obrácení testu na základě identity se používá `is not`.

Smyslem operátoru identita je zjistit, zda se dva odkazy na objekt odkazují na tentýž objekt, nebo zda je daný objekt `None`. Chceme-li porovnat hodnoty objektů, musíme použít porovnávací operátory.

Porovnávací operátory

Jazyk Python nabízí standardní skupinu binárních porovnávacích operátorů s očekávanou sémantikou: < menší než, <= menší nebo rovno, == rovná se, != nerovná se, >= větší nebo rovnost a > větší než. Tyto operátory porovnávají hodnoty objektů, neboli objekty, na které se ukazují odkazy na objekt použité v porovnávání. Zde je několik příkladů zapsaných do okna Python Shell:

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a < b
True
>>> a <= b, a != b, a >= b, a > b
(True, True, False, False)
```

U celých čísel funguje vše tak, jak bychom očekávali. Podobně funguje také porovnávání řetězců:

```
>>> a = "mnoho cest"
>>> b = "mnoho cest"
>>> a is b
False
>>> a == b
True
```

Porovnávání řetězců ➤ 73

Ačkoliv jsou `a` a `b` odlišné objekty (mají odlišné identity), mají stejné hodnoty, a proto jsou při porovnání stejné. Dávejte si však pozor, protože Python používá pro reprezentaci řetězců znakovou sadu Unicode, a proto může být porovnávání řetězců, jež obsahují znaky mimo kódování ASCII, mnohem komplikovanější, než jak by se na první pohled mohlo zdát (tomuto problému se budeme plně věnovat v lekci 2).

Porovnání dvou řetězců nebo čísel na základě identity (např. pomocí `a is b`) vrátí v některých případech hodnotu `True`, přestože jsme oba objekty přiřadili samostatně. To je dáno tím, že některé implementace jazyka Python opětovně použijí kvůli lepší efektivitě stejný objekt (protože hodnota je stejná a neměnitelná). Z toho plyne, že operátory `==` a `!=` se mají používat k porovnání hodnot a operátor `is` a `is not` pouze k porovnání s objektem `None` nebo pokud skutečně chceme zjistit, zda jsou stejné odkazy na objekty, a ne objekty samotné.

Jednou z krásných vlastností porovnávacích operátorů jazyka Python je možnost jejich řetězení:

```
>>> a = 9
>>> 0 <= a <= 10
True
```

Jedná se o hezčí způsob testování, zda je zadaný datový prvek v určitém rozsahu, než dvě samostatná porovnání spojená logickým operátorem `and`, což je nezbytné ve většině ostatních jazyků. Další předností je to, že se datový prvek vyhodnotí pouze jednou (protože se ve výrazu vyskytuje pouze jednou), což může mít značný význam v případě, kdy je výpočet hodnoty datového prvku drahý nebo pokud má přístup k datovému prvku za následek nějaké vedlejší efekty.

Díky „síle" jazyka Python v oblasti dynamické práce s typy způsobí porovnání, které nedává smysl, vyvolání výjimky:

```
>>> "tři" < 4
Traceback (most recent call last):
...
TypeError: unorderable types: str() < int()
```

Při vyvolání výjimky, která není ošetřena, vypíše Python zpětné volání společně s chybovou zprávou výjimky. Pro srozumitelnost jsme část se zpětným voláním nahradili třemi tečkami.* Ke stejné výjimce typu `TypeError` by došlo také v případě porovnání `"3" < 4`, protože Python se nepokouší uhádnout naše záměry. Správně bychom měli provést explicitní převod (např. `int("3") < 4`) nebo použít porovnatelné typy, tedy buď jen čísla, nebo jen řetězce.

Chyby za běhu programu ➤ 402

Jazyk Python usnadňuje vytváření vlastních datových typů, které lze pěkně integrovat tak, že můžeme kupříkladu vytvořit vlastní číselný typ, který by byl schopen účastnit se porovnávání s vestavěným typem `int`, ne však s řetězci nebo s jinými nečíselnými typy.

Alternativní typ `FuzzyBool` ➤ 250

Operátor příslušnosti

U datových typů, které jsou posloupnostmi nebo kolekcemi, jako například řetězce, seznamy nebo n-tice, můžeme testovat příslušnost pomocí operátoru `in` a nepříslušnost pomocí operátoru `not in`:

```
>>> p = (4, "žába", 9, -33, 9, 2)
>>> 2 in p
True
>>> "pes" not in p
True
```

* Zpětné volání je seznam všech volání provedených od okamžiku, kdy došlo k vyvolání neošetřené výjimky, až po vrchol zásobníku volání.

U seznamů a n-tic používá operátor `in` lineární vyhledávání, které může být pomalé u velkých kolekcí (desítky tisíc a více prvků). Na druhou stranu je operátor `in` velmi rychlý při použití na slovník nebo množinu (oba tyto datové typy pro kolekce budeme probírat v lekci 3). Zde je příklad použití operátoru `in` s řetězcem:

```
>>> phrase = "Pestrobarevná kalkulačka"
>>> "k" in phrase
True
>>> "barev" in phrase
True
```

V případě řetězců lze operátor příslušnosti použít i pro testování podřetězců libovolné délky (jak jsme si řekli již dříve, znak je v podstatě řetězec délky 1).

Logické operátory

Jazyk Python nabízí tři logické (neboli booleovské) operátory: `and`, `or` a `not`. Operátory `and` a `or` používají logiku zkráceného vyhodnocování a vracejí operand, který rozhodl o výsledku – nevracejí logickou hodnotu (pokud tedy samy operandy neobsahují logickou hodnotu). Podívejme se, co to znamená v praxi:

```
>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
>>> two and five
5
>>> five and zero
0
```

Pokud se výraz objeví v kontextu logického výrazu, pak je výsledek vyhodnocen jako logická hodnota, takže předchozí výrazy by se například v příkazu `if` vyhodnotily na `True`, `True` a `False`.

```
>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
0
```

Operátor `or` je podobný. Zde bychom v kontextu logického výrazu obdrželi výsledky `True`, `True`, `True` a `False`.

Unární operátor `not` vyhodnotí své argumenty v kontextu logického výrazu a vždy vrátí logickou hodnotu, takže v případě výše uvedeného příkladu by se výraz `not (zero or nought)` vyhodnotil na `True` a výraz `not two` na `False`.

Oblast č. 5: Příkazy pro řízení toku programu

Již dříve jsme si řekli, že příkazy uvedené v souboru `.py` se provádějí jeden za druhým, přičemž se začíná prvním příkazem a pokračuje se řádek po řádku. Tok programu lze odklonit voláním funkcí či metody nebo řídící strukturou, jako je podmíněná větev nebo příkaz cyklu. Tok programu je též odkloněn při vyvolání výjimky.

V této podčásti se podíváme na příkaz `if` jazyka Python a na jeho cykly `while` a `for`, přičemž o funkcích si více řekneme v Oblasti č. 8 a metody ponecháme na lekci 6. Kromě toho se podíváme na naprosté základy ošetřování výjimek (tomuto tématu se budeme plně věnovat v lekci 4). Nejdříve si ale uděláme jasno v terminologii.

Logický výraz je cokoliv, co lze vyhodnotit na logickou hodnotu (`True` nebo `False`). V jazyku Python se takový výraz vyhodnotí na hodnotu `False`, jedná-li se o předdefinovanou konstantu `False`, o speciální objekt `None`, o prázdnou posloupnost nebo kolekci (např. prázdný řetězec, seznam či n-tice) nebo o číselný datový prvek s hodnotou 0. Cokoliv jiného se vyhodnotí na hodnotu `True`. Při vytváření vlastních datových typů (např. v lekci 6) se můžeme sami rozhodnout, co mají v kontextu logického výrazu vrátit.

V řeči jazyka Python se bloku kódu, což je posloupnost jednoho či více příkazů, nazývá *sada* (suite). Některé prvky syntaxe jazyka Python vyžadují přítomnost sady, a proto Python nabízí klíčové slovo `pass`, které představuje příkaz, který neudělá nic a který lze použít všude tam, kde je vyžadována sada (nebo kde chceme říci, že jsme daný případ zvážili), ale kde není potřeba provést žádné zpracování.

Příkaz if

Obecná syntaxe pro příkaz `if` jazyka Python vypadá takto:*

```
if logický_výraz1:
    sada1
elif logický_výraz2:
    sada2
...
elif logický_výrazN:
    sadaN
else:
    jiná_sada
```

Klauzulí `elif` může být nula či více, přičemž finální klauzule `else` je volitelná. Pokud nás určitý případ zajímá, ale nechceme v případě jeho výskytu nic provádět, můžeme pro sadu takové větve použít klíčové slovo `pass`.

* V této knize používáme výpustek (...) pro reprezentaci řádků, které jsme záměrně skryli.

První věcí, která zarazí programátory zvyklé na jazyk C++ nebo Java, je nepřítomnost jednoduchých či složených závorek. Další podstatnou věcí je dvojtečka. Ta je součástí syntaxe a zpočátku se na ni snadno zapomíná. Dvojtečky se používají také u klauzulí `else`, `elif` a v podstatě na libovolném jiném místě, za nímž následuje sada.

Na rozdíl od jiných programovacích jazyků používá Python odsazení k označení své blokové struktury. Někteří programátoři to nemají rádi, zvláště pokud to ještě nevyzkoušeli, a někteří se k tomuto problému staví až příliš afektovaně. Na druhou stranu se na to dá zvyknout za několik dnů a za několik týdnů či měsíců se vám bude kód bez závorek jevit mnohem hezčí a čitelnější.

Sady se označují pomocí odsazení, a proto se můžeme zeptat, o jaké odsazení se vlastně jedná. Směrnice ohledně stylu programování v jazyku Python doporučují použít pro každou úroveň odsazení čtyři mezery (bez tabulátorů). Většinu moderních textových editorů lze nastavit tak, aby se o to postaraly zcela automaticky (což splňuje editor IDLE a většina ostatních editorů s podporou jazyka Python). Python bude fungovat skvěle s libovolným počtem mezer či tabulátorů nebo se směsicí obou, bude-li používané odsazení konzistentní. V této knize budeme dodržovat oficiální směrnice jazyka Python.

Zde je velmi jednoduchá ukázka příkazu `if`:

```
if x:
    print("x je nenulové")
```

V tomto případě se sada (volání funkce `print()`) provede tehdy, pokud se podmínka (`x`) vyhodnotí na hodnotu `True`.

```
if lines < 1000:
    print("malé")
elif lines < 10000:
    print("střední")
else:
    print("velké")
```

Tohle už je malinko propracovanější příkaz `if`, který vypíše slovo popisující číslo uložené v proměnné `lines`.

Příkaz while

Příkaz `while` se používá k provedení sady nulakrát či vícekrát, přičemž tento počet závisí na stavu logického výrazu cyklu `while`. Zde je syntaxe:

```
while logický_výraz:
    sada
```

Úplná syntaxe cyklu `while` je ve skutečnosti mnohem sofistikovanější, poněvadž podporuje příkazy `break` a `continue` a také volitelnou klauzuli `else`, které se budeme věnovat v lekci 4. Příkaz `break` přepne tok programu na příkaz následující za nejvnitřnějším cyklem, v němž se daný příkaz `break` nachází, což znamená, že z tohoto cyklu vyskočí. Příkaz `continue` přepne tok programu na začátek

cyklu. Oba příkazy `break` a `continue` se obvykle používají uvnitř příkazů `if` k podmíněné změně chování cyklu:

```
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```

Tento cyklus `while` má velmi typickou strukturu a běží do té doby, dokud jsou k dispozici prvky na zpracování (`get_next_item()` a `process_item()` jsou naše vlastní funkce definované na jiném místě. V tomto příkladu obsahuje sada příkazu `while` příkaz `if`, který sám o sobě má další sadu (kterou musí mít) obsahující jediný příkaz `break`.

Příkaz for ... in

Cyklus `for` jazyka Python používá opětovně klíčové slovo `in` (které se v jiném kontextu chová jako operátor příslušnosti) a má následující syntaxi:

```
for proměnná in iterovatelný_objekt:
    sada
```

Podobně jako cyklus `while` také cyklus `for` podporuje oba příkazy `break` a `continue` a má také volitelnou klauzuli `else`. Proměnná je nastavena tak, aby postupně odkazovala na každý objekt v iterovatelném objektu, což je libovolný datový typ, který lze procházet, což zahrnuje řetězce (prochází se po jednotlivých znacích), seznamy, n-tice a další datové typy pro kolekce jazyka Python.

```
for country in ["Dánsko", "Finsko", "Norsko", "Švédsko"]:
    print(country)
```

Zde používáme velice zjednodušený přístup k vypisování seznamu zemí. V praxi se mnohem častěji používá proměnná:

```
countries = ["Dánsko", "Finsko", "Norsko", "Švédsko"]
for country in countries:
    print(country)
```

Ve skutečnosti lze celý seznam (nebo n-tici) vypsat pomocí funkce `print()` i přímo (například `print(countries)`), ale často se kvůli lepší kontrole formátování upřednostňuje výpis kolekce pomocí cyklu `for` (nebo pomocí seznamové komprehenze, které se budeme věnovat později):

Seznamové komprehenze ➤ 120

```
for letter in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
    if letter in "AEIOU":
        print(letter, "je samohláska")
    else:
        print(letter, "je souhláska")
```

V tomto úryvku je první použití klíčového slova `in` součástí příkazu `for`, přičemž proměnná `letter` nabývá v každé iteraci cyklu hodnot `"A"`, `"B"` a tak dále až po `"Z"`. Na druhém řádku úryvku používáme opět `in`, ovšem tentokrát jako operátor testující příslušnost. Všimněte si také, že zde máme vnořené sady. Sadou cyklu `for` je příkaz `if ... else` a obě větvě `if` a `else` mají své vlastní sady.

Základní ošetřování výjimek

Řada funkcí a metod jazyka Python hlásí chyby nebo jiné důležité události vyvoláním určité výjimky. Výjimka je objekt stejně jako jakýkoliv jiný objekt jazyka Python a při převodu na řetězec (např. při výpisu) obdržíme textovou zprávu výjimky. Jednoduchý tvar syntaxe pro ošetřování výjimek vypadá takto:

```
try:
    sada_try
except výjimka1 as proměnná1:
    sada_výjimky1
...
except výjimkaN as proměnnáN:
    sada_výjimkyN
```

Část `as proměnná` je volitelná. Může nás totiž zajímat pouze to, že byla vyvolána určitá výjimka, a ne už text její zprávy.

Úplná syntaxe je mnohem sofistikovanější. Například každá klauzule `except` může ošetřit více výjimek a dále tu je volitelná klauzule `else`. Všechny tyto prvky budeme probírat v lekci 4.

Celé to funguje takto. Pokud se všechny příkazy v sadě bloku `try` provedou bez vyvolání výjimky, bloky `except` se přeskočí. Pokud dojde uvnitř bloku `try` k vyvolání výjimky, předá se řízení okamžitě do sady odpovídající první vyhovující výjimce. To znamená, že žádný příkaz v sadě, který následuje za tím, jenž způsobil výjimku, se již neprovede. Pokud k tomu dojde a navíc je uvedena také část `as proměnná`, pak uvnitř sady ošetřující výjimku bude tato proměnná odkazovat na objekt výjimky.

Chyby za běhu programu ➤ 402

Pokud v bloku ošetřujícím výjimku dojde k výjimce nebo pokud se vyvolá výjimka, která neodpovídá žádnému z bloků `except`, pak se Python podívá po odpovídajícím bloku `except` v dalším obklopujícím oboru platnosti. Hledání vhodné obsluhy výjimky probíhá směrem k vnějším oborům platnosti podle zásobníku volání až do té doby, dokud není nalezena shoda a výjimka obsloužena. Pokud není shoda nalezena, program se ukončí s neošetřenou výjimkou. V takovém případě vypíše Python zpětné volání společně s textovou zprávou výjimky.

Zde je příklad:

```
s = input("zadejte celé číslo: ")
try:
  i = int(s)
  print("zadáno platné celé číslo:", i)
except ValueError as err:
  print(err)
```

Pokud uživatel zadá „3.5“, vypíše se toto:

```
invalid literal for int() with base 10: '3.5'
```

Pokud ale zadá „13“, vypíše se:

```
valid integer entered: 13
```

Mnoho knih považuje ošetřování výjimek za pokročilé téma a odkládá je na co nejpozdější dobu. Ale vyvolávání a zvláště ošetřování výjimek je vzhledem ke způsobu fungování Pythonu naprosto zásadní, takže od této chvíle jej budeme využívat. A jak uvidíme, díky používání obsluh výjimek může být kód mnohem čitelnější, protože se „výjimečné“ případy oddělí od vlastního zpracování, které nás ve skutečnosti jako jediné zajímá.

Oblast č. 6: Aritmetické operátory

Python nabízí kompletní sadu aritmetických operátorů, včetně binárních operátorů pro čtyři základní matematické operace: operátor `+` (sčítání), operátor `-` (odčítání), operátor `*` (násobení) a operátor `/` (dělení). Kromě toho řada datových typů jazyka Python lze použít s operátory rozšířeného přiřazení, jako je += a *=. Operátory `+`, `-` a `*` se v případě operandů jakožto celých čísel chovají dle očekávání:

```
>>> 5 + 6
11
>>> 3 - 7
-4
>>> 4 * 8
32
```

Všimněte si, že operátor `-` může být použit jako unární (negace) nebo binární operátor (odčítání), což je obvyklé u většiny programovacích jazyků. V oblasti dělení však jazyk Python vyčnívá z davu:

```
>>> 12 / 3
4.0
>>> 3 / 2
1.5
```

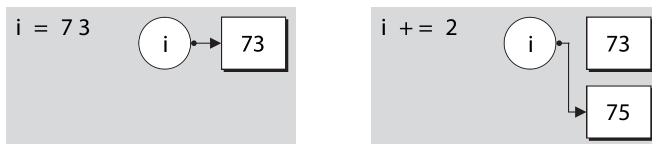
Operátor dělení nevrací celé, ale desetinné číslo. Většina ostatních jazyků vrátí celé číslo vytvořené ořezáním desetinné části. Pokud potřebujeme celočíselný výsledek, můžeme jej vždy převést pomocí `int()` (nebo pomocí ořezávacího operátoru dělení `//`, k němuž se dostaneme později).

Číselné operace a funkce ➢ 61

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

Na první pohled výše uvedené příkazy ničím nepřekvapí, především pak ty, kteří znají jazyky podobné jazyku C. V takovýchto jazycích představuje rozšířené přiřazení zkratku pro přiřazení výsledku operace (např. `a += 8` je v podstatě totéž jako `a = a + 8`). Zde ovšem existují dvě podstatné delikátnosti, jedna specifická pro Python a jedna pro rozšířené operátory v libovolném jazyku.

První věcí, kterou je třeba si zapamatovat, tkví v tom, že datový typ `int` je neměnitelný, což znamená, že po jeho přiřazení již nemůže být jeho hodnota změněna. Při použití operátoru rozšířeného přiřazení na neměnitelný objekt se tedy v pozadí odehraje to, že se provede daná operace a vytvoří se objekt uchovávající výsledek. Poté se cílový odkaz na objekt opětovně sváže tak, aby místo původního objektu odkazoval na objekt s výsledkem. V případě příkazu `a += 8` Python tedy nejdříve vypočítá `a + 8`, výsledek uloží do nového objektu `int` a poté opětovně sváže `a` tak, aby odkazovalo na tento nový objekt `int` (a pokud na původní objekt již neukazují žádné odkazy na objekt, tak jej naplánuje pro úklid z paměti). Tuto situaci znázorňuje obrázek 1.3.



Obrázek 1.3: Rozšířené přiřazení neměnitelného objektu

Druhá delikátnost spočívá v tom, že `a` *operátor*= `b` není úplně totéž jako `a = a` *operátor* `b`. Rozšířená verze vyhledá hodnotu a pouze jednou, je tedy potenciálně rychlejší. Je-li kromě toho `a` komplexní výraz (např. seznam prvků s výpočtem pozice indexu – `items[offset + index]`), pak může být použití rozšířené verze méně náchylné k chybám, protože v případě změny výpočtu není nutné zasahovat do dvou výrazů, ale úpravy stačí provést pouze jednou.

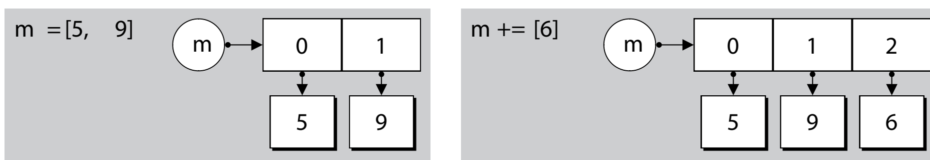
Jazyk Python přetěžuje (tj. opětovně používá pro jiný datový typ) operátory + a += pro řetězce i seznamy, přičemž první znamená zřetězení a druhý připojení pro řetězce a rozšíření (připojení dalšího seznamu) pro seznamy:

```
>>> name = "Jan"
>>> name + "Dudek"
'JanDudek'
>>> name += " Dudek"
>>> name
'Jan Dudek'
```

Podobně jako celá čísla také řetězce jsou neměnitelné, takže při použití operátoru += se vytvoří nový řetězec, s nímž se opětovně sváže odkaz na objekt na levé straně výrazu, tedy úplně stejně jako v případě celých čísel. Seznamy podporují tutéž syntaxi, ovšem v pozadí to probíhá trošku jinak:

```
>>> seeds = ["sezam", "slunečnice"]
>>> seeds += ["dýně"]
>>> seeds
['sezam', 'slunečnice', 'dýně']
```

Seznamy jsou měnitelné, a proto se při použití operátoru += modifikuje původní seznam bez nutnosti opětovného svazování. Tuto situaci zachycuje obrázek 1.4.



Obrázek 1.4: Rozšířené přiřazení měnitelného objektu

Vzhledem k tomu, že syntaxe jazyka Python chytře skrývá odlišnosti mezi měnitelnými a neměnitelnými datovými typy, můžeme se ptát, proč vlastně potřebuje oba druhy? Důvody se povětšinou týkají výkonu. Implementace neměnitelných typů je ve srovnání s měnitelnými typy potenciálně mnohem efektivnější (protože se nikdy nemění). Kromě toho některé datové typy pro kolekce (např. množiny) mohou pracovat pouze s neměnitelnými typy. Na druhou stranu s měnitelnými typy se pohodlněji pracuje. Na tuto odlišnost budeme upozorňovat všude tam, kde má svůj význam (např. v lekci 4 při diskuzi ohledně nastavování výchozích argumentů pro vlastní funkce, v lekci 3 při výkladu seznamů, množin a některých dalších datových typů a opět v lekci 6, kde si ukážeme, jak vytvářet naše vlastní datové typy).

V případě seznamu musí být operand na pravé straně operátoru += iterovatelný, jinak dojde k vyvolání výjimky:

Chyby za běhu programu ➢ 402

```
>>> seeds += 5
Traceback (most recent call last):
...
TypeError: 'int' object is not iterable
```

Seznam lze náležitě rozšířit pomocí iterovatelného objektu, jako je kupříkladu seznam:

```
>>> seeds += [5]
>>> seeds
['sezam', 'slunečnice', 'dýně', 5]
```

Iterovatelný objekt použitý pro rozšíření seznamu může mít samozřejmě více než jeden prvek:

```
>>> seeds += [9, 1, 5, "mák"]
>>> seeds
['sezam', 'slunečnice', 'dýně', 5, 9, 1, 5, 'mák']
```

Připojení obyčejného řetězce (např. `"durian"`) místo seznamu obsahujícího řetězec (`["durian"]`) vede k logickému, i když možná překvapivému výsledku:

```
>>> seeds = ["sezam", "slunečnice", "dýně"]
>>> seeds += "durian"
>>> seeds
['sezam', 'slunečnice', 'dýně', 'd', 'u', 'r', 'i', 'a', 'n']
```

Operátor += rozšířil seznam připojením každého prvku zadaného iterovatelného objektu. Řetězec je iterovatelný, a proto se každý znak zadaného řetězce připojí samostatně. Pokud bychom použili metodu seznamu s názvem `append()`, pak by se argument vždy připojil jako jediný prvek.

Oblast č. 7: Vstup a výstup

Pro psaní opravdu užitečných programů musíme být schopni číst vstup (např. od uživatele z konzoly nebo ze souborů) a vytvářet výstup (ať už do konzoly nebo do souborů). Vestavěnou funkci Pythonu `print()` jsme již používali, její podrobnější popis však odložíme až do lekce 4. V této podčásti se soustředíme na vstup a výstup v rámci konzoly a pro čtení a zápis souborů použijeme přesměrování shellu.

Python nabízí pro přijímání vstupu od uživatele vestavěnou funkci `input()`. Tato funkce přijímá volitelný řetězcový argument (který vypíše na konzolu). Poté počká, až uživatel zadá odpověď, kterou ukončí stiskem klávesy Enter (nebo Return). Pokud uživatel nenapíše žádný text, ale jen stiskne klávesu Enter, funkce `input()` vrátí prázdný řetězec. V opačném případě vrátí řetězec obsahující to, co uživatel zadal, ovšem bez znaků ukončujících řádek.

Zde je náš první úplný „užitečný" prográmek, který staví na řadě již dříve probraných oblastí. Jedinou novinkou je funkce `input()`:

```
print("Zadávejte celá čísla, za každým Enter; nebo jen Enter pro ukončení")

total = 0
count = 0

while True:
    line = input("číslo: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break

if count:
    print("počet =", count, "celkem =", total, "průměr =", total / count)
```

Příklady ke knize ➢ 14

Uvedený program (v příkladech ke knize v souboru `sum1.py`) má pouze 17 prováděných řádků. Takto vypadá jeho typické spuštění:

```
Zadávejte celá čísla, za každým Enter; nebo jen Enter pro ukončení
číslo: 12
číslo: 7
číslo: 1x
invalid literal for int() with base 10: '1x'
číslo: 15
číslo: 5
```

```
číslo:
počet = 4 celkem = 39 průměr = 9.75
```

I když je tento program velice krátký, je poměrně robustní. Pokud uživatel zadá řetězec, který nelze převést na číslo, problém zachytí obsluha výjimky, která vypíše vhodnou zprávu a předá řízení na začátek cyklu. Poslední příkaz `if` zajistí, že pokud uživatel nezadá žádné číslo, souhrnné údaje se nevypíšou, takže nedojde k dělení nulou.

Práci se soubory se budeme plně věnovat v lekci 7. Nyní můžeme vytvářet soubory přesměrováním výstupu funkce `print()` z konzoly:

```
C:\>test.py > results.txt
```

Tento příkaz způsobí, že se výstup obyčejných volání funkce `print()` ve smyšleném programu `test.py` zapíše do souboru `results.txt`. Tato syntaxe funguje v konzole Windows (většinou) a v unixových konzolách. V případě Windows musíme zapsat `C:\Python31\python.exe test.py > results.txt`, pokud je výchozí verzí Python 2 nebo pokud se konzoly týká chyba s asociací souborů. V opačném případě bude za předpokladu, že Python 3 je v proměnné PATH, stačit jen python `test.py > results.txt`, pokud samotné `test.py > results.txt` nefunguje. V případě Unixů musíme změnit soubor na spustitelný soubor (`chmod +x test.py`) a poté jej vyvolat zapsáním `./test.py`. Pokud je adresář, v němž je soubor obsažen, uložen v proměnné PATH, pak pro jeho spuštění stačí zapsat jen `test.py`.

Chyba Windows ohledně asociace souborů ➢ 22

Čtení dat lze realizovat přesměrováním souboru s daty jako vstupu podobným způsobem jako přesměrování výstupu. Pokud bychom však použili přesměrování na soubor `sum1.py`, náš program by selhal. To je dáno tím, že funkce `input()` vyvolá výjimku, pokud obdrží znak EOF (End Of File – konec souboru). Zde je robustnější verze (sum2.py), která přijímá vstup od uživatele píšícího na klávesnici nebo skrze přesměrování souboru:

```
print("Zadávejte celá čísla, za každým Enter; nebo ^D nebo ^Z pro ukončení")

total = 0
count = 0

while True:
    try:
        line = input()
        if line:
            number = int(line)
            total += number
            count += 1
    except ValueError as err:
        print(err)
        continue
    except EOFError:
        break

if count:
    print("počet =", count, "celkem =", total, "průměr =", total / count)
```

Při použití příkazu `sum2.py < data/sum2.dat` (kde soubor `sum2.dat` umístěný v podadresáři `data` obsahuje seznam čísel) obdržíme následující výstup:

```
Zadávejte celá čísla, za každým Enter; nebo ^D nebo ^Z pro ukončení
počet = 37 celkem = 1839 průměr = 49.7027027027
```

Provedli jsme několik malých změn, díky kterým je program mnohem vhodnější pro použití interaktivním způsobem i při přesměrování. Zaprvé jsme změnili ukončení z prázdného řádku na znak EOF (Ctrl+D v Unixu, Ctrl+Z, Enter ve Windows). Díky tomu je náš program robustnější při zpracování vstupních souborů obsahujících prázdné řádky. Dále jsme přestali vypisovat výzvu pro každé číslo, protože v případě přesměrovaného vstupu to nemá žádný význam. A nakonec jsme použili jediný blok `try` se dvěma obsluhami výjimek.

Všimněte si, že pokud je zadáno neplatné číslo (ať už přes klávesnici nebo kvůli nesprávnému řádku dat v přesměrovaném vstupním souboru), vyvolá převod `int()` výjimku `ValueError` a tok programu okamžitě přejde k relevantnímu bloku `except`. To znamená, že při výskytu neplatných dat se nezvýší proměnná `count` ani `total`, což je přesně to, co chceme.

Stejně snadno bychom mohli použít dva samostatné bloky `try`:

```
while True:
    try:
        line = input()
        if line:
            try:
                number = int(line)
            except ValueError as err:
                print(err)
                continue
            total += number
            count += 1
    except EOFError:
        break
```

Ovšem mnohem lepší je seskupit výjimky na konci, aby tak hlavní zpracování zůstalo co nejméně rozházené.

Oblast č. 8: Tvorba a volání funkcí

Není vůbec žádný problém psát programy pomocí datových typů a řídících struktur, které jsme probírali v předchozích oblastech. Nicméně velmi často potřebujeme provádět v podstatě stejné zpracování opakovaně, ale s malou odlišností, jako je odlišná počáteční hodnota. Jazyk Python nabízí prostředky pro zapouzdření sad do podoby funkcí, které lze parametrizovat předávanými argumenty. Zde je obecná syntaxe pro tvorbu funkcí:

```
def názevFunkce(argumenty):
    sada
```

Argumenty jsou volitelné, přičemž více argumentů musí být odděleno čárkou. Každá funkce jazyka Python má nějakou návratovou hodnotu. Ve výchozím stavu se jedná o `None`, pokud ovšem ve funkci nepoužijeme příkaz `return hodnota`, kdy se vrátí zadaná hodnota. Vrácenou hodnotou může být jediná hodnota nebo n-tice hodnot. Volající ji může ignorovat. V takovém případě je prostě zahozena.

Příkaz `return` ➤ 172

Všimněte si, že `def` je příkaz, který funguje podobně jako operátor přiřazení. Při provedení příkazu `def` je vytvořen funkční objekt a dále se vytvoří odkaz na objekt se zadaným názvem, který se nastaví tak, aby ukazoval na nový funkční objekt. Funkce jsou tedy objekty, a proto je lze uchovávat v datových typech pro kolekce a předávat jako argumenty jiným funkcím, k čemuž se ještě dostaneme v pozdějších lekcích.

Jedna z častých potřeb při psaní interaktivní konzolové aplikace spočívá v získání čísla od uživatele. Zde je funkce, která se o to postará:

```
def get_int(msg):
    while True:
        try:
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)
```

Tato funkce přijímá jeden argument `msg`. Uvnitř cyklu `while` je uživatel vyzván k zadání čísla. Pokud zadá něco neplatného, vyvolá se výjimka `ValueError`, vypíše se chybová zpráva a cyklus se opakuje. Jakmile dojde k zadání platného čísla, vrátíme jej volajícímu. Zde je příklad volání naší funkce:

```
age = get_int("zadej svůj věk: ")
```

V tomto příkladu je jediný argument povinný, protože jsme neuvedli jeho výchozí hodnotu. Ve skutečnosti jazyk Python nabízí pro parametry funkcí velice sofistikovanou a flexibilní syntaxi, která podporuje výchozí hodnoty argumentů a argumenty definované dle pozice nebo klíčového slova. Celou tuto syntaxi si podrobně rozebereme v lekci 4.

I když vytváření vlastních funkcí může být velmi uspokojující, v řadě případů není nezbytně nutné. To je dáno tím, že jazyk Python má spoustu vestavěných funkcí a mnohonásobně více funkcí v modulech své standardní knihovny, takže to, co potřebujeme, je již s největší pravděpodobností k dispozici.

Modul Pythonu je obyčejný soubor `.py`, který obsahuje kód jazyka Python, jako jsou definice vlastních funkcí a tříd (vlastních datových typů) a někdy též proměnných. Pro přístup k funkčnosti v určitém modulu je nutné jej nejdříve importovat:

```
import sys
```

K importování modulu se používá příkaz import, za nímž následuje název soubor `.py` bez přípony.*

* Modul sys, některé další vestavěné moduly a moduly implementované v jazyku C nemusejí mít nutně odpovídající soubor .py. Používají se však naprosto stejně jako ty, které jej mají.

Jakmile je modul importován, můžeme přistupovat k libovolným funkcím, třídám nebo proměnným, které obsahuje:

```
print(sys.argv)
```

Modul `sys` poskytuje proměnnou `argv` obsahující seznam, jehož první prvek je název, pod kterým byl daný program spuštěn, a jehož druhý prvek a všechny následující představují argumenty předané programu z příkazového řádku. Dva výše uvedené řádky tvoří dohromady celý program `echoargs.py`. Pokud program spustíme na příkazovém řádku jako `echoargs.py -v`, vypíše na konzoli `['echoargs.py', '-v']` (v Unixu může být první záznam `'./echoargs.py'`).

Operátor tečka (.) ➤ 30

Syntaxe pro použití funkce z nějakého modulu má obecný tvar `názevModulu.názevFunkce(argumenty)`. Využíváme zde operátor tečka („přístup k atributu"), s nímž jsme se seznámili v Oblasti č. 3. Standardní knihovna obsahuje spoustu modulů a my budeme v této knize používat velkou část z nich. Názvy všech standardních modulů mají malá písmena, takže někteří programátoři používají pro odlišení svých vlastních modulů názvy, jejichž slova začínají velkými písmeny (např. `MyModule`).

Podívejme se nyní na jeden příklad využívající modul `random` (umístěný v souboru `random.py` standardní knihovny), který nabízí řadu užitečných funkcí:

```
import random
x = random.randint(1, 6)
y = random.choice(["jablko", "banán", "hruška", "švestka"])
```

Po provedení těchto příkazů bude proměnná x obsahovat číslo mezi 1 a 6 včetně a proměnná y bude obsahovat jeden z řetězců ze seznamu, který jsme předali funkci `random.choice()`.

Řádek shebang (#!) ➤ 22

Všechny příkazy import se standardně umisťují na začátek souborů `.py` za řádek shebang a za dokumentaci k modulu (na dokumentování modulů se podíváme v lekci 5). Doporučujeme nejdříve importovat moduly standardní knihovny, poté moduly třetích stran a nakonec své vlastní moduly.

Příklady

V předchozí části jsme se o jazyku Python naučili dost na to, abychom mohli vytvářet skutečné programy. V této části prostudujeme dva hotové programy, které využívají pouze ty části jazyka Python, které jsme již probrali. Na těchto programech si ukážeme, co je možné vytvořit, a také si na nich upevníme dosud získané znalosti.

V následujících lekcích se budeme postupně zabývat dalšími oblastmi jazyka Python a jeho knihovny, abychom pak byli schopni psát programy, které budou stručnější a robustnější ve srovnání s těmi, které si ukážeme v této části. Nejdříve ale musíme položit základy, na nichž pak budeme stavět.

Program bigdigits.py

První program, na který se podíváme, je docela krátký, i když má několik zajímavých aspektů, mezi něž patří seznam seznamů. Program funguje tak, že na příkazovém řádku zadáme číslo a program jej vypíše do konzoly pomocí „velkých" číslic.

Na serverech se spoustou uživatelů, kteří sdílejí vysokorychlostní řádkovou tiskárnu, to běžně probíhalo tak, že před tiskovou úlohou každého uživatele se pomocí této techniky vytiskl úvodní list, který obsahoval jeho uživatelské jméno a nějaké další identifikační údaje.

Kód programu prostudujeme ve třech částech: příkaz `import`, tvorba seznamů uchovávajících data a samotné zpracování. Podívejme se nejdříve na jeho ukázkové spuštění:

```
bigdigits.py 41072819
   *     *     ***    *****   ***    ***    *    ****
  **    **    *   *       *  *   *  *   *  **   *   *
 * *     *   *     *     *   *  *   *   *   *   *   *
*  *     *   *     *    *      *     ***     *    ****
******   *   *     *   *      *     *   *   *       *
   *     *    *   *   *      *      *   *   *       *
   *    ***    ***    *      *****   ***   ***      *
```

Výzvu konzoly (nebo úvodní znaky `./` pro uživatele Unixu) jsme si zde neukázali. Od této chvíle je budeme považovat za samozřejmou součást.

```
import sys
```

Vzhledem k tomu, že argument (číslo, které se má vypsat) musíme načíst z příkazového řádku, potřebujeme přístup k seznamu `sys.argv`. Z tohoto důvodu začínáme importem modulu `sys`.

Každé číslo reprezentujeme jako seznam řetězců. Například nula vypadá takto:

```
Zero = ["  ***  ",
        " *   * ",
        "*     *",
        "*     *",
        "*     *",
        " *   * ",
        "  ***  "]
```

Všimněte si, že seznam řetězců `Zero` je rozložen na několik řádků. Příkazy jazyka Python obvykle zabírají jediný řádek, lze je však rozložit na více řádků, jedná-li se o uzávorkovaný výraz, literál seznamu, množiny či slovníku, seznam argumentů při volání funkce nebo víceřádkový příkaz, v němž je každý znak konce řádku vyjma posledního potlačen předsazením zpětného lomítka (`\`). Ve všech těchto případech lze použít libovolný počet řádků, přičemž u druhého a následujících řádků již nezáleží na odsazení.

Typ `set` ➤ 123

Typ `dict` ➤ 128

Každý seznam reprezentující určité číslo má sedm řetězců jednotné délky, která však může být u každého čísla jiná. Seznamy pro ostatní čísla fungují stejným způsobem jako pro nulu, ovšem kvůli kompaktnosti jsou uvedena na jediném řádku:

```
One = [" * ", "** ", " * ", " * ", " * ", " * ", "***"]
Two = [" *** ", "*   *", "*  * ", "  *  ", " *   ", "*    ", "*****"]
# ...
Nine = [" ****", "*   *", "*   *", " ****", "    *", "    *", "    *"]
```

Posledním kouskem dat, která potřebujeme, je seznam seznamů všech čísel:

```
Digits = [Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine]
```

Seznam `Digits` bychom mohli vytvořit také přímo a vyhnout se tak tvorbě dalších proměnných:

```
Digits = [
    [" *** ", " * * ", "* *", "* *", "* *", " * * ", " *** "], # Nula
    [" * ", "** ", " * ", " * ", " * ", " * ", "***"], # Jedna
    # ...
    [" ****", "* *", "* *", " ****", " *", " *", " *"] # Devět
]
```

Raději jsme však pro každé číslo použili samostatnou proměnnou – jednak kvůli snazšímu pochopení, a také kvůli tomu, že kód pak působí čistším dojmem.

Nyní uvedeme celou zbývající část kódu, takže si ještě před tím, než začnete číst vysvětlení, můžete vyzkoušet, zda přijdete na to, jak vlastně funguje:

```
try:
    digits = sys.argv[1]
    row = 0
    while row < 7:
        line = ""
        column = 0
        while column < len(digits):
            number = int(digits[column])
            digit = Digits[number]
            line += digit[row] + " "
            column += 1
        print(line)
        row += 1
except IndexError:
    print("použití: bigdigits.py <číslo>")
except ValueError as err:
    print(err, "v", digits)
```

Celý kód je zabalen do obsluhy výjimky, která zachytí dvě chybové situace. Začínáme načtením argumentu z příkazového řádku programu. Seznam `sys.argv` začíná jako všechny seznamy jazyka Python od nuly. Prvek na pozici 0 je název, pod kterým byl program spuštěn, takže v běžícím programu má tento seznam vždy alespoň jeden prvek. Pokud nebyl zadán žádný argument, pak budeme zkoušet přistupovat ke druhému prvku jednoprvkového seznamu, což způsobí vyvolání výjimky `IndexError`. V takovém případě se řízení předá odpovídajícímu bloku obsluhy výjimky, kde jednoduše vypíšeme, jak se program používá. Provádění pak pokračuje za koncem bloku `try`, kde však již není žádný kód, a proto program skončí.

range() ➤ 141

Pokud k výjimce `IndexError` nedojde, uchovává řetězec `digits` argument příkazového řádku, o němž se domníváme, že obsahuje posloupnost číselných znaků. (Vzpomeňte si z Oblasti č. 2, že u identifikátorů se rozlišuje velikost písmen, takže `digits` a `Digits` jsou dvě různé proměnné.) Každá velká číslice je reprezentována sedmi řetězci, takže pro správný výstup musíme nejdříve vypsat horní řádek každé číslice, pak další řádek a tak pořád dál, dokud se nevypíše všech sedm řádků. K průchodu přes všechny řádky používáme cyklus `while`. Stejně by fungoval i cyklus `for` (`for row in (0, 1, 2, 3, 4, 5, 6):`). Později se seznámíme s mnohem lepším způsobem využívajícím vestavěnou funkci `range()`.

Řetězec `line` používáme pro uložení řetězců daného řádku ze všech zpracovávaných číslic. Poté procházíme jednotlivé sloupce, což znamená jednotlivé znaky v argumentu příkazového řádku. Každý znak získáváme pomocí výrazu `digits[column]` a převádíme jej na celé číslo s názvem `number`. Pokud převod selže, vyvolá se výjimka `ValueError` a řízení se okamžitě předá odpovídající obsluze výjimky. V tomto případě vypíše chybovou zprávu a program pokračuje za blokem `try`. Zde ale, jak jsme si řekli již dříve, není již žádný kód, a proto náš program jednoduše skončí.

Pokud převod uspěje, použijeme proměnnou `number` jako index do seznamu `Digits`, z něhož extrahujeme seznam řetězců příslušné číslice. Poté z tohoto seznamu přidáme řetězec na pozici `row` do proměnné `line`, k níž přidáme také mezeru pro vodorovné odsazení každé číslice.

Při každém ukončení vnitřního cyklu `while` vypíšeme obsah sestavený v proměnné `line`. Klíčem k pochopení tohoto programu je místo, kde připojujeme řetězec s řádkem každé číslice k proměnné `line` reprezentující aktuální řádek. Vyzkoušejte si spustit program, abyste se přesvědčili, jak pracuje. Vrátíme se k němu ve cvičení, kde malinko vylepšíme jeho výstup.

Program generate_grid.py

Jedna z častých potřeb je generování testovacích dat. Neexistuje žádný generický program, který by toto prováděl, poněvadž testovací data se značně liší. Python se často používá k vytváření testovacích dat, protože jeho programy se velice snadno píšou a upravují. V této podčásti vytvoříme program, který generuje tabulku náhodných čísel. Uživatel může stanovit, kolik má mít řádků a sloupců a v jakém rozmezí se mají generovaná čísla nacházet. Začneme pohledem na ukázkové spuštění:

```
generate_grid.py
řádků: 4x
invalid literal for int() with base 10: '4x'
řádků: 4
sloupců: 7
minimum (nebo Enter pro 0): -100
maximum (nebo Enter pro 1000):
554 720 550 217 810 649 912
-24 908 742 -65 -74 724 825
711 968 824 505 741 55 723
180 -60 794 173 487 4 -35
```

Program pracuje interaktivně a na začátku jsme udělali chybu při zadávání počtu řádků. Program reagoval vypsáním chybové zprávy a poté nás znovu požádal o zadání počtu řádků. Pro maximum jsme jen stiskli klávesu Enter, čímž jsme přijali výchozí hodnotu.

Kód si prohlédneme ve čtyřech částech: příkaz `import`, definice funkce `get_int()` (sofistikovanější varianta původní verze definované v Oblasti č. 8), interakce s uživatelem pro získání hodnot a samotné zpracování.

```
import random
```

random.randint() ➤ 46

Pro přístup k funkci `random.randint()` potřebujeme modul `random`.

```
def get_int(msg, minimum, default):
    while True:
        try:
            line = input(msg)
            if not line and default is not None:
                return default
            i = int(line)
            if i < minimum:
                print("musí být >=", minimum)
            else:
                return i
        except ValueError as err:
            print(err)
```

Tato funkce vyžaduje tři argumenty: řetězec se zprávou, minimální hodnotu a výchozí hodnotu. Pokud uživatel stiskne jen klávesu Enter, pak nastávají dvě možnosti. Má-li parametr default hodnotu `None`, což znamená, že výchozí hodnota nebyla zadána, předá se řízení programu až na řádek s převodem `int()`, který selže (protože `''` nelze převést na číslo) a vyvolá výjimku `ValueError`. Pokud ale parametr default není `None`, vrátíme jeho hodnotu. V opačném případě se funkce pokusí převést text zadaný uživatelem na celé číslo a v případě úspěchu zkontroluje, zda je toto číslo alespoň zadané minimum.

Funkce tedy vždy vrátí buď výchozí hodnotu (pokud uživatel jen stiskl Enter), nebo platné celé číslo, které je větší nebo rovno než stanovené minimum.

```
rows = get_int("řádků: ", 1, None)
columns = get_int("sloupců: ", 1, None)
minimum = get_int("minimum (nebo Enter pro 0): ", -1000000, 0)

default = 1000
if default < minimum:
    default = 2 * minimum
maximum = get_int("maximum (nebo Enter pro " + str(default) + "): ",
                  minimum, default)
```

Naše funkce `get_int()` usnadňuje získávání počtu řádků a sloupců a minimální požadované hodnoty náhodného čísla. Pro řádky a sloupce nastavujeme výchozí hodnotu na `None`, což znamená žádná výchozí hodnota, takže uživatel musí zadat nějaké číslo. V případě minima používáme výchozí hodnotu 0 a pro maximum máme výchozí hodnotu 1000 nebo dvojnásobek minima v případě, že je minimum větší nebo rovno 1000.

Jak jsme si řekli již u předchozího příkladu, seznam argumentů při volání funkce se může rozkládat na více řádcích, přičemž na druhém a na všech následujících řádcích je odsazení bezvýznamné.

Jakmile víme, kolik řádků a sloupců uživatel požaduje a jaké jsou hodnoty pro minima a maxima pro náhodná čísla, můžeme přejít k vlastnímu zpracování.

```
row = 0
while row < rows:
    line = ""
    column = 0
    while column < columns:
        i = random.randint(minimum, maximum)
        s = str(i)
        while len(s) < 10:
            s = " " + s
        line += s
        column += 1
    print(line)
    row += 1
```

K vygenerování tabulky používáme tři cykly `while`. Vnější pracuje s řádky, prostřední se sloupci a vnitřní s jednotlivými znaky. V prostředním cyklu získáváme náhodné číslo ve stanoveném rozsahu, které poté převedeme na řetězec. Vnitřní cyklus `while` používáme k vyplnění řetězce úvodními mezerami, aby tak každé číslo reprezentoval řetězec dlouhý 10 znaků. Pro nashromáždění čísel na každém řádku používáme řetězec `line`, který vypisujeme vždy po přidání čísel každého sloupce. Tím jsme se dostali na konec našeho druhého příkladu.

Python nabízí velmi sofistikované funkce pro formátování řetězců a také skvělou podporu pro cykly `for ... in`, takže v realističtější verzi obou programů `bigdigits.py` a `generate_grid.py` bychom použili cykly `for ... in` a v programu `generate_ grid.py` bychom místo hrubě vyplňovaných mezer použili funkce Pythonu pro formátování řetězce. Omezili jsme se však na osm oblastí jazyka Python, s kterými jsme se seznámili v této lekci a které jsou dostatečné pro psaní ucelených a užitečných programů. V každé z následujících lekci si osvojíme nové prvky jazyka Python, takže při postupu touto knihou budou naše programy a dovednosti stále sofistikovanější.

str.format() ➤ 83

Shrnutí

V této lekci jsme se naučili, jak upravovat a spouštět programy napsané v jazyku Python, a prostudovali jsme několik malých, ale ucelených programů. Většina stránek této lekce byla věnována osmi oblastem „nádherného srdce" jazyka Python, jejichž osvojení stačí pro psaní skutečných programů.

Začali jsme dvěma nejzákladnějšími datovými typy jazyka Python: `int` (celé číslo) a `str` (řetězec). Celočíselné literály se zapisují stejně jako ve většině ostatních programovacích jazyků. Řetězcové literály se zapisují pomocí jednoduchých nebo dvojitých uvozovek. Nezáleží na tom, které použijeme, podstatné je, aby byl na obou koncích stejný druh uvozovek. Řetězce a celá čísla můžeme navzájem převádět (např. `int("250")` a `str(125)`). Pokud převod na celé číslo selže, vyvolá se výjimka `ValueError`. Na druhou stranu na řetězec lze převést téměř cokoliv.

Řetězce jsou posloupnosti, takže funkce a operace, které lze použít u posloupností, lze použít také pro řetězce. Můžeme tak například pomocí operátoru pro přístup k prvku (`[]`) přistoupit k určitému znaku, pomocí operátoru + spojit řetězce a pomocí operátoru += připojit jeden řetězec k druhému. Řetězce jsou neměnitelné, a proto se při připojování vytvoří v pozadí nový řetězec, který vznikne spojením daných řetězců, a s výsledným řetězcem se opětovně sváže odkaz na objekt řetězce na levé straně operátoru. Pomocí cyklu `for ... in` můžeme procházet jednotlivé znaky řetězce a pomocí vestavěné funkce `len()` můžeme zjistit počet znaků v řetězci.

U neměnitelných objektů, jako jsou řetězce, celá čísla a n-tice, můžeme psát svůj kód tak, jako by odkaz na objekt byl proměnná, což znamená, jako by odkaz na objekt byl samotný objekt, na který ukazuje. Totéž můžeme dělat i v případě měnitelných objektů, i když jakákoliv změna provedená na měnitelném objektu ovlivní všechny výskyty daného objektu (tj. všechny odkazy na daný objekt). Tomuto tématu se budeme podrobněji věnovat v lekci 3.

Jazyk Python nabízí několik vestavěných datových typů pro kolekce a několik dalších je k dispozici v jeho standardní knihovně. Seznámili jsme se s typy `list` (seznam) a `tuple` (n-tice) a s tím, jak se vytvářejí n-tice a seznamy z literálů (např. `even = [2, 4, 6, 8]`). Seznamy, podobně jako vše ostatní v jazyku Python, jsou objekty, takže na nich můžeme volat metody. Například `even.append(10)` přidá další prvek do seznamu. Seznamy a n-tice jsou stejně jako řetězce posloupnosti, a proto je můžeme pomocí cyklu `for ... in` procházet po jednotlivých prvcích a pomocí funkce `len()` zjistit, kolik prvků obsahují. Pomocí operátoru pro přístup k prvku (`[]`) můžeme též získat určitý prvek ze seznamu nebo n-tice, pomocí operátoru + dva seznamy nebo n-tice spojíme a pomocí operátoru += připojíme jeden k druhému. Pokud bychom chtěli připojit jediný prvek k seznamu, pak musíme použít buď metodu `list.append()`, nebo operátor += s prvkem přetvořeným do jednoprvkového seznamu (např. `even += [12]`). Seznamy jsou měnitelné, a proto můžeme použít operátor [] ke změně jednotlivých prvků (např. `even[1] = 16`).

Rychlé operátory `is` a `is not` lze použít pro kontrolu, zda dva odkazy na objekty ukazují na tentýž objekt, což je zvláště užitečné při kontrole proti unikátnímu vestavěnému objektu `None`. K dispozici jsou všechny obvyklé porovnávací operátory (<, <=, ==, !=, >=, >), lze je však použít pouze s kompatibilními datovými typy, které navíc musejí danou operaci podporovat. Všechny datové typy, s nimiž jsme se dosud seznámili (`int`, `str`, `list` a `tuple`), podporují celou skupinu porovnávacích operátorů. Při porovnávání nekompatibilních typů (např. typ `int` s typem `str` nebo `list`) dojde logicky k výjimce `TypeError`.

Jazyk Python podporuje standardní logické operátory `and`, `or` a `not`. Operátory `and` a `or` používají logiku zkráceného vyhodnocování, takže vracejí operand, který rozhoduje o výsledku, což nemusí nutně být logická hodnota (i když jej lze převést na logickou hodnotu). To znamená, že ne vždy obdržíme buď `True`, nebo `False`.

Příslušnost k typům pro posloupnosti včetně řetězců, seznamů a n-tic můžeme testovat pomocí operátorů `in` a `not in`. Při testování příslušnosti se u seznamů a n-tic používá pomalé lineární vyhledávání a u řetězců potenciálně mnohem rychlejší hybridní algoritmus, avšak výkon je jen málokdy problém, tedy až na velmi dlouhé řetězce, seznamy a n-tice. V lekci 3 se seznámíme s datovými typy jazyka Python pro asociativní pole a množinové kolekce, které realizují velmi rychlé testování příslušnosti. Pomocí funkce `type()` je možné zjistit typ objektu proměnné (tj. typ objektu, na který ukazuje daný odkaz na objekt). Tato funkce se však většinou používá pouze pro ladění a testování.

Jazyk Python nabízí několik řídících struktur, mezi něž patří podmíněné větvení `if ... elif ... else`, podmíněný cyklus `while`, cyklus přes posloupnosti `for ... in` a bloky pro ošetřování výjimek `try ... except`. Cykly `while` i `for ... in` lze předčasně ukončit pomoc příkazu `break` a pomocí příkazu `continue` lze též předat řízení zpět na začátek cyklu.

Podporovány jsou obvyklé aritmetické operace, včetně `+`, `-`, `*` a `/`, ačkoliv Python je výjimečný v tom, že operátor / vždy vrací desetinné číslo, a to i tehdy, jsou-li operandy celočíselné. (Ořezávané dělení používané v řadě ostatních jazyků je v Pythonu k dispozici jako operátor `//`.) Python nabízí také operátory rozšířeného přiřazení, jako je += a *=. Tyto operátory vytvářejí v pozadí nové objekty a provádějí opětovné svázání, je-li jejich levý operand neměnitelný. Aritmetické operace jsou přetížené také pro typy `str` a `list`.

Vstup a výstup na konzolu lze realizovat pomocí funkcí `input()` a `print()`, přičemž prostřednictvím přesměrování souboru v konzole můžeme tytéž vestavěné funkce použít i pro čtení a zapisování souborů.

Kromě bohaté palety vestavěných funkcí nabízí Python také svoji rozsáhlou standardní knihovnu s moduly přístupnými po jejich importu příkazem `import`. Jedním z často importovaných modulů je modul `sys`, který uchovává seznam `sys.argv` s argumenty příkazového řádku. Pokud Python nějakou funkci, kterou potřebujeme, nemá, můžeme si ji pomocí příkazu `def` snadno vytvořit.

S využitím funkčních prvků popsaných v této lekci je možné psát v jazyku Python krátké, ale užitečné programy. V následující lekci se dozvíme více o datových typech jazyka Python, podrobněji se podíváme na typy `int` a `str` a představíme si několik zcela nových datových typů. V lekci 3 se naučíme více o n-ticích a seznamech a také o některých z dalších datových typů jazyka Python určených pro kolekce. Pak se v lekci 4 budeme detailněji věnovat řídícím strukturám jazyka Python a naučíme se, jak vytvářet své vlastní funkce tak, abychom funkčnost správně zabalili, vyhnuli se duplicitnímu kódu a podporovali jeho znovupoužití.

Cvičení

Smyslem cvičení nejen v této lekci, ale v celé knize je přimět vás experimentovat s Pythonem, abyste si v praxi osvojili látku probíranou v dané lekci. V příkladech a cvičeních se budeme věnovat zpracování číselných i textových údajů, přičemž jednotlivé příklady budou co nejmenší, abyste se mohli soustředit na uvažování a učení spíše než na psaní kódu. Ke každému cvičení existuje řešení, které najdete v příkladech ke knize.

Příklady ke knize ➢ 14

1. Jedna pěkná varianta programu `bigdigits.py` může vypadat tak, že místo vypsání hvězdiček (`*`) vypíše odpovídající číslici:

```
bigdigits_ans.py 719428306
77777   1    9999     4     222    888    333     000     666
    7  11   9   9    44    2   2  8   8  3   3   0   0   6
   7    1   9   9   4 4    2  2   8   8      3  0     0  6
  7     1    9999  4  4      2     888     33    0     0  6666
 7      1       9  444444   2     8   8      3  0     0  6   6
7       1       9     4    2      8   8  3   3   0   0   6   6
7      111      9     4    22222  888    333     000     666
```

Lze použít dva přístupy. Nejjednodušší je prostě změnit hvězdičky v seznamech. To však není příliš flexibilní, a proto byste takto neměli postupovat. Místo toho změňte kód provádějící zpracování tak, aby se místo jednorázového přidání řetězce s řádkem každé číslice přidávaly jednotlivé znaky postupně a při každém výskytu hvězdičky se použila příslušná číslice.

To lze provést zkopírováním souboru `bigdigits.py` a změnou asi pěti řádků. Není to těžké, ale malinko zákeřné. Hotové řešení najdete v souboru `bigdigits_ans.py`.

2. Editor IDLE lze použít jako velice výkonnou a flexibilní kalkulačku, někdy je ale užitečné mít kalkulačku specifickou pro určitý úkol. Vytvořte program, který vyzve uživatele k zadání čísla v cyklu `while`, v němž bude postupně sestavovat seznam zadaných čísel. Jakmile uživatel dokončí zadávání (prostým stiskem klávesy Enter), vypíšou se zadaná čísla, jejich počet, součet, nejmenší a největší čísla a jejich průměr (součet / počet). Zde je ukázkový běh programu:

```
average1_ans.py
zadejte číslo nebo Enter pro ukončení: 5
zadejte číslo nebo Enter pro ukončení: 4
zadejte číslo nebo Enter pro ukončení: 1
zadejte číslo nebo Enter pro ukončení: 8
zadejte číslo nebo Enter pro ukončení: 5
zadejte číslo nebo Enter pro ukončení: 2
zadejte číslo nebo Enter pro ukončení:
čísla: [5, 4, 1, 8, 5, 2]
počet = 6 součet = 25 nejmenší = 1 největší = 8 průměr = 4.16666666667
```

Inicializace proměnných (prázdného seznamu je prostě `[]`) zabere přibližně čtyři řádky a cyklus `while` včetně základního ošetření chyb méně než 15 řádků. Výpis na konci lze provést pomocí několika málo řádků, takže celý program včetně prázdných řádků kvůli čitelnosti by měl zabírat kolem 25 řádků.

3. V některých situacích potřebujeme vygenerovat testovací text – například pro naplnění návrhu webové stránky před tím, než bude dostupný skutečný obsah, nebo pro testování obsahu při vývoji generátoru sestav. Za tímto účelem napište program, který generuje děsivé básničky (takový ten typ, při kterém by se zastyděl i Vogon).

Vytvořte nějaké seznamy slov – například zájmena („můj", „tvůj", „její", „jeho"), podstatná jména („kočka", „pes", „muž", „žena"), slovesa („zpívá", „běží", „skáče") a příslovce („hlasitě", „tiše", „kvalitně", „hrozně"). Poté proveďte pět cyklů a v každé iteraci použijte funkci `random.choice()` pro

`random.randint()` a `random.choice()` ➤ 46

výběr zájmena, podstatného jména, příslovce a slovesa nebo jen zájmena, podstatného jména a slovesa a výslednou větu vypište. Zde je ukázkový běh programu:

```
awfulpoetry1_ans.py
její muž kvalitně říka
můj pes cítí
jeho holka tiše hopká
tvůj holka cítí
její pes plave
```

V tomto programu musíte importovat modul `random`. Seznam lze napsat na 4–10 řádků v závislosti na tom, kolik slov do něj chcete vložit, přičemž samotný cyklus vyžaduje méně než deset řádků, takže s nějakými prázdnými řádky by měl mít celý program přibližně 20 řádků kódu. Řešení najdete v souboru `awfulpoetry1_ans.py`.

4. Aby byl program s hroznou poezií všestrannější, přidejte do něj takový kód, aby v případě, že uživatel zadá na příkazovém řádku nějaké číslo (mezi 1 a 10 včetně), program vypsal zadaný počet řádků. Pokud žádný argument na příkazovém řádku není zadán, vypíše se 5 řádků jako dříve. Musíte změnit hlavní cyklus (např. na cyklus `while`). Mějte na paměti, že porovnávací operátory jazyka Python lze řetězit, není tedy nutné při kontrole rozsahu argumentu používat logický operátor `and`. Dodatečnou funkčnost lze zajistit přidáním přibližně deseti řádků kódu. Řešení najdete v souboru `awfulpoetry2_ans.py`.

5. Bylo by pěkné mít možnost vypočítat medián (střední hodnotu) a také střed pro průměry ze cvičení 2, k tomu je ale nutné seznam seřadit. V Pythonu lze seznam snadno seřadit pomocí metody `list.sort()`, o které jsme se zatím nezmínili, takže ji zde nepoužijeme. Rozšiřte program `average1_ans.py` o blok kódu, který seřadí seznam čísel. Efektivita nás nezajímá, použijte ten nejjednodušší postup, který vás napadne. Jakmile je seznam seřazen, je mediánem prostřední hodnota, má-li seznam lichý počet prvků, nebo průměr dvou prostředních hodnot, má-li seznam sudý počet prvků. Vypočtěte medián a společně s ostatními informacemi jej vypište.

Tohle je malinko složitější, zvláště pro nezkušené programátory. Může to být těžší i pro ty, kteří s Pythonem již nějaké ty zkušenosti mají, alespoň tedy v případě, že se budete držet stanoveného omezení a použijete pouze ty části Pythonu, které jsme již probírali. Řazení lze provést asi na desítce řádků a výpočet mediánu (v němž nemůžeme použít operátor modulo, protože jsme jej ještě neprobírali) nezabere víc než čtyři řádky. Řešení najdete v souboru `average2_ans.py`.

LEKCE 2

Datové typy

V této lekci:

- Identifikátory a klíčová slova
- Celočíselné typy
- Typy s pohyblivou řádovou čárkou
- Řetězce

V této lekci se na jazyk Python podíváme mnohem detailněji. Začneme diskuzí o pravidlech pojmenovávání odkazů na objekty a ukážeme si seznam klíčových slov jazyka Python. Poté se podíváme na všechny důležité datové typy jazyka Python vyjma datových typů představujících kolekce, kterým se budeme věnovat v lekci 3. Všechny zde probírané datové typy jsou vestavěné kromě jednoho, který pochází ze standardní knihovny. Jediný rozdíl mezi vestavěnými a knihovními datovými typy spočívá v tom, že v druhém případě musíme nejdříve importovat příslušný modul a název datového typu kvalifikovat názvem modulu, z něhož pochází (více viz Lekce 5).

Identifikátory a klíčová slova

Odkazy na objekty ➢ 26

Při vytváření datového prvku jej můžeme buď přiřadit proměnné, nebo vložit do nějaké kolekce. (Jak jsme si řekli již v předchozí lekci, při přiřazování se v Pythonu ve skutečnosti děje to, že se sváže odkaz na objekt tak, aby ukazoval na objekt v paměti uchovávající daná data.) Název, který dáme našemu odkazu na objekt, se nazývá *identifikátor* nebo prostě *jméno*.

Platným identifikátorem v jazyku Python je neprázdná posloupnost znaků libovolné délky, která se skládá z „počátečního znaku“ a nulového nebo většího počtu „pokračovacích znaků“. Takovýto identifikátor musí splňovat několik pravidel a dodržovat následující konvence. První pravidlo se týká počátečního a pokračovacích znaků. Počátečním znakem může být cokoliv, co znaková sada Unicode považuje za písmeno, tedy všechny písmena ASCII („a“, „b“, ..., „z“, „A“, „B“, ..., „Z“), podtržítko („_“) a písmena z většiny neanglických jazyků. Každý pokračovací znak může být libovolným znakem, který je povolen pro počáteční znak, nebo téměř jakýkoliv znak různý od bílého místa, což mimo jiné znamená všechny znaky, které znaková sada Unicode považuje za číslice, jako je („0“, „1“, ..., „9“), nebo český znak „ř“. U identifikátorů se rozlišuje velikost písmen, takže například `TAXRATE`, `Taxrate`, `TaxRate`, `taxRate` a `taxrate` je pět odlišných identifikátorů.

Přesná sada znaků, které jsou povolené pro počáteční a pokračovací znaky, je popsána v dokumentaci (viz *http://docs.python.org/reference/lexical_analysis.html#identifiers*) a v návrhu PEP 3131 (viz *http://www.python.org/dev/peps/pep-3131/*).*

Druhé pravidlo říká, že žádný identifikátor nemůže mít stejný název jako některé klíčové slovo jazyka Python, takže nemůžeme použít žádné ze jmen uvedených v tabulce 2.1.

Tabulka 2.1: Klíčová slova jazyka Python

and	continue	except	global	lambda	pass	while
as	def	False	if	None	raise	with
assert	del	finally	import	nonlocal	return	yield
break	elif	for	in	not	True	
class	else	from	is	or	try	

S většinou z těchto klíčových slov jsme se již setkali v předchozí lekci, i když 11 z nich (`assert`, `class`, `del`, `finally`, `from`, `global`, `lambda`, `nonlocal`, `raise`, `with` a `yield`) budeme ještě probírat.

* Zkratka PEP znamená Python Enhancement Proposal (návrh na rozšíření Pythonu). Pokud chce někdo Python změnit nebo rozšířit, pak může za předpokladu, že má dostatečnou podporu komunity Pythonu, předložit návrh PEP s podrobnostmi o navrhovaných změnách, aby jej bylo možné formálně posoudit a v některých případech, jako je třeba právě PEP 3131, přijmout a implementovat. Všechny návrhy PEP jsou přístupné ze stránky *www.python.org/dev/peps/*.

První konvence zní takto: Pro své vlastní identifikátory nepoužívejte název žádného z předdefinovaných identifikátorů Pythonu. To tedy znamená, že byste neměli používat názvy `NotImplemented` a `Ellipsis`, žádný z názvů vestavěných datových typů (jako je `int`, `float`, `list`, `str` a `tuple`), funkcí nebo výjimek jazyka Python. Jak ale zjistíme, zda daný identifikátor spadá do některé z těchto kategorií? Python nabízí vestavěnou funkci `dir()`, která vrací seznam atributů zadaného objektu. Pokud ji zavoláme bez argumentů, vrátí seznam vestavěných atributů Pythonu:

```
>>> dir() # senam Pythonu 3.1 má navíc prvek '__package__'
['__builtins__', '__doc__', '__name__']
```

Atribut `__builtins__` je ve skutečnosti modul, který uchovává všechny vestavěné atributy Pythonu. Můžeme jej tedy použít jako argument funkce `dir()`:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

V tomto seznamu je přibližně 130 jmen, takže většinu z nich jsme vypustili. Jména začínající velkým písmenem jsou názvy vestavěných výjimek Pythonu. Ostatní jsou názvy funkcí a datových typů.

Druhá konvence se týká použití podtržítek (_). Neměly by se používat názvy, které začínají a končí podtržítky (např. `__lt__`). Python definuje rozličné speciální metody a proměnné s těmito názvy (takovéto speciální metody můžeme opětovně implementovat, to znamená vytvořit si jejich vlastní verze), sami bychom však nové názvy tohoto typu zavádět neměli. Více se těmto typům názvů budeme věnovat v lekci 6. S názvy, které začínají jedním nebo dvěma podtržítky (a které nekončí dvěma podtržítky), se v určitých kontextech zachází zvláštním způsobem. Příklad použití názvů s jedním úvodním podtržítkem si ukážeme v lekci 5 a na praktické využití těch, které obsahují dvě úvodní podtržítka, se podíváme v lekci 6.

Jediné osamocené podtržítko lze použít jako identifikátor, přičemž v interaktivním interpretu nebo v okně Python Shell uchovává _ výsledek posledního výrazu, který byl vyhodnocen. V normálním běžícím programu žádná proměnná _ neexistuje, pokud ji sami explicitně nepoužijeme. Někteří programátoři rádi používají _ v cyklech `for ... in`, když se nezajímají o procházené prvky:

```
for _ in (0, 1, 2, 3, 4, 5):
    print("Ahoj")
```

Mějte však na paměti, že při psaní internacionalizovaných programů se _ používá často jako název překladové funkce. Důvodem je to, že místo `gettext.gettext("Přelož mne")` stačí napsat jen `_("Přelož mne")`. (Abychom mohli přistupovat k funkci `gettext()`, musíme nejdříve importovat modul `gettext`.)

`import` ➢ 45
`import` ➢ 194

Pojďme se nyní podívat na několik platných identifikátorů, jejichž názvy jsou v češtině. V kódu předpokládáme, že jsme již provedli příkaz `import math` a že proměnné `poloměr` a `stará_oblast` již byly vytvořeny v předchozí části programu:

```
π = math.pi
ε = 0.0000001
nová_oblast = π * poloměr * poloměr
if abs(nová_oblast - stará_oblast) < ε:
    print("oblast konverguje")
```

Použili jsme modul `math`, proměnnou epsilon (ε) jsme nastavili na velmi malé desetinné číslo a pomocí funkce `abs()` jsme získali absolutní hodnotu rozdílu mezi oblastmi – ke všem těmto prvků se ještě vrátíme v pozdější části této lekce. Pro nás je ale podstatné to, že pro identifikátory můžeme klidně používat znaky s diakritikou a znaky řecké abecedy. Stejně jednoduše bychom mohli vytvořit identifikátory s použitím arabských, čínských, hebrejských, japonských a ruských znaků anebo znaků z kteréhokoli jiného jazyka podporovaného znakovou sadou Unicode.

Nejjednodušším způsobem, jak otestovat, zda je daný identifikátor platný, je vyzkoušet přiřazení v interaktivním interpretu jazyka Python nebo v okně Python Shell v editoru IDLE. Zde je několik příkladů:

```
>>> stretch-factor = 1
SyntaxError: can't assign to operator (...)
>>> 2miles = 2
SyntaxError: invalid syntax (...)
>>> str = 3 # Platné, ale NEVHODNÉ
>>> l'impôt31 = 4
SyntaxError: EOL while scanning single-quoted string (...)
>>> l_impôt31 = 5
>>>
```

Syntaktické chyby ➤ 401

Použití neplatného identifikátoru způsobí vyvolání výjimky `SyntaxError`. V jednotlivých případech se část chybové zprávy umístěné v závorkách liší, a proto jsme ji nahradili výpustkem. První přiřazení selže, protože „-“ není písmenem, číslicí ani podtržítkem znakové sady Unicode. Druhé selže kvůli tomu, že počáteční znak není písmenem nebo podtržítkem znakové sady Unicode (pouze pokračovací znaky mohou být číslicemi). Pokud vytvoříme identifikátor, který je platný, tak se žádná výjimka nevyvolá, a to ani tehdy, je-li jeho název stejný jako některý z vestavěných datových typů, výjimek nebo funkcí. Třetí přiřazení tedy funguje, i když je krajně nevhodné. Čtvrté přiřazení selže, protože uvozovka není písmenem, číslicí ani podtržítkem znakové sady Unicode. Páté přiřazení je v pořádku.

Celočíselné typy

Python nabízí dva vestavěné celočíselné typy `int` a `bool`.* Celá čísla i logické hodnoty jsou neměnitelné, ale díky operátorům rozšířeného přiřazení jazyka Python si toho jen zřídkakdy všimnete. Při použití v logických výrazech má `0` a `False` hodnotu `False`, přičemž jakékoliv jiné celé číslo a `True` mají hodnotu `True`. Při použití v číselných výrazech se `True` vyhodnotí na `1` a `False` na `0`. To zname-

* Standardní knihovna nabízí také typ `fractions.Fraction` (racionální čísla s neomezenou přesností), který může být v některých specializovaných matematických a vědeckých kontextech užitečný.

ná, že můžeme psát poněkud zvláštní věci. Můžeme tak například inkrementovat celé číslo `i` pomocí výrazu `i += True`. Správně bychom měli samozřejmě použít `i += 1`.

Celá čísla

Velikost celého čísla je omezena pouze pamětí počítače, takže můžeme snadno vytvářet a zpracovávat celá čísla o velikosti stovek cifer, i když to bude pomalé ve srovnání s celými čísly, která lze reprezentovat nativně procesorem počítače.

Tabulka 2.2: Číselné operace a funkce

Syntaxe	Popis
x + y	Sečte číslo `x` a číslo `y`.
x - y	Odečte číslo `y` od čísla `x`.
x * y	Vynásobí číslo `x` číslem `y`.
x / y	Podělí číslo `x` číslem `y`, přičemž vždy vrátí typ `float` (nebo `complex`, je-li `x` nebo `y` typu `complex`).
x // y	Podělí číslo `x` číslem `y`, přičemž ořeže případnou desetinnou část, takže vždy vrátí typ `int` (viz také funkce `round()`).
x % y	Vrátí modul (zbytek) po dělení čísla `x` číslem `y`.
x ** y	Umocní číslo `x` mocninou `y` (viz také funkce `pow()`).
-x	Obrátí číslo `x`, takže změní jeho znaménko, pokud se nejedná o nulu (nula zůstane nezměněna).
+x	Neudělá nic (používá se pro lepší srozumitelnost kódu).
abs(x)	Vrátí absolutní hodnotu čísla `x`.
divmod(x, y)	Vrátí podíl a zbytek po dělení čísla `x` číslem `y` jako n-tici dvou celých čísel.
pow(x, y)	Umocní `x` na mocninu `y` (stejný výsledek jako operátor `**`).
pow(x, y, z)	Rychlejší alternativa k `(x ** y) % z`.
round(x, n)	Vrátí číslo `x` zaokrouhlené na `n` celočíselných cifer, je-li `n` záporné celé číslo, nebo na `n` desetinných míst, je-li `n` kladné celé číslo. Vrácená hodnota má stejný typ jako `x` (viz následující text).

N-tice ➢ 28

Typ `tuple` ➢ 110

Tabulka 2.3: Celočíselné převodní funkce

Syntaxe	Popis
bin(i)	Vrátí binární reprezentaci celého čísla `i` ve formě řetězce (např. `bin(1980) == '0b11110111100'`).
hex(i)	Vrátí šestnáctkovou reprezentaci čísla `i` ve formě řetězce (např. `hex(1980) == '0x7bc'`).
int(x)	Převede objekt `x` na celé číslo. V případě chyby vyvolá výjimku `ValueError` nebo výjimku `TypeError`, pokud datový typ objektu `x` nepodporuje převod na celé číslo. Je-li `x` desetinné číslo, tak se ořeže.
int(s, base)	Převede řetězec `s` na celé číslo. V případě chyby vyvolá výjimku `ValueError`. Je-li zadán volitelný argument `base`, pak by měl obsahovat celé číslo v rozmezí od 2 a do 36 včetně.
oct(i)	Vrátí osmičkovou reprezentaci `i` ve formě řetězce (např. `oct(1980) == '0o3674'`).

Celočíselné literály se standardně zapisují s použitím základu 10 (desítková čísla), lze však použít i čísla o jiném základu:

```
>>> 14600926                    # desítkové číslo
14600926
>>> 0b110111101100101011011110  # binární číslo
14600926
>>> 0o67545336                  # osmičkové číslo
14600926
>>> 0xDECADE                    # šestnáctkové číslo
14600926
```

Binární čísla se zapisují s předponou `0b`, osmičková čísla s předponou `0o`* a šestnáctková čísla s předponou `0x`. Pro všechny uvedené předpony lze použít také velká písmena.

Jak je patrné z tabulky 2.2, s celými čísly lze použít všechny obvyklé matematické funkce a operátory. Některé funkce poskytují vestavěné funkce, jako je `abs()` (např. `abs(i)` vrací absolutní hodnotu celého čísla `i`), a jiné poskytují operátory pro typ `int` (např. `i + j` vrátí součet celých čísel `i` a `j`.)

Nyní se podíváme pouze na jednu z funkcí uvedených v tabulce 2.2, protože všechny ostatní jsou dostatečné popsány v samotné tabulce. Funkce `round()` funguje v případě typu `float` očekávaným způsobem (např. `round(1.246, 2)` vrátí `1.25`), avšak v případě typu `int` nemá použití kladné zaokrouhlovací hodnoty žádný efekt a vede k vrácení téhož čísla, poněvadž zde nejsou žádná desetinná místa. Ovšem při použití záporné zaokrouhlovací hodnoty můžeme dosáhnout užitečného chování (např. `round(13579, -3)` vrátí `14000` a `round(34.8, -1)` vrátí `30.0`).

* Uživatelé zvyklí na jazyky ve stylu C si jistě všimli, že pro zápis osmičkového čísla samotná předpona `0` nestačí. V jazyku Python je nutné použít `0o` (nula a písmeno „*o*“).

Veškeré číselné binární operace (+, -, /, //, % a **) mají příslušné verze s rozšířeným přiřazením (+=, -=, /=, //=, %= a **=), kde `x op= y` je v běžném případě, kdy čtení hodnoty `x` nemá žádný vedlejší efekt, logickým ekvivalentem `x = x op y`.

Objekty lze vytvářet přiřazením literálu proměnné (např. `x = 17`) nebo zavoláním relevantního datového typu jako funkce (např. `x = int(17)`). Některé objekty (např. typu `decimal.Decimal`) můžeme vytvářet pouze pomocí datového typu, protože nemají odpovídající literálovou reprezentaci. Při vytvoření objektu pomocí jeho datového typu existují tři možné případy užití.

První případ užití nastává při volání datového typu bez argumentů. V takovém případě se vytvoří objekt s výchozí hodnotou (např. `x = int()` vytvoří celé číslo s hodnotou `0`). Všechny vestavěné typy lze zavolat bez argumentů.

K druhému případu užití dochází tehdy, když se datový typ volá s jedním argumentem. Je-li zadaný argument stejného typu, vytvoří se nový objekt, který je mělkou kopií původního objektu (mělké kopírování budeme probírat v lekci 3). Je-li zadán argument jiného typu, vyzkouší se jeho převod. Tuto situaci jsme si ukázali v tabulce 2.3 pro typ `int`. Je-li argument takového typu, který podporuje převod na daný typ, a tento převod selže, vyvolá se výjimka `ValueError`. V opačném případě se vrátí výsledný objekt daného typu. Pokud datový typ argumentu nepodporuje převod na daný typ, vyvolá se výjimka `TypeError`. Vestavěné typy `float` a `str` poskytují převod na celá čísla. Tento a další převody můžeme implementovat také v našich vlastních datových typech (viz Lekce 6).

Kopírování kolekcí ➢ 146

Převody mezi typy ➢ 250

Třetí případ užití nastává při použití dvou a více argumentů. To však nepodporují všechny typy a u těch, které to podporují, se typy a význam argumentů liší. U typu `int` jsou povoleny dva argumenty. První je řetězec, který představuje celé číslo, a druhý je číselný základ řetězcové reprezentace. Například `int("A4", 16)` vytvoří celé číslo `164` (toto použití ukazuje tabulka 2.3).

Bitové operátory jsou uvedeny v tabulce 2.4. Veškeré bitové binární operace (|, ^, &, << a >>) mají příslušné verze s rozšířeným přiřazením (|=, ^=, &=, <<= a >>=), kde `i op= j` je v běžném případě, kdy čtení hodnoty `i` nemá žádný vedlejší efekt, logickým ekvivalentem `i = i op j`.

3.1

Od Pythonu ve verzi 3.1 je k dispozici metoda `int.bit_length()`, která vrací počet bitů nezbytných k reprezentaci daného celého čísla. Například `(2145).bit_length()` vrátí 12. (Závorky jsou při použití celočíselného literálu nezbytné. V případě celočíselné proměnné je však můžeme vypustit.)

Pokud potřebujeme uchovávat mnoho příznaků pravda/nepravda, pak můžeme použít jediné celé číslo a testovat jednotlivé bity pomocí bitových operátorů. Toho stejného lze docílit sice méně kompaktním, ale zato pohodlnějším způsobem – pomocí seznamu logických hodnot.

Tabulka 2.4: Celočíselné bitové operátory

Syntaxe	Popis
i | j	Bitová disjunkce (OR) celých čísel `i` a `j`. Záporná čísla jsou reprezentována pomocí dvojkového doplňkového kódu.
i ^ j	Bitová nonekvivalence (XOR) čísel `i` a `j`.
i & j	Bitová konjunkce (AND) čísel `i` a `j`.

Syntaxe	Popis
i << j	Posune číslo `i` doleva o `j` bitů. Stejné jako `i * (2 ** j)`, ale bez kontroly přetečení.
i >> j	Posune číslo `i` doprava o `j` bitů. Stejné jako `i // (2 ** j)`, ale bez kontroly přetečení.
~i	Obrátí bity čísla `i`.

Logické hodnoty

Existují dva vestavěné Booleovské objekty: `True` a `False`. Podobně jako všechny ostatní datové typy jazyka Python (ať už vestavěné, knihovní nebo vlastní), také datový typ `bool` lze volat jako funkci. Bez argumentů vrací hodnotu `False`, s argumentem typu `bool` vrací jeho kopii a s libovolným dalším argumentem se pokusí daný objekt převést na typ `bool`. Všechny vestavěné datové typy a datové typy standardní knihovní lze převést na logickou (Booleovskou) hodnotu, přičemž je snadné implementovat převod na logickou hodnotu i ve vlastních datových typech. Zde je několik Booleovských přiřazení a výrazů:

```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

Logické operátory ➢ 34

Jak jsme si řekli již dříve, jazyk Python nabízí tři logické operátory: `and`, `or` a `not`. Operátory `and` a `or` používají logiku zkráceného vyhodnocování a vracejí operand, který rozhodl o výsledku. To znamená, že ne vždy vracejí `True` nebo `False`.

Programátoři, kteří jsou zvyklí na starší verzi Pythonu, používají někdy místo hodnot `True` a `False` hodnoty `1` a `0`. Tento přístup téměř vždy funguje bez problémů, ale nový kód by měl v místech, kde se vyžaduje logická hodnota, používat vestavěné booleovské objekty.

Typy s pohyblivou řádovou čárkou

Jazyk Python nabízí tři druhy hodnot s pohyblivou řádovou čárkou: vestavěné typy `float` a `complex` a typ `decimal.Decimal` ze standardní knihovny. Všechny tři uvedené typy jsou neměnitelné. Typ `float` uchovává čísla s pohyblivou řádovou čárkou s dvojitou přesností, jejichž rozsah závisí na kompilátoru jazyka C (nebo C# nebo Java) použitém pro sestavení Pythonu. Tato čísla mají omezenou přesnost a nelze je spolehlivě porovnávat na rovnost. Čísla typu `float` se zapisují s desetinnou tečkou nebo pomocí exponenciální notace (např. `0.0`, `4.`, `5.7`, `-2.5`, `-2e9`, `8.9e-4`).

Počítače nativně reprezentují čísla s pohyblivou řádovou čárkou pomocí kódování base 2, což znamená, že některá desetinná čísla lze reprezentovat přesně (např. 0,5), ale jiná jen přibližně (např. 0,1 nebo 0,2). Tato reprezentace dále používá fixní počet bitů, takže počet uchovávaných číslic je omezen. Zde je zajímavý příklad zapsaný do editoru IDLE:

3.0

```
3.0 >>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4000000000000004, -2.5, 0.00088999999999999995)
```

Nepřesnost není problém, který by se týkal pouze jazyka Python. Všechny programovací jazyky trpí tímto problémem s čísly s pohyblivou řádovou čárkou.

3.1

Python 3.1 vrací mnohem smysluplněji vypadající výstup:

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4, -2.5, 0.00089)
```

Když Python 3.1 vypisuje číslo s pohyblivou řádovou řádkou, pak ve většině případů používá algoritmus Davida Gaye. Ten vypisuje nejmenší možný počet číslic bez ztráty přesnosti. I když takto získáme hezčí výstup, nic tím nezměníme na faktu, že počítače (bez ohledu na použitý počítačový jazyk) v podstatě ukládají čísla s pohyblivou řádovou čárkou jako aproximace.

Pokud potřebujeme opravdu vysokou přesnost, pak můžeme postupovat dvěma způsoby. Jeden z nich spočívá ve využití typů `int` (např. pro práci s desetinami či setinami) a jejich škálování v případě potřeby. To vyžaduje jistou opatrnost, zejména pak tehdy, když provádíme dělení nebo počítáme procentní podíl. Další možností je využít typ `decimal.Decimal` z modulu `decimal`. Čísla tohoto typu provádějí výpočty, které jsou přesné až do námi stanovené úrovně (ve výchozím stavu až na 28 desetinných míst). Tato čísla mohou navíc přesným způsobem reprezentovat periodická čísla jako 0,1. Jejich zpracování je ale ve srovnání s čísly typu `float` mnohem pomalejší. Díky své přesnosti jsou čísla typu `decimal.Decimal` vhodná pro finanční výpočty.

Podporovány jsou též aritmetické operace se smíšenými typy, takže při použití typů `int` a `float` obdržíme typy `float` a při použití typů `float` a `complex` obdržíme `complex`. Čísla typu `decimal.Decimal` mají fixní přesnost, a proto je lze použít pouze s čísly typu `decimal`. Pokud s těmito čísly použijeme celá čísla (`int`), obdržíme výsledek typu `decimal.Decimal`. Pokud se pokusíme provést nějakou operaci s nekompatibilními typy, vyvolá se výjimka `TypeError`.

Čísla s pohyblivou řádovou čárkou

Veškeré číselné operace a funkce v tabulce 2.2 (strana 61) lze použít i s typem `float`, a to včetně verzí s rozšířením přiřazením. Datový typ `float` je možné zavolat i jako funkci. Bez argumentů vrátí 0.0, s argumentem typu `float` vrátí jeho kopii a argument jiného typu se pokusí převést na typ `float`. Při použití pro převod lze použít i řetězcový argument buď v obyčejné desetinné notaci, nebo s použitím exponenciální notace. Při výpočtech s čísly typu `float` můžeme obdržet výsledek `NaN` (Not a Number – není číslo) nebo „nekonečno“. Toto chování však není konzistentní napříč jednotlivými implementacemi a může se lišit v závislosti na systémem používané matematické knihovně.

Zde je jednoduchá funkce pro porovnání čísel typu `float` na rovnost omezená pouze přesností daného počítače:

```
def equal_float(a, b):
    return abs(a - b) <= sys.float_info.epsilon
```

Pro tento kód musíme importovat modul `sys`. Objekt `sys.float_info` nabízí celou řadu atributů. Atribut `sys.float_info.epsilon` je v podstatě nejmenším rozdílem dvou čísel s pohyblivou

řádovou čárkou, který dokáže daný stroj rozlišit. Na jistém 32bitovém stroji je to jen něco přes 0,000 000 000 000 000 2. (Toto číslo se tradičně označuje jako epsilon.) Typ `float` jazyka Python nabízí běžně spolehlivou přesnost až po 17 významných číslic.

Pokud napíšete objekt `sys.float_info` do editoru IDLE, zobrazí se všechny jeho atributy. Patří mezi ně minimální a maximální čísla s pohyblivou řádovou čárkou, která dokáže daný stroj reprezentovat. Pokud napíšete `help(sys.float_info)`, obdržíte určité informace o objektu `sys.float_info`.

Čísla s pohyblivou řádovou čárkou lze převést na celá čísla buď pomocí funkce `int()`, která vrátí celou část a desetinnou část zahodí, nebo pomocí funkce `round()`, která zohlední desetinnou část, anebo pomocí funkce `math.floor()`, resp. `math.ceil()`, která vrátí nejbližší menší resp. větší celé číslo. Metoda `float.is_integer()` vrátí hodnotu `True`, je-li zlomková část 0, přičemž zlomkovou reprezentaci lze získat pomocí metody `float.as_integer_ratio()`. Předpokládejme, že `x = 2.75`, pak volání `x.as_integer_ratio()` vrátí `(11, 4)`. Celá čísla je možné převést na čísla s pohyblivou řádovou čárkou pomocí funkce `float()`.

Čísla s pohyblivou řádovou čárkou lze reprezentovat jako řetězce v šestnáctkovém formátu pomocí metody `float.hex()`. Takové řetězce lze pak převést zpět na čísla s pohyblivou řádovou čárkou metodou `float.fromhex()`:

```
s = 14.25.hex()        # str s == '0x1.c800000000000p+3'
f = float.fromhex(s)  # float f == 14.25
t = f.hex()            # str t == '0x1.c800000000000p+3'
```

Exponent je místo písmena `e` označen písmenem `p` (power – mocnina), protože `e` je platná šestnáctková číslice.

Tabulka 2.5: Funkce a konstanty modulu `Math`

Syntaxe	Popis
math.acos(x)	Vrátí arkus kosinus x (v radiánech).
math.acosh(x)	Vrátí hyperbolický arkus kosinus x (v radiánech).
math.asin(x)	Vrátí arkus sinus x (v radiánech).
math.asinh(x)	Vrátí hyperbolický arkus sinus x (v radiánech).
math.atan(x)	Vrátí arkus tangens x (v radiánech).
math.atan2(y, x)	Vrátí arkus tangens x / y (v radiánech).
math.atanh(x)	Vrátí hyperbolický arkus tangens x (v radiánech).
math.ceil(x)	Vrátí $\lceil x \rceil$, tj. nejmenší celé číslo (`int`) větší nebo stejné jako x (např. `math.ceil(5.4) == 6`).
math.copysign(x,y)	Vrátí číslo x se znaménkem čísla y.
math.cos(x)	Vrátí kosinus x (v radiánech).

Syntaxe	Popis
math.cosh(x)	Vrátí hyperbolický kosinus x (v radiánech).
math.degrees(r)	Převede číslo `r` typu `float` z radiánů na stupně.
math.e	Konstanta e (přibližně `2.7182818284590451`).
math.exp(x)	Vrátí e^x, tj. `math.e**x`.
math.fabs(x)	Vrátí $\|x\|$, tj. absolutní hodnotu x jako typ `float`.
math.factorial(x)	Vrátí x!
math.floor(x)	Vrátí $\lfloor x \rfloor$, tj. největší celé číslo (`int`) menší nebo stejné jako x (např. `math.floor(5.4) == 5`).
math.fmod(x, y)	Vrátí modul (zbytek) po dělení čísla x číslem y (pro čísla typu `float` vrací lepší výsledky než operátor `%`).
math.frexp(x)	Vrátí dvouprvkovou n-tici s mantisou (jako typ `float`) a exponentem (jako typ `int`) tak, že $x = m \times 2^e$ (viz `math.ldexp()`).
math.fsum(i)	Vrátí součet hodnot v iterovatelném objektu `i` jako číslo typu `float`.
math.hypot(x, y)	Vrátí odmocninu z $\sqrt{x^2 + y^2}$.
math.isinf(x)	Vrátí hodnotu `True`, pokud číslo x typu `float` je ±`inf`(±∞).
math.isnan(x)	Vrátí hodnotu `True`, pokud číslo x typu `float` je NaN (není číslo).
math.ldexp(m, e)	Vrátí $m \times 2^e$, což je v podstatě inverze funkce `math.frexp()`.
math.log(x, b)	Vrátí $\log_b x$ (b je volitelné a jeho výchozí hodnota je `math.e`).
math.log10(x)	Vrátí $\log_{10} x$.
math.log1p(x)	Vrátí $\log_e(1+ x)$ (přesné i v případě, kdy je x blízko 0).
math.modf(x)	Vrátí zlomkovou a celou část čísla x jako dvě čísla typu `float`.
math.pi	Konstanta π (přibližně 3.1415926535897931).
math.pow(x, y)	Vrátí x^y jako číslo typu `float`.
math.radians(d)	Převede číslo `d` typu `float` ze stupňů na radiány.
math.sin(x)	Vrátí sinus x (v radiánech).
math.sinh(x)	Vrátí hyperbolický sinus x (v radiánech).
math.sqrt(x)	Vrátí odmocninu z čísla $\sqrt{x}$.

N-tice ➢ 28

Typ `tuple` ➢ 110

Syntaxe	Popis
math.tan(x)	Vrátí tangens čísla x (v radiánech).
math.tanh(x)	Vrátí hyperbolický tangens čísla x (v radiánech).
math.trunc(x)	Vrátí celou část x jako číslo typu `int` (stejné jako `int(x)`).

Jak je patrné z tabulky 2.5, nabízí modul `math` kromě vestavěných funkcí pro práci s pohyblivou řádovou čárkou ještě spoustu dalších funkcí, které pracují na těchto číslech. Zde je několik úryvků kódu, které ukazují, jak se používají funkce tohoto modulu:

3.x

```
>>> import math
>>> math.pi * (5 ** 2)  # Python 3.1 vypíše: 78.53981633974483
78.539816339744831
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732)   # Python 3.1 vypíše: (0.7319999999999993, 13.0)
(0.73199999999999932, 13.0)
```

Funkce `math.hypot()` vypočítá vzdálenost od počátku k bodu (x, y), takže vrátí stejný výsledek jako `math.sqrt((x ** 2) + (y ** 2))`.

Modul `math` je velice závislý na matematické knihovně, s níž byl Python zkompilován. To znamená, že některé chybové podmínky a hraniční případy se mohou na různých platformách lišit.

Komplexní čísla

Typ `complex` je neměnitelný datový typ uchovávající dvojici čísel typu `float`, kdy jedno reprezentuje reálnou a druhé imaginární složku komplexního čísla. Literály komplexních čísel se zapisují s reálnou a imaginární složkou spojenou znaménkem + nebo -, přičemž za imaginární složkou se nachází písmeno `j`.* Zde je několik příkladů: `3.5+2j`, `0.5j`, `4+0j` a `-1-3.7j`. Všimněte si, že pokud je reálná část, tak ji můžeme zcela vypustit.

Jednotlivé složky komplexního čísla jsou přístupné jako atributy `real` a `imag`:

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

Až na operace `//`, `%`, `divmod()` a funkci `pow()` se třemi argumenty lze s komplexními čísly použít všechny číselné operace a funkce uvedené v tabulce 2.2 (strana 61) včetně verzí s rozšířeným přiřazením. Kromě toho nabízí typ `complex` metodu `conjugate()`, která změní znaménko imaginární části:

```
>>> z.conjugate()
(-89.5-2.125j)
>>> 3-4j.conjugate()
(3+4j)
```

* Matematici mohou pro označení $\sqrt{-1}$ používat i, avšak Python dodržuje označení j tradiční pro inženýrské prostředí.

Všimněte si, že jsme metodu zavolali na literálovém komplexním čísle. Python nám obvykle umožňuje volat metody nebo přistupovat k atributům na libovolném literálu, pokud datový typ literálu poskytuje volanou metodu nebo atribut. Nicméně to se netýká speciálních metod, protože ty mají vždy odpovídající operátory, jako je +, které by se měly použít místo nich. Například `4j.real` vrátí `0.0`, `4j.imag` vrátí `4.0` a `4j + 3+2j` vrátí `3+6j`.

Datový typ `complex` lze volat jako funkci. Bez parametrů vrátí `0j`, s argumentem typu `complex` vrátí jeho kopii a argument jiného typu se pokusí převést na typ `complex`. Při použití pro převod přijímá funkce `complex()` buď jediný řetězec, nebo jedno či dvě čísla typu `float`. Je-li zadáno pouze jedno, pak je imaginární část nastavena na `0j`.

Funkce v modulu `math` s komplexními čísly nefungují. Jedná se o úmyslné návrhové rozhodnutí, které zajišťuje, aby uživatelé modulu `math` neobdrželi v některých situacích nevědomě komplexní čísla, ale aby se raději vyvolaly výjimky.

Uživatelé komplexních čísel mohou importovat modul `cmath`, který nabízí verze většiny trigonometrických a logaritmických funkcí modulu `math` pro komplexní čísla plus několik další funkcí specifických pro komplexní čísla, jako jsou například `cmath.phase()`, `cmath.polar()` a `cmath.rect()`, a také konstanty `cmath.pi` a `cmath.e`, které uchovávají stejné hodnoty typu `float` jako jejich protějšky v modulu `math`.

Desetinná čísla

V mnoha aplikacích nepředstavují nepřesnosti, které se mohou vyskytnout při použití čísel typu `float`, žádný problém a tak jako tak jsou vyváženy rychlostí výpočtu nabízené typem `float`. Ovšem v některých případech potřebujeme dát přednost přesnosti, a to i za cenu zpomalení výpočtu. Modul `decimal` poskytuje neměnitelný typ `Decimal`. Čísla tohoto typu jsou tak přesná, jak si sami stanovíme. Výpočty zahrnující čísla typu `Decimal` jsou ve srovnání s čísly typu `float` pomalejší, ale zda to bude postřehnutelné, závisí na konkrétní aplikaci.

K vytvoření čísla typu `Decimal` musíme nejdříve importovat modul `decimal`:

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
>>> a + b
Decimal('64197.012345678987654321')
```

Desetinná čísla se vytvářejí pomocí funkce `decimal.Decimal()`. Tato funkce přijímá celé číslo nebo řetězec, ne však číslo typu `float`, protože tato čísla jsou uchovávána nepřesně, zatímco desetinná čísla jsou reprezentována přesně. Pokud použijeme řetězec, můžeme použít desetinnou nebo exponenciální notaci. Přesná reprezentace čísel typu `decimal.Decimals` znamená kromě vyšší přesnosti také schopnost porovnávání na rovnost.

Počínaje Pythonem ve verzi 3.1 je možné převádět čísla typu `float` na čísla typu `decimal` funkcí `decimal.Decimal.from_float()`. Tato funkce přijímá číslo typu `float` a vrací číslo typu `decimal.Decimal`, které je nejblíže číslu, které aproximuje zadaný argument.

S typem `decimal.Decimal` lze použít všechny číselné operace a funkce uvedené v tabulce 2.2 (strana 61) včetně verzí s rozšířeným přiřazením. Je tu však několik pravidel, která je nutné dodržovat. Pokud je levý operand operátoru `**` typu `decimal.Decimal`, pak pravý operand musí být celé číslo. A podobné je to i u funkce `pow()`. Je-li její první argument typu `decimal.Decimal`, pak její druhý i volitelný třetí argument musí být celé číslo.

Moduly `math` a `cmath` nejsou vhodné pro použití s typem `decimal.Decimal`, avšak některé funkce poskytované modulem `math` jsou k dispozici jako metody typu `decimal.Decimal`. Například pro výpočet e^x, kde `x` je typu `float`, napíšeme `math.exp(x)`. Je-li však `x` typu `decimal.Decimal`, pak napíšeme `x.exp()`. Z diskuze v Oblasti č. 3 předchozí lekce (strana 29) je patrné, že metoda `x.exp()` je v podstatě syntaktickým cukrem pro metodu `decimal.Decimal.exp(x)`.

Datový typ `decimal.Decimal` nabízí také metodu `ln()`, která počítá přirozený (o základu e) logaritmus (podobně jako funkce `math.log()` s jedním argumentem), `log10()` a `sqrt()` společně s řadou dalších metod specifických pro datový typ `decimal.Decimal`.

Čísla typu `decimal.Decimal` pracují v rámci určitého kontextu. Tímto kontextem je kolekce nastavení, jež ovlivňují způsob chování těchto čísel. Kontext stanoví přesnost, která by se měla používat (výchozí je 28 desetinných míst), zaokrouhlovací techniku a některé další detaily.

V některých situacích je rozdíl v přesnosti mezi čísly typu `float` a `decimal.Decimal` zřejmý:

```
>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.9047619048
>>> print(decimal.Decimal(23) / decimal.Decimal("1.05"))
21.90476190476190476190476190
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.90476190476190476190476190')
```

Ačkoliv dělení pomocí typu `decimal.Decimal` je mnohem přesnější než s použitím typu `float`, v tomto případě (32bitový stroj) se rozdíl objevil až na patnáctém desetinném místě. V mnoha situacích je tak malý rozdíl nepodstatný, a proto budeme ve všech příkladech v této knize používat pro čísla s pohyblivou řádovou čárkou typ `float`.

Další věcí, na kterou je třeba upozornit, je skutečnost, že poslední dva výše uvedené příklady poprvé odhalují, že při vypisování objektu dochází v pozadí k určitému formátování. Když se zavolá funkce `print()` na výsledku `decimal.Decimal(23) / decimal.Decimal("1.05")`, vypíše se holé číslo. Tento výstup tedy probíhá v řetězcové formě. Pokud jednoduše zadáme výraz, pak obdržíme výstup z typu `decimal.Decimal`. Jedná se tedy o výstup v reprezentační formě. Všechny objekty jazyka Python mají dvě formy výstupu. Řetězcová forma je navržena tak, aby byla čitelná pro člověka. Reprezentační forma je navržena pro vytvoření výstupu, který po předložení interpretu jazyka Python (je-li to možné) povede k vytvoření reprezentovaného objektu. K tomuto tématu se ještě vrátíme v následující části, v níž se budeme věnovat řetězcům, a pak znovu v lekci 6, kde budeme probírat implementaci řetězcové a reprezentační formy pro naše vlastní datové typy.

Veškeré podrobnosti ohledně modulu `decimal` (včetně dalších příkladů a seznamu otázek a odpovědí), které jsou nad rámec této knihy, najdete v příslušné dokumentaci.

Řetězce

Řetězce představují neměnitelný datový typ `str`, který uchovává posloupnost znaků ze znakové sady Unicode. Datový typ `str` lze pro vytvoření řetězcových objektů zavolat také jako funkci. Bez argumentů vrátí prázdný řetězec, s neřetězcovým argumentem vrátí jeho řetězcovou podobu a s řetězcovým argumentem vrátí jeho kopii. Funkci `str()` lze též použít jako převodní funkci. V takovém případě by prvním argumentem měl být řetězec nebo něco, co lze na řetězec převést. Pak mohou následovat další dva nepovinné argumenty, z nichž jeden stanoví kódování, které se má použít, a druhý způsob zpracování kódovacích chyb.

Kódování znaků ➢ 95

Již dříve jsme si řekli, že se řetězcové literály vytvářejí pomocí uvozovek a že je jen na nás, zda použijeme jednoduché či dvojité uvozovky, jsou-li na obou stranách stejné. Kromě toho můžeme použít řetězec s trojitými uvozovkami, což v řeči jazyka Python znamená, že začíná a končí *třemi uvozovkami* (ať už jednoduchými nebo dvojitými):

```
text = """Řetězec s trojitými uvozovkami může obsahovat 'uvozovky' a
také "uvozovky" bez jakýchkoli formalit. Můžeme též potlačit nové řádky \
tkaže tento řetězec bude mít pouze dva řádky."""
```

Pokud bychom chtěli použít uvozovky uvnitř běžně uzavřeného řetězce, pak je můžeme zapsat přímo, pokud se liší od ohraničujících uvozovek. V opačném případě je musíme potlačit zpětným lomítkem:

```
a = "Jednoduché 'uvozovky' jsou v pořádku; \"dvojité\" musíme potlačit."
b = 'Jednoduché \'uvozovky\' musíme potlačit; "dvojité" jsou v pořádku.'
```

Tabulka 2.6: Speciální řetězcové posloupnosti v jazyku Python

Speciální řetězcová posloupnost	Popis
`\`*`newline`*	Potlač (tj. ignoruj) nový řádek
`\\`	Zpětné lomítko (\)
`\'`	Jednoduché uvozovky (')
`\"`	Dvojité uvozovky (")
`\a`	Zvonek ve znakové sadě ASCII (BEL)
`\b`	Zpětné lomítko ve znakové sadě ASCII (BS – Backspace)
`\f`	Posun na další stránku ve znakové sadě ASCII (FF – Form Feed)
`\n`	Posun na další řádek ve znakové sadě ASCII (LF – Line Feed)
`\N{`*`název`*`}`	Znak se zadaným názvem ze znakové sady Unicode

Speciální řetězcová posloupnost	Popis
`\ooo`	Znak se zadanou osmičkovou hodnotou
`\r`	Návrat tiskové hlavy na začátek ve znakové sadě ASCII (CR – Carriage Return)
`\t`	Tabulátor ve znakové sadě ASCII (TAB)
`\uhhhh`	Znak se zadanou 16bitovou hexadecimální hodnotou ze znakové sady Unicode
`\Uhhhhhhhh`	Znak se zadanou 32bitovou hexadecimální hodnotou ze znakové sady Unicode
`\v`	Vertikální tabulátor ve znakové sadě ASCII (VT – Vertical Tab)
`\xhh`	Znak se zadanou 8bitovou hexadecimální hodnotou

Python používá nové řádky jako terminátor příkazů. Výjimkou je vnitřek kulatých závorek (`()`), hranatých závorek (`[]`), složených závorek (`{}`) nebo vnitřek řetězců s trojitými uvozovkami. Nové řádky lze v řetězcích s trojitými uvozovkami použít přímo, přičemž do libovolného řetězcového literálu můžeme vložit nový řádek pomocí speciální posloupnosti `\n`. Všechny speciální řetězcové posloupnosti jazyka Python uvádí tabulka 2.6. V některých situacích (např. při psaní regulárních výrazů) potřebujeme vytvořit řetězec se spoustou zpětných lomítek (regulární výrazy jsou předmětem lekce 13). To může být nepohodlné, protože každé musíme dalším zpětným lomítkem potlačit:

```
import re
phone1 = re.compile("^((?:[(]\\d+[)])?\\s*\\d+(?:-\\d+)?)$")
```

Řešením jsou *holé* řetězce. Jedná se libovolným způsobem ohraničené řetězce, které mají před první uvozovkou umístěno písmeno `r`. Uvnitř takovýchto řetězců jsou všechny znaky považovány za literály, takže není nutné používat speciální řetězcové posloupnosti. Zde je regulární výraz pro telefon napsaný jako holý řetězec.

```
phone2 = re.compile(r"^((?:[(]\d+[)])?\s*\d+(?:-\d+)?)$")
```

Pokud chceme bez použití trojitých uvozovek napsat dlouhý řetězcový literál rozprostřený na dva či více řádků, můžeme použít jednu z následujících možností:

```
t = "Toto není nejvhodnější způsob spojení dvou dlouhých řetězců, " + \
    "protože se opírá o nevzhledné potlačení nového řádku"

s = ("Toto je pěkný způsob spojení dvou dlouhých řetězců, "
     "který se opírá o řetězení řetězcových literálů.")
```

Všimněte si, že ve druhém případě musíme pro vytvoření jediného výrazu použít závorky. Bez nich by se proměnné `s` přiřadil pouze první řetězec a druhý by způsobil výjimku `IndentationError`. Jistá část dokumentace jazyka Python (*http://docs.python.org/dev/howto/doanddont.html*) doporučuje používat pro příkazy jakéhokoliv druhu rozprostřené na více řádků vždy závorky, ne potlačování nových řádků. Tímto doporučením se zde budeme řídit.

Vzhledem k tomu, že soubory `.py` používají ve výchozím nastavení kódování UTF-8, můžeme do svých řetězcových literálů psát libovolné znaky přímo. Můžeme také do řetězců umístit libovolné znaky ze znakové sady Unicode pomocí hexadecimální speciální posloupnosti nebo pomocí názvů znakové sady Unicode:

```
>>> euros = " \N{euro sign} \u20AC \U000020AC"
>>> print(euros)
```

V tomto případě bychom nemohli použít hexadecimální speciální posloupnost, protože ty jsou omezeny na dvě číslice, takže nemohou přesáhnout hodnotu `0xFF`. Je třeba poznamenat, že u názvů znaků ve znakové sadě Unicode se nerozlišuje velikost písmen a mezery, které obsahují, jsou volitelné.

Pokud chceme pro určitý znak v řetězci zjistit jeho kódový bod ve znakové sadě Unicode (celočíselná hodnota přiřazená danému znaku v kódování Unicode), můžeme použít vestavěnou funkci `ord()`:

Kódování znaků ➤ 95

```
>>> ord(euros[0])
8364
>>> hex(ord(euros[0]))
'0x20ac'
```

Podobně můžeme pomocí vestavěné `chr()` funkce převést libovolné celé číslo, jež reprezentuje platný kódový bod, na odpovídající znak ve znakové sadě Unicode:

```
>>> s = "anarchitsti jsou " + chr(8734) + chr(0x23B7)
>>> s
'anarchists are ∞√'
>>> ascii(s)
"'anarchists are \u221e\u23b7'"
```

Pokud v editoru IDLE zadáme jen samotné `s`, pak obdržíme výstup v jeho řetězcové formě, což u řetězců znamená výstup znaků uzavřených do uvozovek. Pokud chceme pouze znaky ze znakové sady ASCII, můžeme použít vestavěnou funkci `ascii()`, která vrátí reprezentační formu svého argumentu pomocí 7bitových znaků ASCII všude, kde je to možné, a pomocí co nejkratší formy speciální řetězcové posloupnosti `\xhh`, `\uhhhh` nebo `\Uhhhhhhhh` ve všech ostatních případech. O způsobu přesné kontroly výstupu řetězců si více řekneme v pozdější části této lekce.

str.format() ➤ 83

Porovnávání řetězců

Řetězce podporují obvyklé porovnávací operátory `<`, `<=`, `==`, `!=`, `>` a `>=`. Tyto operátory porovnávají řetězce po jednotlivých bajtech v paměti. Při porovnávání (např. při řazení seznamu řetězců) však mohou vyvstat dva problémy, které nepostihují jen Python, ale každý programovací jazyk používající řetězce s kódováním Unicode.

Prvním problém je, že některé znaky znakové sady Unicode lze reprezentovat dvěma či více různými posloupnostmi bajtů. Například znak A (kódový bod `0x00C5` ve znakové sadě Unicode) může být reprezentován bajty v kódování UTF-8 třemi různými způsoby: `[0xE2, 0x84, 0xAB]`, `[0xC3, 0x85]` a `[0x41, 0xCC, 0x8A]`. Tento problém naštěstí můžeme vyřešit. Pokud importujeme modul

Kódování znaků ➤ 95

`unicodedata` a zavoláme metodu `unicodedata.normalize()`, které jako argumenty předáme `"NFKC"` a řetězec obsahující znak A zakódovaný v libovolné platné posloupnosti, pak obdržíme řetězec, který bude v kódování UTF-8 vždy obsahovat pro znak A posloupnost bajtů `[0xC3, 0x85]`.

Druhý problém tkví v tom, že řazení některých znaků je závislé na konkrétním jazyku. Jako příklad můžeme uvést písmeno „ä", které je ve Švédsku řazeno za písmenem „z", kdežto v Německu je řazeno jako „ae". Dalším příkladem je angličtina, kde se písmeno „ø" řadí standardním způsobem, zatímco v Dánsku a Norsku se řadí až za písmeno „z". Těchto problémů je celá řada a mohou být dále zkomplikovány tím, když jednu aplikaci používají lidé různých národností (kteří tím pádem očekávají odlišné pořadí při řazení) nebo když řetězce obsahují směsici více jazyků (např. něco v češtině, něco v angličtině), přičemž některé znaky (např. šipky, ozdobné znaky či matematické symboly) nemají ve skutečnosti žádnou smysluplnou pozici pro řazení.

Python ze zásady (aby zamezil zákeřným chybám) nedělá žádné odhady. V případě porovnávání řetězců se tedy používá bajtová reprezentace řetězců v paměti. Řetězce se tak řadí podle kódových bodů znakové sady Unicode, což v případě angličtiny znamená řazení podle kódování ASCII. Při porovnávání s malými a velkými písmeny proto v angličtině obdržíme mnohem přirozenější uspořádání. Normalizace většinou není potřeba, pokud řetězce nejsou z externího zdroje, jako jsou soubory nebo síťové sokety. Avšak ani v těchto případech by se normalizace neměla provádět, nemáme-li jistotu, že je skutečně zapotřebí. Řadicí metody Pythonu si samozřejmě můžeme přizpůsobit, o čemž se přesvědčíme v lekci 3. Celý problém řazení řetězců v kódování Unicode je podrobně vysvětlen v dokumentu s názvem Unicode Collation Algorithm (viz *www.unicode.org/reports/tr10/*).

Řezání a krokování řetězců

Oblast č. 3 ➤ 28

V Oblasti č. 3 jsme se dozvěděli, že jednotlivé prvky v posloupnosti, a tedy i znaky v řetězci lze extrahovat pomocí operátoru pro přístup k prvku (`[]`). Ve skutečnosti je tento operátor mnohem všestrannější a je možné jej použít pro extrakci ne jen jediného prvku či znaku, ale pro celý řez (podposloupnost) prvků či znaků. V tomto kontextu jej proto označujeme jako řezací (slicing) operátor.

Nejdříve se podíváme na extrahování jednotlivých znaků. Pozice indexů v řetězci začínají od 0 a pokračují až do délky řetězce minus 1. Můžeme nicméně použít i zápornou pozici indexu. Tyto pozice se počítají od posledního znaku zpět k prvnímu. Mějme přiřazení `s = "Ostrý šíp"`. Pak obrázek 2.1 ukazuje všechny platné pozice indexů pro řetězec `s`.

s[-9]	s[-8]	s[-7]	s[-6]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]
O	s	t	r	ý		š	í	p
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]

Obrázek 2.1: Pozice indexů v řetězci

Záporné indexy jsou překvapivě užitečné, zejména index -1, který vždy ukazuje na poslední znak v řetězci. Přístup k indexu mimo rozsah (nebo k libovolnému indexu v prázdném řetězci) způsobí vyvolání výjimky `IndexError`.

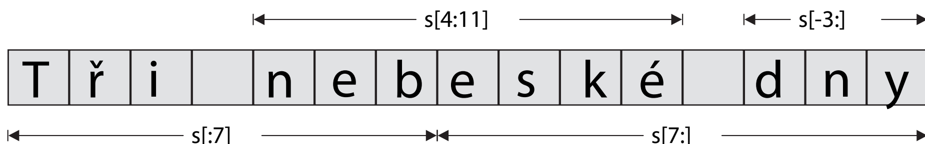
Řezací operátor má tři varianty syntaxe:

```
seq[začátek]
seq[začátek:konec]
seq[začátek:konec:krok]
```

Symbol `seq` představuje libovolnou posloupnost, jako je seznam, řetězec nebo n-tice. Hodnoty `začátek`, `konec` a `krok` jsou vždy celá čísla (nebo proměnné uchovávající celá čísla). První variantu syntaxe jsme již používali. Tato varianta extrahuje prvek posloupnosti na pozici `začátek`. Druhá varianta syntaxe extrahuje řez, jehož počáteční prvek je na pozici `začátek`, a koncový prvek *před* prvkem na pozici `konec`. Ke třetí variantě se dostaneme za okamžik.

Pokud použijeme druhou variantu syntaxe (s jednou dvojtečkou), pak můžeme kterýkoli z indexů vynechat. Vynecháme-li začáteční index, bude mít výchozí hodnotu 0. Pokud vynecháme koncový index, bude mít výchozí hodnotu `len(seq)`. To znamená, že pokud vynecháme oba indexy, tedy například `s[:]`, je to stejné, jako kdybychom napsali `s[0:len(s)]`, čímž extrahujeme (tj. zkopírujeme) celou sekvenci.

Mějme přiřazení `s = "Tři nebeské dny"`. Pak obrázek 2.2 znázorňuje několik ukázkových řezů pro řetězec `s`.



Obrázek 2.2: Řezání posloupnosti

Jednou z možností, jak vložit do řetězce podřetězec, je spojit řezání se zřetězením:

```
>>> s = s[:12] + "be" + s[12:]
>>> s
'Tři nebeské bedny'
```

Vzhledem k tomu, že se text „be“ objevuje také v původním řetězci, můžeme stejného výsledku docílit přiřazením `s[:12] + s[6:8] + s[12:]`.

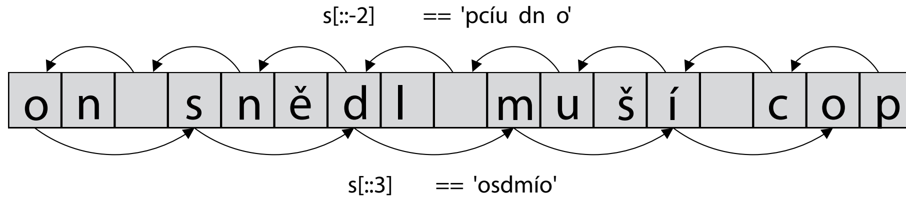
Používání operátoru + pro spojení a += pro připojení není příliš efektivní, pokud pracujeme s více řetězci. Pro spojování většího počtu řetězců je obvykle nejlepší sáhnout po metodě `str.joint()`, o které si více řekneme v následující podčásti.

Řetězcové operátory a metody ➤ 76

Třetí varianta syntaxe řezacího operátoru (se dvěma dvojtečkami) je podobná druhé, pouze se místo všech znaků extrahuje vždy jen i-tý znak, kde i je definováno zadaným krokem. A podobně jako ve druhé variantě syntaxe můžeme i zde kterýkoliv index vynechat. Pokud vynecháme počáteční index, bude mít výchozí hodnotu 0, pokud ovšem není zadán záporný krok. V takovém případě bude mít výchozí hodnotu -1. Pokud vynecháme koncový index, bude mít výchozí hodnotu `len(seq)`. Je-li ovšem krok záporný, pak bude mít koncový index výchozí hodnotu ležící před začátkem řetězce. Pokud použijeme dvě dvojtečky, ale vynecháme velikost kroku, pak se pro krok použije výchozí hodnota 1. Je však naprosto zbytečné používat dvě dvojtečky s krokem velikosti 1, protože se tak jako tak jedná o výchozí hodnotu kroku. Dále je třeba si uvědomit, že krok velikosti nula není povolen.

Mějme přiřazení `s = "on` snědl muší cop`"`. Pak obrázek 2.3 znázorňuje několik ukázkových krokování pro řetězec `s`.

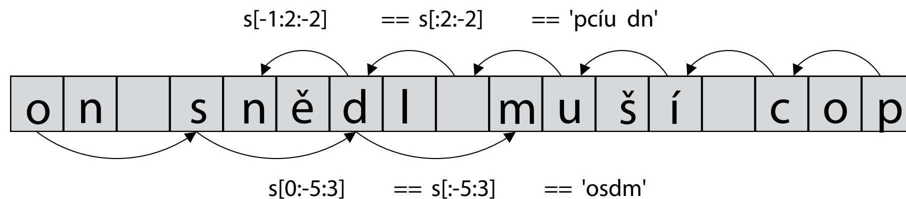
```
"he ate camel food"
"on snědl muší cop"
```



Obrázek 2.3: Krokování posloupnosti

Zde jsme použili výchozí počáteční a koncové indexy, takže `s[::-2]` začíná na posledním znaku a extrahuje každý druhý znak ve směru k začátku řetězce. Podobně `s[::3]` začíná na prvním znaku a extrahuje každý třetí znak směrem ke konci řetězce.

Je též možné zkombinovat řezací indexy s krokováním, což zachycuje obrázek 2.4.



Obrázek 2.4: Řezání a korkování posloupnosti

Krokování se nejčastěji nepoužívá s řetězci, ale s posloupnostmi jiného typu. Existuje však jedna situace, v níž se používá i pro řetězce:

```
>>> s, s[::-1]
('on snědl muší cop', 'poc íšum lděns no')
```

Krokování o -1 znamená, že se extrahuje každý znak od konce na začátek, čímž získáme obrácený řetězec.

Řetězcové operátory a metody

Iterovatelné operátory a funkce ➤ 138

Třída `Sized` ➤ 369

Řetězce jsou neměnitelné posloupnosti, a proto veškerá funkčnost, která je dostupná u neměnitelných posloupností, je k dispozici také u řetězců. To zahrnuje testování příslušnosti, spojování operátorem +, připojování operátorem += a replikace rozšířeným přiřazením *=. V této podčásti se kromě řady řetězcových metod podíváme také na všechny tyto operace v kontextu řetězců. Tabulka 2.7 uvádí přehled všech řetězcových metod kromě dvou poněkud specializovanějších (`str.maketrans()` a `str.translate()`), které si ve stručnosti probereme později.

Řetězce jakožto posloupnosti znají pojem velikosti, a proto na nich můžeme volat metodu `len()`. Vrácená délka představuje počet znaků v řetězci (nula pro prázdný řetězec).

Viděli jsme, že operátor + je přetížen i pro řetězce, kde funguje jako spojování řetězců. Pro případy, kdy potřebujeme spojovat spoustu řetězců, nabízí metoda `str.join()` lepší řešení. Tato metoda přijímá jako svůj argument posloupnost (např. seznam nebo n-tici řetězců), jejíž prvky spojí do jediného řetězce, přičemž mezi ně umístí řetězec, na němž byla metoda zavolána:

```
>>> treatises = ["Aritmetika", "Kuželosečky", "Základy"]
>>> " ".join(treatises)
'Aritmetika Kuželosečky Základy'
>>> "-<>-".join(treatises)
'Aritmetika-<>-Kuželosečky-<>-Základy'
>>> "".join(treatises)
'AritmetikaKuželosečkyZáklady'
```

První příklad demonstrující spojování s jediným znakem (v tomto případě s mezerou) se používá pravděpodobně nejčastěji. Třetí příklad představuje díky prázdnému řetězci čisté spojení, což znamená, že se posloupnost řetězců spojí bez jakékoli výplně.

Metodu `str.joint()` můžeme společně s vestavěnou funkcí `reversed()` použít k obrácení řetězce (např. `"".join(reversed(s))`), ačkoliv stejného výsledky lze snadněji docílit krokováním (např. `s[::-1]`).

Operátor * funguje jako replikace řetězce:

```
>>> s = "=" * 5
>>> print(s)
=====
>>> s *= 10
>>> print(s)
==================================================
```

Jak je z tohoto příkladu patrné, můžeme použít také rozšířenou verzi přiřazení operátoru replikace.[*]

Při aplikaci na řetězce vrací operátor příslušnosti hodnotu `True` v případě, kdy je jeho levý argument podřetězcem pravého řetězcového argumentu nebo kdy jsou si jeho argumenty rovny.

Pro situace, kdy potřebujeme najít pozici jednoho řetězce uvnitř jiného, máme k dispozici dvě metody. První je metoda `str.index()`, která vrací indexovou pozici podřetězce nebo v případě neúspěchu vyvolá výjimku `ValueError`. Druhou je metoda `str.find()`, která vrací indexovou pozici podřetězce nebo -1 v případě neúspěchu. Obě metody přijímají jako první argument řetězec, který se má najít, a dále mohou přijímat několik volitelných argumentů. Druhým argumentem může být počáteční a třetím koncová pozice v prohledávaném řetězci.

---

* Řetězce podporují také operátor % pro formátování. Tento operátor je však zastaralý a slouží pouze ke snazšímu přechodu z Pythonu 2 na Python 3. V příkladech této knihy jej proto vůbec nebudeme používat.

Tabulka 2.7: Řetězcové metody

Syntaxe	Popis
s.capitalize()	Vrátí kopii řetězce `s`, jejíž první písmeno bude převedené na velké písmeno (viz též metoda `str.title()`).
`s.center(délka, znak)`	Vrátí kopii řetězce `s` vycentrovanou v řetězci zadané délky a vyplněnou mezerami nebo případně zadaným znakem, což je řetězec délky 1 (viz metody `str.ljust()`, `str.rjust()` a `str.format()`).
`s.count(t, začátek, konec)`	Vrátí počet výskytů řetězce `t` v řetězci `s` (nebo v řezu *`začátek`*`:`*`konec`* řetězce `s`).
`s.encode(kódování, chyby)`	Vrátí objekt typu `bytes` představující řetězec, který používá výchozí kódování nebo zadané kódování, přičemž chyby se ošetří podle volitelného argumentu *`chyby`*.
`s.endswith(x, začátek, konec)`	Vrátí hodnotu `True`, pokud `s` (nebo řez *`začátek`*`:`*`konec`* řetězce `s`) končí řetězcem `x` nebo libovolným řetězcem v n-tici `x`. V opačném případě vrátí hodnotu `False` (viz též metoda `str.startswith()`).
`s.expandtabs(počet)`	Vrátí kopii řetězce `s`, která má místo tabulátorů mezery. Každý tabulátor se nahradí 8 mezerami nebo zadaným počtem mezer.
`s.find(t, začátek, konec)`	Vrátí nejlevější pozici řetězce `t` v řetězci `s` (nebo v řezu *`začátek`*`:`*`konec`* řetězce `s`) nebo -1, pokud nebyl nalezen. K vyhledání nejpravějšího řetězce se používá metoda `str.rfind()` (viz též metoda `str.index()`).
s.format(...)	Vrátí kopii řetězce `s` naformátovanou podle zadaných argumentů. Této metodě a jejím argumentům se budeme věnovat v následující podčásti.
s.index(t, start, end)	Vrátí nejlevější pozici řetězce `t` v řetězci `s` (nebo v řezu *`začátek`*`:`*`konec`* řetězce `s`) nebo vyvolá výjimku `ValueError`, pokud nebyl nalezen. K vyhledání nejpravějšího řetězce se používá metoda `str.rindex()` (viz též metoda `str.find()`).
s.isalnum()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v `s` alfanumerický.
s.isalpha()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v `s` abecední.
s.isdecimal()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v `s` číslicí o základu 10 ze znakové sady Unicode.
s.isdigit()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v `s` číslicí ze znakové sady ASCII.
s.isidentifier()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a představuje-li platný identifikátor.

typ `bytes` ➤ 286

Kódování znaků ➤ 95

`str.format()` ➤ 83

Identifikátory a klíčová slova ➤ 58

Syntaxe	Popis
s.islower()	Vrátí hodnotu `True`, má-li řetězec `s` alespoň jeden znak, který může mít podobu malého písmene, a všechny znaky, které mohou být malými písmeny, malými písmeny skutečně jsou (viz též metoda `str.isupper()`).
s.isnumeric()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v řetězci `s` číselným znakem ze znakové sady Unicode ve formě číslice nebo zlomku.
s.isprintable()	Vrátí hodnotu `True`, je-li řetězec `s` prázdný nebo je-li každý znak v `s` považován za tisknutelný, což zahrnuje mezeru, ale nezahrnuje znak nového řádku.
s.isspace()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a je-li každý znak v `s` bílým znakem.
s.istitle()	Vrátí hodnotu `True`, je-li řetězec `s` neprázdný a splňuje-li podmínky nadpisu (viz metoda `str.title()`).
s.isupper()	Vrátí hodnotu `True`, má-li řetězec `s` alespoň jeden znak, který může mít podobu velkého písmene, a všechny znaky, které mohou být velkými písmeny, velkými písmeny skutečně jsou (viz též metoda `str.islower()`).
s.join(seq)	Vrátí zřetězení všech prvků v posloupnosti *seq*, mezi které vloží řetězec `s` (který může být prázdný).
s.ljust(délka, znak)	Vrátí kopii řetězce `s` zarovnaného směrem doleva v řetězci zadané délky, který vyplní mezerami nebo volitelným argumentem znak (což je řetězec délky 1). Pro zarovnání doprava se používá metoda `str.rjust()` a pro zarovnání na střed metoda `str.center()` (viz též metoda `str.format()`).
s.lower()	Vrátí kopii řetězce `s` převedeného na malá písmena.
s.maketrans()	Doprovodná metoda metody `str.translate()`.
s.partition(t)	Vrátí n-tici tří řetězců: část řetězce `s` před nejlevějším výskytem řetězce `t`, řetězec `t` a část za řetězcem `t`.
s.replace(t, u, n)	Vrátí kopii řetězce `s`, v níž je každý výskyt (nebo maximálně `n` výskytů) řetězce `t` nahrazen řetězcem `u`.
s.split(t, n)	Vrátí seznam řetězců rozdělených nejvýše `n`-krát podle řetězce `t`. Není-li `n` zadáno, pak dělení probíhá v maximálním možné míře. Není-li zadáno `t`, pak se řetězec rozdělí podle mezer. Pro dělení zprava se používá metoda `str.rsplit()` – výsledek je odlišný pouze tehdy, je-li zadáno `n` a je-li jeho hodnota menší než maximální počet možných dělení.
s.splitlines(f)	Vrátí seznam řádků, který je výsledkem rozdělení řetězce s podle oddělovačů řádků. Nemá-li `f` hodnotu `True`, pak se oddělovače řádků odstraní.

Syntaxe	Popis
s.startswith(x, začátek, konec)	Vrátí hodnotu True, pokud řetězec s (nebo jeho řez *začátek*:*konec*) začíná řetězcem x nebo některým z řetězců v n-tici x.
s.strip(znaky)	Vrátí kopii řetězce s bez počátečního a koncového bílého znaku (nebo bez znaků v řetězci *znaky*).
s.swapcase()	Vrátí kopii řetězce s, která má místo malých písmen písmena velká a místo velkých písmen malá.
s.title()	Vrátí kopii řetězce s, jejíž první znak každého slova obsahuje velké písmeno a všechna ostatní písmena jsou malá (viz str.istitle()).
s.translate()	Doprovodná metoda metody str.maketrans() (podrobnosti viz následující text).
s.upper()	Vrátí kopii řetězce s převedeného na malá písmena (viz str.lower()).
s.zfill(délka)	Vrátí kopii řetězce s, která je na začátku vyplněna nulami, je-li kratší než zadaná délka.

Kterou vyhledávací metodu použijeme, je čistě záležitostí chuti a aktuálních okolností, i když pokud hledáme více indexových pozic, pak s metodou str.index() budeme mít obvykle čistší kód, což demonstrují následující dvě ekvivalentní funkce:

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    try:
        i = line.index(opener)
        start = i + len(opener)
        j = line.index(closer, start)
        return line[start:j]
    except ValueError:
        return None
```

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    i = line.find(opener)
    if i != -1:
        start = i + len(opener)
        j = line.find(closer, start)
        if j != -1:
            return line[start:j]
    return None
```

Obě verze funkce extract_from_tag() se chovají naprosto stejně. Například volání extract_from_tag("red", "jaká nádherná <red>růže</red>") vrátí řetězec „růže". Verze vlevo s ošetřováním výjimek odděluje kód, který dělá to, co chceme, od kódu, který ošetřuje chyby, kdežto verze vpravo promíchává vlastní zpracování s ošetřením chyb.

Všechny metody str.count(), str.endswith(), str.find(), str.rfind(), str.index(), str.rindex() a str.startswith() přijímají dva volitelné argumenty: počáteční a koncovou pozici. Jejich použití si ukážeme na následujících ekvivalentních příkazech (s je nějaký řetězec):

```
s.count("m", 6) == s[6:].count("m")
s.count("m", 5, -3) == s[5:-3].count("m")
```

Jak můžeme vidět, řetězcové metody, jež přijímají počáteční a koncové indexy, operují na řezu řetězce stanoveném těmito indexy.

Nyní se podíváme na další dva úryvky kódu, na nichž si objasníme chování metody str.partition() – i když v tomto příkladu použijeme metodu str.rpartition():

```
result = s.rpartition("/")
```

```
i = s.rfind("/")
if i == -1:
    result = "", "", s
else:
    result = s[:i], s[i], s[i + 1:]
```

Úryvky kódu na levé i pravé straně nejsou úplně stejné, protože kód vpravo navíc vytváří novou proměnnou i. Všimněte si, že můžeme přímo přiřazovat n-tice a že v obou případech hledáme nejpravější výskyt lomítka (/). Pokud řetězec s obsahuje "/usr/local/bin/firefox", pak oba úryvky vytvoří stejný výsledek: ('/usr/local/bin', '/', 'firefox').

Metodu str.endswith() (a str.startswith()) můžeme použít s jediným řetězcovým argumentem (např. s.startswith("From:")) nebo s n-ticí řetězců. Zde je příkaz, který pomocí metod str.endswith() a str.lower() otestuje, zda se jedná o soubor JPEG:

```
if filename.lower().endswith((".jpg", ".jpeg")):
    print(filename, " je obrázek JPEG")
```

Metody s názvem ve tvaru is*() (např. isalpha() nebo isspace()) vracejí hodnotu True, pokud řetězec, na kterém jsou zavolány, obsahuje nejméně jeden znak a pokud každý znak v tomto řetězci splňuje určitá kritéria:

```
>>> "917.5".isdigit(), "".isdigit(), "-2".isdigit(), "203".isdigit()
(False, False, False, True)
```

Metody is*() fungují na bázi klasifikace znaků ve znakové sadě Unicode, takže například volání metody str.isdigit() na řetězcích "\N{circled digit two}03" a "②03" vrátí v obou případech hodnotu True. Z tohoto důvodu nemůžeme předpokládat, že hodnota True vrácená metodou isdigit() automaticky znamená, že lze daný řetězec převést na celé číslo.

Když přijímáme řetězce z externích zdrojů (jiné programy, soubory, síťová připojení a zejména vstup od uživatelů), mohou tyto řetězce obsahovat nechtěné úvodní nebo koncové bílé místo. Úvodní bílé

místo odstraníme metodou `str.lstrip()`, koncové bílé místo metodou `str.rstrip()` a obě metodou `str.strip()`. Těmto metodám můžeme též předat jako argument řetězec, přičemž se odstraní každý výskyt všech znaků uvedených v zadaném řetězci:

```
>>> s = "\t neparkovat "
>>> s.lstrip(), s.rstrip(), s.strip()
(' neparkovat ', '\t neparkovat', 'neparkovat')
>>> "<[bez závorek]>".strip("[](){}<>")
'bez závorek'
```

příklad csv2html.py ➤ 100

Pomoc metody `str.replace()` můžeme nahrazovat řetězce uvnitř řetězců. Tato metoda přijímá dva řetězcové argumenty a vrací kopii řetězce, na kterém je zavolána. V této kopii jsou všechny výskyty prvního řetězce nahrazeny druhým. Je-li druhý argument prázdný řetězec, pak se všechny výskyty prvního řetězce vymažou. Na ukázky použití metody `str.replace()` a některých dalších řetězcových metod se podíváme v části Příklady v příkladu `csv2html.py` (na konci této lekce).

Častým požadavkem je rozdělení řetězce na seznam řetězců. Představte si, že máme textový soubor s daty obsahující na každém řádku jeden záznam, jehož pole jsou oddělena hvězdičkou. V této situaci můžeme využít metodu `str.split()`, které předáme jako první argument řetězec, který se má rozdělit, a dále volitelný druhý argument představující maximální počet dělení. Pokud druhý argument neuvedeme, provede se maximální možný počet dělení. Zde je příklad:

```
>>> record = "Lev N. Tolstoj*28.8.1828*20.11.1910"
>>> fields = record.split("*")
>>> fields
['Lev N. Tolstoj', '28.8.1828', '20.11.1910']
```

Nyní můžeme znovu aplikovat metodu `str.split()` na data narození a úmrtí a vypočítat počet roků, kterých se Lev N. Tolstoj dožil (plus minus jeden rok):

```
>>> born = fields[1].split(".")
>>> born
['28', '8', '1828']
>>> died = fields[2].split(".")
>>> print("dožil se asi", int(died[2]) - int(born[2]), "let")
dožil se asi 82 let
```

Kromě funkce `int()`, kterou jsme použili pro převod let z řetězců na celá čísla, je uvedený úryvek kódu poměrně jednoduchý. K rokům se můžeme dostat také přímo ze seznamu `fields` (např. `year_born = int(fields[1].split(".")[2])`).

V tabulce 2.7 jsme u metod `str.maketrans()` a `str.translate()` neuvedli žádný popis. Metoda `str.maketrans()` se používá ke tvorbě překladové tabulky, která mapuje znaky na znaky. Tato metoda přijímá jeden, dva nebo tři argumenty. My si zde ale ukážeme pouze nejjednodušší volání (se dvěma argumenty), kdy je první argument řetězec obsahující znaky, z nichž se má překládat, a druhý argument řetězec obsahující znaky, na které se má překlad provést.

Oba řetězcové argumenty musejí mít stejnou délku. Metoda `str.translate()` přijímá jako argument překladovou tabulku a vrací kopii jejího řetězce se znaky přeloženými podle překladové tabulky. Zde je příklad přeložení řetězců obsahujících bengálské číslice na anglické číslice:

```
table = "".maketrans("\N{bengali digit zero}"
    "\N{bengali digit one}\N{bengali digit two}"
    "\N{bengali digit three}\N{bengali digit four}"
    "\N{bengali digit five}\N{bengali digit six}"
    "\N{bengali digit seven}\N{bengali digit eight}"
    "\N{bengali digit nine}", "0123456789")
print("20749".translate(table))              # vypíše: 20749
print("\N{bengali digit two}07\N{bengali digit four}"
    "\N{bengali digit nine}".translate(table))  # vypíše: 20749
```

Všimněte si, že jsme při volání metody `str.maketrans()` a při druhém volání funkce `print()` využili spojování řetězcových literálů Pythonu, díky kterému jsme mohli rozprostřít řetězce na více řádků bez nutnosti potlačení znaků nového řádku nebo explicitního zřetězení.

Metodu `str.maketrans()` jsme klidně zavolali na prázdném řetězci, protože vůbec nezáleží na tom, na jakém řetězci ji zavoláme. Tato metoda totiž jen zpracuje své argumenty a vrátí překladovou tabulku. Metody `str.maketrans()` a `str.translate()` lze použít také k vymazání znaků. Stačí metodě `str.maketrans()` předat jako třetí argument řetězec obsahující nechtěné znaky. Pokud potřebujeme sofistikovanější překlady znaků, pak bychom mohli vytvořit vlastní kodek. Více informací na toto téma naleznete v dokumentaci k modulu `codecs` (viz *http://docs.python.org/library/codecs.html*).

Python nabízí ještě několik dalších knihovních modulů, které poskytují funkce související s řetězci. Již jsme se stručně zmínili o modulu `unicodedata`, jehož použití si ukážeme v následující podčásti. Mezi další moduly, které stojí za povšimnutí, patří modul `difflib`, který lze použít k zobrazení rozdílů mezi soubory nebo řetězci, třída `io.StringIO` modulu `io`, která umožňuje čtení a zápis řetězců jako by se jednalo o soubory, a modul `textwrap`, který nabízí funkce pro zabalování a vyplňování řetězců. K dispozici je též modul `string`, který obsahuje několik užitečných konstant, jako jsou `ascii_letters` a `ascii_lowercase`. S příklady použití některých z těchto modulů se setkáme v lekci 5. Kromě toho poskytuje Python v modulu `re` skvělou podporu regulárních výrazů (viz Lekce 13).

Formátování řetězců metodou str.format()

Metoda `str.format()` nabízí velmi flexibilní a výkonný prostředek pro vytváření řetězců. Pro jednoduché případy se sice používá snadno, ale pro složité formátování se musíme naučit formátovací syntaxi, kterou tato metoda vyžaduje.

Metoda `str.format()` vrací nový řetězec, který má místo nahrazovacích polí vhodně naformátované argumenty:

```
>>> "Román '{0}' vyšel v roce {1}".format("Těžké časy", 1854)
"Román 'Těžké časy' vyšel v roce 1854"
```

Každé nahrazovací pole je označeno názvem pole ve složených závorkách. Má-li název pole podobu celého čísla, pak se jedná o index jednoho z argumentů předaných metodě `str.format()`. V tomto případě bylo tedy pole s názvem `0` nahrazeno prvním argumentem a pole s názvem `1` druhým argumentem.

Pokud potřebujeme do formátovacího řetězce umístit složené závorky, musíme je zdvojit. Zde je příklad:

```
>>> "{{{0}}} {1} ;-}}".format("Já jsem v závorkách", "A já ne")
'{Já jsem v závorkách} A já ne ;-}'
```

Pokud se pokusíme zřetězit řetězec a číslo, Python tiše vyvolá výjimku `TypeError`. Požadovaného výsledku však můžeme snadno dosáhnout pomocí metody `str.format()`:

```
>>> "{0}{1} Kč".format("Dlužná částka činí ", 200)
'Dlužná částka činí 200 Kč'
```

Pomocí metody `str.format()` můžeme také spojovat řetězce (i když metoda `str.join()` je pro tento účel vhodnější):

```
>>> x = "tři"
>>> s ="{0} {1} {2}"
>>> s = s.format("Moje", x, "tečky")
>>> s
'Moje tři tečky'
```

Zde jsme použili několik řetězcových proměnných, avšak ve většině příkladů v této části budeme v souvislosti s metodou `str.format()` používat řetězcové literály. Mějte ale na paměti, že jakýkoliv příklad používající řetězcový literál by mohl naprosto stejným způsobem používat řetězcovou proměnnou.

Nahrazovací pole může mít kteroukoli z následujících obecných syntaxí:

```
{název_pole}
{název_pole!převod}
{název_pole:specifikace_formátu}
{název_pole!převod:specifikace_formátu}
```

Důležité je, že nahrazovací pole mohou *obsahovat* další nahrazovací pole. Vnořená nahrazovací pole nemohou mít žádné formátování. Jejich účelem je podpora odvozených formátovacích specifikací. Podrobněji se tuto situaci podíváme v rámci výkladu specifikace formátu. Nyní prostudujeme postupně každou z částí nahrazovacího pole, přičemž začneme názvem pole.

Název pole

Názvem pole může být buď celé číslo odpovídající jednomu z argumentů metody `str.format()`, nebo jméno jednoho z klíčovaných argumentů metod. Klíčované argumenty budeme probírat v lekci 4, nejedná se ale o nic složitého, a proto se pro úplnost podíváme na několik příkladů:

```
>>> "{who} je tento rok {age}".format(who="Janě", age=88)
'Janě je tento rok 88'
>>> "{who} bylo minulý týden {0}".format(12, who="chlapci")
'chlapci bylo minulý týden 12'
```

První příklad používá dva klíčované argumenty `who` a `age` a druhý příklad používá jeden poziční (jediný druh, který jsme dosud používali) a jeden klíčovaný argument. Všimněte si, že v seznamu argumentů jsou klíčované argumenty vždy uvedeny až za pozičními argumenty. Ve formátovacím řetězci můžeme samozřejmě využít kterýkoli z argumentů v libovolném pořadí.

Názvy polí se mohou odkazovat na datové typy představující kolekce – například tedy na seznamy. V takových případech můžeme uvést index (ne řez!) označující konkrétní prvek:

```
>>> stock = ["papír", "obálky", "zápisníky", "pera", "sponky"]
>>> "Na skladě máme {0[1]} a {0[2]}".format(stock)
'Na skladě máme obálky a zápisníky'
```

Číslice `0` označuje poziční argument, takže `{0[1]}` je druhý prvek argumentu `stock` a `{0[2]}` je třetí prvek argumentu `stock`.

Později se seznámíme se slovníky jazyka Python. Ty totiž uchovávají prvky spojující klíče s hodnotami a vzhledem k tomu, že je lze použít s metodou `str.format()`, ukážeme si zde rychlý příklad. Nic si z toho nedělejte, pokud vám to zatím nebude dávat smysl. Vše se vyjasní v lekci 3.

Typ `dict` ➤ 128

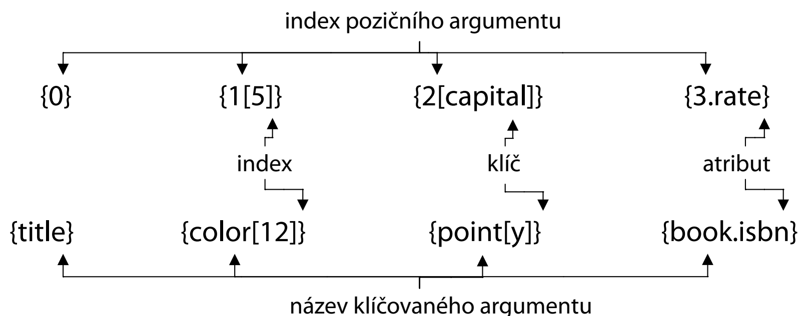
```
>>> d = dict(animal="slon", weight=12000)
>>> "Náš {0[animal]} váží {0[weight]} kg".format(d)
'Náš slon váží 12000 kg'
```

Stejně jako přistupujeme k prvkům seznamu a n-tice pomocí číselné pozice indexu, přistupujeme k prvkům slovníku pomocí klíče.

Můžeme také přistupovat k pojmenovaným atributům. Po importování modulů `math` a `sys` můžeme napsat následující kód:

```
>>> "math.pi=={0.pi} sys.maxunicode=={1.maxunicode}".format(math, sys)
'math.pi==3.14159265359 sys.maxunicode==65535'
```

Podtrženo sečteno: pomocí syntaxe pro název pole můžeme adresovat poziční i klíčované argumenty, které předáváme metodě `str.format()`. Jsou-li argumenty některého z datových typů představujících kolekce, jako jsou seznamy nebo slovníky, nebo pokud mají atributy, pak můžeme k části, kterou chceme, přistupovat pomocí závorkové (`[]`) nebo tečkové (`.`) notace. Tuto situaci zachycuje obrázek 2.5.



Obrázek 2.5: Ukázky specifikace formátu pro názvy polí

3.1

Od Pythonu 3.1 je možné názvy polí vynechat, přičemž Python za nás doplní čísla začínající od 0:

```
>>> "{} {} {}".format("Python", "umí", "počítat")
'Python umí počítat'
```

Pokud používáte Python 3.0, pak musíte ve výše uvedeném příkladu použít formátovací řetězec ve tvaru "{0} {1} {2}". Tato technika je výhodná pro formátování jednoho nebo dvou prvků, avšak postup, který si ukážeme vzápětí, je při práci s větším počtem prvků vhodnější a funguje skvěle i v Pythonu 3.0.

Ještě před uzavřením výkonu o názvech polí pro formátování řetězců je dobré zmínit poněkud odlišný způsob vkládání hodnot do formátovacího řetězce. K tomu je nutné sáhnout po pokročilé technice, kterou je vhodné si co nejdříve osvojit.

Rozbalení mapování ➤ 177

Lokální proměnné, které jsou aktuálně v oboru platnosti, jsou dostupné přes vestavěnou funkci locals(). Tato funkce vrací slovník, jehož klíče odpovídají názvům lokálních proměnných a jehož hodnoty se odkazují na hodnoty těchto proměnných. Nyní můžeme použít *rozbalení mapování* pro naládování tohoto slovníku do metody str.format(). K rozbalení mapování slouží operátor **, jehož aplikací na mapování (např. na slovník) vytvoříme seznam klíč-hodnota vhodný pro předání funkci:

```
>>> element = "Stříbro"
>>> number = 47
>>> "Prvek {number} je {element}".format(**locals())
'Prvek 47 je Stříbro'
```

Rozbalení parametrů ➤ 176

Tato syntaxe se může na první pohled jevit poněkud zvláštně, takže všichni programátoři v Perlu se budou cítit jako doma, ale ničeho se neobávejte – vše si vysvětlíme v lekci 4. Prozatím stačí vědět, že ve formátovacích řetězcích můžeme použít názvy proměnných a nechat Python, aby jejich hodnoty předal metodě str.format() prostým rozbalením slovníku vráceného funkcí locals() (nebo nějakým jiným slovníkem). Například výše uvedený příklad se slonem můžeme přepsat tak, aby měl hezčí formátovací řetězec s jednoduššími názvy polí:

```
>>> "Náš {animal} váží {weight} kg".format(**d)
'Náš slon váží 12000 kg'
```

Rozbalením slovníku do metody str.format() můžeme použít jeho klíče jako názvy polí. Díky tomu jsou formátovací řetězce mnohem srozumitelnější a také se snadněji udržují, protože nejsou závislé na pořadí argumentů. Je však třeba poznamenat, že pokud potřebujeme metodě str.format() předat více než jeden argument, pak můžeme rozbalení mapování použít pouze na poslední.

Převody

Desetinná čísla ➤ 69

Když jsme probírali čísla typu decimal.Decimal, řekli jsme si, že tato čísla se vypisují jedním ze dvou způsobů:

```
>>> decimal.Decimal("3.4084")
Decimal('3.4084')
>>> print(decimal.Decimal("3.4084"))
3.4084
```

eval()
➤ 334

První možnost, jak zobrazit číslo typu `decimal.Decimal`, je použít jeho reprezentační formu. Smyslem této formy je poskytovat řetězec, který by po interpretaci Pythonem opětovně vytvořil objekt, který reprezentuje. Programy napsané v Pythonu mohou vyhodnocovat úryvky kódu nebo celých programů napsaných v jazyku Python, takže tato schopnost může být v určitých situacích užitečná. Ne všechny objekty umí poskytovat reprodukční reprezentaci. V takovém případě vracejí řetězec uzavřený do lomených závorek. Například reprezentační forma modulu `sys` je řetězec "<module 'sys' (built-in)>".

Druhou možnost pro zobrazení čísla typu `decimal.Decimal` nabízí jeho řetězcová forma. Tato forma je určena pro lidské čtenáře, a proto je zaměřena na zobrazení něčeho, co je srozumitelné pro člověka. Pokud datový typ řetězcovou formu nemá a je vyžadován řetězec, tak Python použije reprezentační formu.

Vestavěné datové typy jazyka Python znají metodu `str.format()`, takže při předání této metodě ve formě argumentu vrátí vhodný řetězec pro své zobrazení. Jak uvidíme v lekci 6, přidání podpory pro metodu `str.format()` do našich vlastních datových typů je také velice jednoduché. Kromě toho je možné běžné chování datového typu přepsat a přinutit jej poskytovat buď jeho řetězcovou, nebo jeho reprezentační formu. To lze provést tak, že k danému poli přidáme převodní specifikátor. V současnosti jsou k dispozici tři převodní specifikátory: `s` pro vynucení řetězcové formy, `r` pro vynucení reprezentační formy a specifikátor `a` pro vynucení reprezentační formy obsahující pouze znaky z kódování ASCII. Zde je příklad:

```
>>> "{0} {0!s} {0!r} {0!a}".format(decimal.Decimal("93.4"))
"93.4 93.4 Decimal('93.4') Decimal('93.4')"
```

V tomto případě vytvoří řetězcová forma čísla typu `decimal.Decimal` stejný řetězec, jako je ten, který poskytuje pro metodu `str.format()`, což se je docela obvyklé. Dále v tomto příkladu není žádný rozdíl mezi obyčejnou reprezentační formou a reprezentační formou v kódování ASCII, protože obě používají pouze znaky ze znakové sady ASCII.

Zde je další příklad, který se tentokrát týká řetězce, jenž obsahuje název filmu "翻訳で失われる" uchovávaný v proměnné `movie`. Pokud tento řetězec vypíšeme příkazem `"{0}".format(movie)`, pak se vypíše beze změny. Pokud však nechceme znaky mimo kódování ASCII, pak můžeme použít buď příkaz `ascii(movie)`, nebo `"{0!a}".format(movie)`, protože oba příkazy vytvoří řetězec `'\u7ffb\u8a33\u3067\u5931\u308f\u308c\u308b'`.

Dosud jsme viděli, jak umístit hodnoty proměnných do formátovaného řetězce a jak si vynutit použití řetězcové nebo reprezentační formy. Nyní jsme tedy připraveni podívat se formátování samotných hodnot.

Specifikace formátu

Často je výchozí formátování celých čísel, čísel s pohyblivou řádovou čárkou a řetězců naprosto dostačující. K snadnému procvičení jemné kontroly formátování můžeme využít specifikace formátu. Pro snazší osvojení podrobností se budeme věnovat formátování jednotlivých typů samostatně. Obecnou syntaxi vztahující se ke všem typům uvádí tabulka 2.8.

Tabulka 2.8: Obecný tvar specifikace formátu

Specifikátor	Popis
:	
výplň	Libovolný znak kromě znaku `}`.
zarovnání	< vlevo, > vpravo, ^ na střed, = vyplnění mezi znaménkem a číslicemi u čísel.
znaménko	+ vynucený zápis znaménka, - znaménko jen v případě potřeby, " " mezera nebo – dle aktuální potřeby.
#	Před celé číslo umístí prefix `0b`, `0o` nebo `0x`.
0	Číslo se vyplní nulami do požadované šířky.
šířka	Minimální šířka pole.
,	Čárka se používá pro seskupování*.
.přesnost	Maximální šířka pole pro řetězce. V případě čísel s pohyblivou řádovou čárkou udává počet desetinných míst.
typ	Pro typ `int`: `b`, `c`, `d`, `n`, `o`, `x`, `X`; pro typ `float`: `e`, `E`, `f`, `g`, `G`, `n`, `%`.

U řetězců můžeme nastavovat výplňový znak, zarovnání uvnitř pole a minimální a maximální šířku pole.

Specifikace formátu řetězce začíná dvojtečkou (`:`), za kterou následuje volitelná dvojice znaků: výplňový znak (což nesmí být znak `}`) a zarovnávací znak (`<` pro zarovnání vlevo, `^` pro zarovnání na střed, `>` pro zarovnání vpravo). Pak je na řadě volitelné celé číslo udávající minimální šířku, za nímž může následovat maximální šířka, která se zapisuje jako tečka a celé číslo.

Všimněte si, že pokud zadáme výplňový znak, musíme zadat též zarovnání. Části se znaménkem a typem jsme vynechali, protože na řetězce nemají žádný vliv. Dvojtečka bez volitelných prvků je sice neškodná, ale naprosto zbytečná.

Podívejme se na několik příkladů:

```
>>> s = "Žhnoucí meč pravdy"
>>> "{0}".format(s)       # výchozí formátování
'Žhnoucí meč pravdy'
>>> "{0:25}".format(s)    # minimální šířka 25
'Žhnoucí meč pravdy       '
>>> "{0:>25}".format(s)   # zarovnání vpravo, minimální šířka 25
'       Žhnoucí meč pravdy'
>>> "{0:^25}".format(s)   # zarovnání na střed, minimální šířka 25
'   Žhnoucí meč pravdy    '
>>> "{0:-^25}".format(s)  # výplň -, zarovnání na střed, minimální šířka 25
'---Žhnoucí meč pravdy----'
>>> "{0:.<25}".format(s)  # výplň ., zarovnání vlevo, minimální šířka 25
'Žhnoucí meč pravdy.......'
>>> "{0:.10}".format(s)   # maximální šířka 10
'Žhnoucí me'
```

* Seskupování bylo zavedeno v Pythonu 3.1.

V předposledním příkladu jsme museli uvést zarovnání vlevo (i když se jedná o výchozí zarovnání). Pokud bychom značku `<` vynechali, měli bychom `:.25`, což ale znamená, že maximální šířka pole je 25 znaků.

Jak jsme si řekli již dříve, specifikace formátu může obsahovat nahrazovací pole. Díky tomu můžeme formát definovat dynamicky. Zde jsou například dva způsoby nastavení maximální šířky řetězce pomocí proměnné `maxwidth`:

```
>>> maxwidth = 12
>>> "{0}".format(s[:maxwidth])
'Žhnoucí meč '
>>> "{0:.{1}}".format(s, maxwidth)
'Žhnoucí meč '
```

V prvním případě používáme standardní řezání řetězce, zatímco ve druhém máme vnitřní nahrazovací pole.

U celých čísel nám specifikace formátu umožňuje nastavovat výplňový znak, zarovnání uvnitř pole, znaménko, zda se má použít jiný oddělovač pro seskupení číslic (od Pythonu 3.1), minimální šířku pole a základ pro zobrazení čísla.

Specifikace formátu celého čísla začíná dvojtečkou, za níž můžeme uvést volitelnou dvojici znaků: výplňový znak (což nesmí být znak `}`) a zarovnávací znak (`<` pro zarovnání vlevo, `^` pro zarovnání na střed, `>` pro zarovnání vpravo a `=` pro výplň mezi znaménkem a číslem). Pak je na řadě volitelné celé číslo udávající minimální šířku, za nímž může následovat maximální šířka, která se zapisuje jako tečka a celé číslo. Dále zde máme volitelný znak znaménka: `+` způsobí vynucený výpis znaménka, `-` vypíše znaménko pouze pro záporná čísla a mezera vypíše mezeru pro kladná čísla a znaménko `-` pro záporná čísla. Pak následuje volitelná minimální šířka celého čísla, před níž může být umístěn znak `#` způsobující vypsání prefixu označujícího základ čísla (pro binární, osmičková a šestnáctková čísla) a znak `0` pro vyplnění znakem `0`. Počínaje Pythonem 3.1 můžeme uvést volitelnou čárku, která zapříčiní, že se cifry čísla seskupí do skupin po třech cifrách, při čemž se každá skupina oddělí čárkou. Pokud chceme mít výstup v jiném než desítkovém základu, musíme přidat znak pro typ: `b` pro binární, `o` pro osmičkový, `x` pro šestnáctkový s malými písmeny a `X` pro šestnáctkový s velkými písmeny (a pro úplnost lze uvést také `d` pro desítková celá čísla). Kromě toho jsou k dispozici ještě další dva znaky pro typ: `c`, což znamená, že se má použít znak ze znakové sady Unicode odpovídající celému číslu, a `n`, které vypíše číslo podle aktuálního národního prostředí (všimněte si, že při použití znaku `n` již nemá smysl používat čárku).

Vyplnění nulami můžeme provést dvěma různými způsoby:

```
>>> "{0:0=12}".format(8749203)  # vyplnění nulami, minimální šířka 12
'000008749203'
>>> "{0:0=12}".format(-8749203) # vyplnění nulami, minimální šířka 12
'-00008749203'
>>> "{0:012}".format(8749203)   # doplní nulami na minimální šířku 12
'000008749203'
>>> "{0:012}".format(-8749203)  # doplní nulami na minimální šířku 12
'-00008749203'
```

V prvních dvou příkladech je výplňový znak 0 a k vyplnění dochází mezi znaménkem a samotným číslem (=). Druhé dva příklady definují minimální šířku 12 a doplnění nulami.

Zde je několik příkladů zarovnání:

```
>>> "{0:*<15}".format(18340427)  # vyplnění *, zarovnání vlevo, min. šířka 15
'18340427*******'
>>> "{0:*>15}".format(18340427)  # vyplnění *, zarovnání vpravo, min. šířka 15
'*******18340427'
>>> "{0:*^15}".format(18340427)  # vyplnění *, zarovnání na střed, min. šířka 15
'***18340427****'
>>> "{0:*^15}".format(-18340427) # vyplnění *, zarovnání na střed, min. šířka 15
'***-18340427***'
Zde je několik příkladů, které ukazují výsledek aplikace znaménkových znaků:
>>> "[{0: }] [{1: }]".format(539802, -539802) # mezera nebo znak -
'[ 539802] [-539802]'
>>> "[{0:+}] [{1:+}]".format(539802, -539802) # vynucené vypsání znaménka
'[+539802] [-539802]'
>>> "[{0:-}] [{1:-}]".format(539802, -539802) # vypsání znaménka -, je-li třeba
'[539802] [-539802]'
```

A zde jsou dva příklady, které používají některé ze znaků pro typ:

```
>>> "{0:b} {0:o} {0:x} {0:X}".format(14613198)
'110111101111101011001110 67575316 deface DEFACE'
>>> "{0:#b} {0:#o} {0:#x} {0:#X}".format(14613198)
'0b110111101111101011001110 0o67575316 0xdeface 0XDEFACE'
```

V případě celých čísel nemůžeme stanovit maximální šířku pole. To je dáno tím, že by jinak bylo nutné osekat cifry, čímž by výpis čísla pozbyl smyslu.

3.1

Pokud používáme Python 3.1 a ve specifikaci formátu uvedeme čárku, pak celé číslo použije čárky pro seskupení:

```
>>> "{0:,} {0:*>13,}".format(int(2.39432185e6))
'2,394,321 ****2,394,321'
```

Na obě pole jsme aplikovali seskupení, přičemž druhé pole je doplněno hvězdičkami, zarovnáno vpravo a má nastavenou minimální šířku 13 znaků. Jedná se o obvyklý způsob zápisu v řadě vědeckých a finančních programů, který však nebere v potaz aktuální národní prostředí. Například v České republice se tisíce obvykle oddělují mezerou a jako oddělovač desetinné části se používá čárka.

Posledním formátovacím znakem dostupným pro celá čísla (a také pro čísla s pohyblivou řádovou čárkou) je znak n. Ten má při zadání celého čísla resp. čísla, s pohyblivou řádovou čárkou, stejný účinek jako znak d, resp. g. Jeho zvláštností je ovšem to, že respektuje národní prostředí, takže ve výstupu použije oddělovač desetinné části a seskupení pro aktuální národní prostředí. Výchozí národní prostředí označované jako C definuje pro oddělovač desetinné části tečku a pro oddělovač seskupení

prázdný řetězec. Národní prostředí uživatele můžeme respektovat tak, že na začátku svých programů napíšeme následující dva řádky kódu:*

```
import locale
locale.setlocale(locale.LC_ALL, "")
```

Předáním prázdného řetězce jako národního prostředí říkáme Pythonu, aby se automaticky pokusil zjistit národní prostředí uživatele (např. prozkoumáním proměnné prostředí `LANG`) s tím, že se v případě neúspěchu vrátí k národnímu prostředí `C`. Zde je několik příkladů, které ukazují účinky různých národních prostředí na formátování celých čísel a čísel s pohyblivou řádovou čárkou:

```
x, y = (1234567890, 1234.56)
locale.setlocale(locale.LC_ALL, "C")
c = "{0:n} {1:n}".format(x, y)    # c == '1234567890 1234.56'
locale.setlocale(locale.LC_ALL, "")
cz = "{0:n} {1:n}".format(x, y)   # cz == '1 234 567 890 1 234,56'
```

I když volba `n` je pro celá čísla velice užitečná, v případě čísel s pohyblivou řádovou čárkou jsou její možnosti omezené, protože jakmile jsou tato čísla příliš velká, zobrazují se pomocí exponenciální formy.

U čísel s pohyblivou řádovou čárkou nám specifikace formátu poskytuje kontrolu nad výplňovým znakem, zarovnáním uvnitř pole, znaménkem, dále nad tím, zda se má použít oddělovač skupin cifer bez ohledu na národní prostřední (od Pythonu 3.1), nad minimální šířkou pole, počtem desetinných míst a nad tím, zda se má číslo prezentovat ve standardní či exponenciální formě nebo jako procentní podíl.

Specifikace formátu pro čísla s pohyblivou řádovou čárkou je stejná jako pro celá čísla, ovšem až na dvě odlišnosti na konci. Za volitelnou minimální šířkou (od Pythonu 3.1 za volitelnou čárkou pro seskupení) můžeme zapsáním tečky a celého čísla stanovit počet číslic za oddělovačem desetinných míst. Na konec můžeme také přidat znak pro typ: `e` pro exponenciální formu s malým písmenem „e", `E` pro exponenciální formu s velkým písmenem „E", `f` pro standardní formu čísla s pohyblivou řádovou čárkou, `g` pro „obecnou" formu (ta je stejná jako v případě `f`, pokud ovšem číslo není příliš velké, což způsobí, že se použije `e`) a `G`, což je stejné jako `g`, ale používá buď `f`, nebo `E`. Kromě toho máme k dispozici také znak `%`, který způsobí to, že se číslo vynásobí 100 a výsledek se zobrazí ve formátu f s připojeným symbolem `%`.

3.x

Zde je několik příkladů, které ukazují exponenciální a standardní formu:

```
>>> amount = (10 ** 3) * math.pi
>>> "[{0:12.2e}] [{0:12.2f}]".format(amount)
'[    3.14e+03] [     3141.59]'
>>> "[{0:*>12.2e}] [{0:*>12.2f}]".format(amount)
'[****3.14e+03] [*****3141.59]'
>>> "[{0:*>+12.2e}] [{0:*>+12.2f}]".format(amount)
'[***+3.14e+03] [****+3141.59]'
```

* Ve vícevláknových programech je nejvhodnější volat metodu `locale.setlocale()` pouze jednou při spouštění programu, než se spustí jakákoli další vlákna, neboť tato metoda obvykle není ve vícevláknovém prostředí bezpečná.

V prvním příkladu máme minimální šířku 12 znaků a 2 číslice na desetinných místech. Druhý příklad staví na prvním a přidává výplňový znak *. Pokud používáme výplňový znak, musíme uvést také zarovnávací znak. Proto jsme stanovili zarovnání vpravo (i když se jedná o výchozí zarovnání pro čísla). Třetí příklad staví na předchozích dvou a přidává znak +, kterým si vynutíme výpis znaménka.

3.1

V Pythonu 3.0 nepracuje metoda `str.format()` s čísly typu `decimal.Decimal` jako s čísly, ale jako s řetězci. Kvůli tomu je docela těžké vytvořit pěkně naformátovaný výstup. Počínaje Pythonem 3.1 lze čísla typu `decimal.Decimal` formátovat jako typ `float` včetně podpory pro volbu čárka (`,`) vytvářející čárkou oddělené skupiny. Zde je příklad (název pole jsme vynechali, protože jej v Pythonu 3.1 nepotřebujeme):

```
>>> "{:,.6f}".format(decimal.Decimal("1234567890.1234567890"))
'1,234,567,890.123457'
```

Pokud formátovací znak `f` vynecháme (nebo použijeme formátovací znak `g`), pak se číslo naformátuje jako `'1.23457E+9'`.

Python 3.0 neposkytuje žádnou přímou podporu pro formátování komplexních čísel – ta byla přidána až v Pythonu 3.1. To však můžeme snadno vyřešit naformátováním reálné a imaginární části jako samostatná čísla s pohyblivou řádovou čárkou:

```
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917+1.2042j)
'4.759+1.204j'
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917-1.2042j)
'4.759-1.204j'
```

Ke každému atributu komplexního čísla přistupujeme samostatně a formátujeme jej jako číslo s pohyblivou řádovou čárkou, v tomto případě se třemi číslicemi na desetinných místech. Vynutili jsme si také vypisování znaménka pro imaginární část, přičemž znak „j" musíme přidat sami.

3.1

Python 3.1 podporuje formátování komplexních čísel pomocí stejné syntaxe, která se používá pro čísla typu `float`:

```
>>> "{:,.4f}".format(3.59284e6-8.984327843e6j)
'3,592,840.0000-8,984,327.8430j'
```

Jediná malinká nevýhoda tohoto přístupu tkví v tom, že stejné formátování se aplikuje na reálnou i imaginární část. Pokud potřebujeme každou z těchto částí naformátovat odlišným způsobem, můžeme vždy sáhnout po technice pro Python 3.0, v níž přistupujeme k atributům komplexního čísla samostatně.

Příklad: print_unicode.py

V předchozích podčástech jsme si podrobně rozebrali specifikaci formátu metody `str.format()` a viděli jsme řadu úryvků kódu, na kterých jsme si ukázali nejrůznější aspekty formátování řetězců. Zde se podíváme na malý, přesto však užitečný příklad, který využívá metodu `str.format()` tak, že si můžeme prohlédnout specifikace formátu v realistickém kontextu. V tomto příkladu používáme

také některé z řetězcových metod, s nimiž jsme se seznámili v předchozí části, a dále si představíme funkci z modulu `unicodedata`*.

Tento program má pouze 25 řádků spustitelného kódu. Importuje dva moduly, `sys` a `unicodedata`, a definuje jednu vlastní funkci `print_unicode_table()`. Začneme pohledem na ukázkový běh programu, abychom si ukázali, co program dělá, a poté se podíváme na kód na konci programu, kde začíná vlastní zpracování, a nakonec se zaměříme na námi definovanou funkci.

```
print_unicode.py spoked
desítk.  hexa.  znak                   název
-------  -----  ----  ----------------------------------------
  10018   2722   ✢   Four Teardrop-Spoked Asterisk
  10019   2723   ✣   Four Balloon-Spoked Asterisk
  10020   2724   ✤   Heavy Four Balloon-Spoked Asterisk
  10021   2725   ✥   Four Club-Spoked Asterisk
  10035   2733   ✳   Eight Spoked Asterisk
  10043   273B   ✻   Teardrop-Spoked Asterisk
  10044   273C   ✼   Open Centre Teardrop-Spoked Asterisk
  10045   273D   ✽   Heavy Teardrop-Spoked Asterisk
  10051   2743   ❃   Heavy Teardrop-Spoked Pinwheel Asterisk
  10057   2749   ❉   Balloon-Spoked Asterisk
  10058   274A   ❊   Eight Teardrop-Spoked Propeller Asterisk
  10059   274B   ❋   Heavy Eight Teardrop-Spoked Propeller Asterisk
```

Pokud program spustíme bez argumentů, obdržíme tabulku všech znaků znakové sady Unicode, počínaje znakem mezery až po znak s nejvyšší dostupným kódovým bodem. Zadáme-li nějaký argument, jako ve výše uvedeném případě, vypíšou se pouze ty řádky v tabulce, v nichž název znaku znakové sady Unicode převedený na malá písmena obsahuje zadaný argument.

```
word = None
if len(sys.argv) > 1:
    if sys.argv[1] in ("-h", "--help"):
        print("použití: {0} [řetězec]".format(sys.argv[0]))
        word = 0
    else:
        word = sys.argv[1].lower()
if word != 0:
    print_unicode_table(word)
```

Po importování modulů a vytvoření funkce `print_unicode_table()` se provádění programu dostane do výše uvedeného místa. Začneme předpokladem, že uživatel na příkazovém řádku nezadal slovo, které se má vyhledat. Pokud je na příkazovém řádku zadán argument `-h` nebo `--help`, vypíše-

* V tomto programu předpokládáme, že konzola používá kódování UTF-8 znakové sady Unicode. Naneštěstí konzola systému Windows má chabou podporu pro kódování UTF-8. Tento problém obcházíme zvláštní verzí příkladu s názvem `print_unicode_uni.py`, která svůj výstup zapisuje do souboru, který lze poté otevřít v nějakém editoru, který umí pracovat s kódováním UTF-8, jako je kupříkladu editor IDLE.

Lekce 7 (Práce se soubory) ➢ 280

me informace o použití programu a nastavíme `word` na hodnotu `0`, což signalizuje, že jsme skončili. V opačném případě nastavíme `word` na kopii argumentu zadaného uživatelem, který jsme převedli na malá písmena. Pokud proměnná `word` nemá hodnotu `0`, vypíšeme tabulku.

Při vypisování informací o použití programu používáme specifikaci formátu, která obsahuje jen formátovací název, kterým je v tomto případě číslo pozice argumentu. Tento řádek bychom mohli zapsat také takto:

```
print("použití: {0[0]} [řetězec]".format(sys.argv))
```

V tomto případě je první `0` indexem argumentu, který chceme použít, a `[0]` indexem *uvnitř* tohoto argumentu, což může být, protože `sys.argv` je seznam.

```
def print_unicode_table(word):
    print("desítk.  hexa.  znak  {0:^40}".format("název"))
    print("-------  -----  ----  {0:-<40}".format(""))

    code = ord(" ")
    end = min(0xD800, sys.maxunicode)

    while code < end:
        c = chr(code)
        name = unicodedata.name(c, "*** neznámé ***")
        if word is None or word in name.lower():
            print("{0:7} {0:5X} {0:^3c} {1}".format(code, name.title()))
        code += 1
```

Kvůli lepší čitelnosti jsme použili několik prázdných řádků. První dva řádky těla funkce vypíšou záhlaví tabulky. První volání metody `str.format()` vypíše text „název" zarovnaný na střed pole širokého 40 znaků, zatímco druhé vypíše s použitím výplňového znaku „-" prázdný řetězec v poli širokém 40 znaků. (Pokud stanovíme výplňový znak, musíme uvést zarovnání.) Druhý řádek bychom mohli zapsat také tímto způsobem:

```
print("-------  -----  ---- {0}".format("-" * 40))
```

Zde jsme pomocí operátoru řetězcové replikace (`*`) vytvořili vhodný řetězec, který jsme prostě vložili do formátovacího řetězce. Třetí možností by bylo jednoduše zapsat 40 pomlček a použít literálový řetězec.

Kódové body znakové sady Unicode máme uloženy v proměnné code, kterou inicializujeme kódovým bodem pro mezeru (`0x20`). Koncovou proměnnou nastavujeme na nejvyšší dostupný kódový bod znakové sady Unicode, který se liší podle toho, zda byl Python zkompilován tak, aby používal formát UCS-2, nebo UCS-4.

Uvnitř cyklu `while` vezmeme pomocí funkce `chr()` znak ze znakové sady Unicode, který odpovídá kódovému bodu. Funkce `unicodedata.name()` vrací název zadaného znaku ze znakové sady Unicode. Jejím volitelným druhým argumentem je název, který se má použít, pokud pro daný znak není definován žádný název.

Pokud uživatel nezadal nějaké slovo (`word` je `None`) nebo pokud jej zadal a nachází se v kopii názvu znaku Unicode převedeného na malá písmena, pak daný řádek vypíšeme.

I když proměnnou `code` předáváme metodě `str.format()` pouze jednou, používá se při formátování řetězce hned třikrát. Poprvé pro vypsání kódu ve formě celého čísla v poli širokém 7 znaků (výplňový znak je ve výchozím stavu mezera, takže jej nemusíme uvádět), podruhé pro vypsání kódu jako šestnáctkového čísla s velkými písmeny v poli širokém 5 znaků a potřetí pro vypsání znaku Unicode, který odpovídá aktuálnímu kódu (pro tento účel používám formátovací specifikátor „c“) a který zarovnáme na střed do pole s minimální šířkou 3 znaky. Všimněte si, že v první specifikaci formátu nemusíme zadávat typ „d“, což je dáno tím, že se jedná o výchozí typ pro celočíselné argumenty. Druhý argument je název znaku znakové sady Unicode vypsaný způsobem, kdy je první písmeno každého slova velké a všechna ostatní písmena jsou malá.

Nyní již známe všestranné použití metody `str.format()`, a proto ji budeme ve zbývající části této knihy bohatě využívat.

Kódování znaků

Počítače dokážou v podstatě uchovávat pouze bajty, což jsou 8bitové hodnoty, které mají bez znaménka rozsah od `0x00` do `0xFF`. Každý znak tak musí být nějakým způsobem reprezentován pomocí bajtů. V dřevních dobách počítačů vymysleli průkopníci kódovací schémata, která přiřazovala určitý znak určitému bajtu. Například v kódování ASCII je znak A reprezentován hodnotou `0x41`, B hodnotou `0x42` a tak dále. Ve Spojených státech a v západní Evropě se často používalo kódování Latin-1. Jeho znaky z rozsahu 0x20-0x7E jsou stejné jako odpovídající znaky v 7bitovém kódování ASCII, přičemž znaky z rozsahu 0xA0-0xFF se používaly pro písmena s diakritikou a další symboly nezbytné pro ty, kteří používali pro písmena latinku, ale ne anglickou. V průběhu let byla vymyšlena řada dalších kódování, z nichž mnohá se používají dodnes. Nicméně vývoj se pro spoustu z nich zastavil ve prospěch kódování Unicode.

Existence všech těchto odlišných kódování se ukázala jako velice nevhodná, zejména pak při psaní internacionalizovaného softwaru. Jedno řešení, které bylo již téměř celosvětově přijato, je kódování Unicode. Toto kódování přiřazuje každý znak celému číslu (označovanému řečí Unicode jako *kódový bod*), podobně jako předchozí kódování. Avšak kódování Unicode není omezeno na používání jednoho bajtu pro jeden znak, a proto je schopné reprezentovat každý znak v každém jazyku v jediném kódování. To znamená, že na rozdíl od ostatních kódování není kódování Unicode omezeno na jeden jazyk, ale dokáže pracovat se znaky z více jazyků.

Jak se ale kódování Unicode ukládá? V současnosti je definováno více než 100 000 znaků znakové sady Unicode, takže i při použití čísel se znaménky je 32bitové celé číslo více než dostačující pro uložení libovolného kódového bodu v kódování Unicode. Nejjednodušší způsob uložení znaků Unicode je posloupnost 32bitových celých čísel, kde jedno celé číslo představuje jeden znak. To zní docela příhodně, neboť tím bychom dostali přímé mapování znaků na 32bitová čísla, což by vedlo k tomu, že by indexování určitého znaku bylo velice rychlé. Nicméně v praxi nejsou věci tak jednoduché, poněvadž některé znaky Unicode lze reprezentovat jedním či dvěma kódovými body. Například „é“ lze reprezentovat jediným kódovým bodem 0xE9 nebo dvěma kódovými body: 0×65 a 0×301 („e“ a „čárka“).

V současnosti se znaková sada Unicode obvykle ukládá na disk a do paměti pomocí kódování UTF-8, UTF-16 nebo UTF-32. První z nich, UTF-8, je zpětně kompatibilní se 7bitovým kódováním ASCII, protože jeho prvních 128 kódových bodů je reprezentováno jednobajtovými hodnotami, které jsou stejné jako hodnoty 7bitových znaků v kódování ASCII. K reprezentaci všech ostatních znaků Unicode používá kódování UTF-8 pro jeden znak dva, tři nebo více bajtů. Díky tomu je UTF-8 velice kompaktní pro reprezentaci textu, který je celý nebo z velké většiny tvořen znaky anglické abecedy. Knihovna Gtk (používaná mimo jiné okénkovým systémem GNOME) používá kódování UTF-8 a vypadá to, že toto kódování se stává de facto standardním formátem pro ukládání textu ve znakové sadě Unicode do souborů. Kódování UTF-8 je kupříkladu výchozím formátem pro kód jazyka XML a toto kódování dále používá řada soudobých webových stránek.

Formát UCS-2 (který je v moderní formě stejný jako UTF-16) používá i spousta dalších softwarových produktů, jako je kupříkladu Java. Tato reprezentace používá pro jeden znak dva nebo čtyři bajty, přičemž nejčastěji používané znaky jsou reprezentovány dvěma bajty. Reprezentace UTF-32 (označovaná též jako UCS-4) používá pro jeden znak čtyři bajty. Použití kódování UTF-16 nebo UTF-32 pro ukládání do souborů nebo posílání přes síťové připojení však skrývá jisté úskalí. Pokud se totiž data posílají jako celá čísla, tak záleží na uspořádání bajtů (buď od nejvyšší nebo od nejnižší adresy). Jedním z řešení tohoto problému je umístit před data označení pořadí bajtů, aby se příjemce dokázal tomuto pořadí přizpůsobit. Tento problém ovšem kódování UTF-8 nepostihuje, což je další důvod, proč je tak populární.

Python reprezentuje znakovou sadu Unicode pomocí formátu UCS-2 (UTF-16) nebo UCS-4 (UTF-32). V případě formátu UCS-2 používá Python malinko zjednodušenou verzi, která pro jeden znak používá vždy dva bajty, a proto dokáže reprezentovat kódové body po hodnotu 0xFFFF. Při použití formátu UCS-4 dokáže Python reprezentovat všechny kódové body znakové sady Unicode. Maximální kódový bod je uložen v atributu určeném pouze pro čtení s názvem `sys.maxunicode`. Je-li jeho hodnota 65535, pak byl Python zkompilován tak, aby používal formát UCS-2. Je-li větší, můžeme říci, že používá formát UCS-4.

Metoda `str.encode()` vrací posloupnost bajtů (ve skutečnosti jde o objekt typu `bytes`, jemuž se budeme věnovat v lekci 7) zakódovanou podle zadaného argumentu. Pomocí této metody můžeme získat určitý vhled do odlišností mezi jednotlivými kódováními a přesvědčit se, proč může vést nesprávně zvolené kódování k chybám:

```
>>> artist = "Tage Ĺsén"
>>> artist.encode("Latin1")
b'Tage \xc5s\xe9n'
>>> artist.encode("CP850")
b'Tage \x8fs\x82n'
>>> artist.encode("utf8")
b'Tage \xc3\x85s\xc3\xa9n'
>>> artist.encode("utf16")
b'\xff\xfeT\x00a\x00g\x00e\x00 \x00\xc5\x00s\x00\xe9\x00n\x00'
```

Písmeno „b" před otevíracími uvozovkami říká, že literál není řetězec, ale že je typu `bytes`. Standardně můžeme při vytváření literálů typu `bytes` používat směsici tisknutelných znaků ASCII a bajtů ve formě čísel v šestnáctkové soustavě, před nimiž jsou umístěny znaky „\x".

Jméno „Tage Åsén“ nemůžeme zakódovat pomocí znakové sady ASCII, protože ta neobsahuje znak „*Å*“ ani žádný jiný znak s diakritikou, takže by došlo k vyvolání výjimky `UnicodeEncodeError`. Kódování Latin-1 (označované též jako ISO-8859-1) je 8bitové kódování, jež obsahuje všechny nezbytné znaky pro uvedené jméno. Na druhou stranu jméno „Pavöl Šemík“ by již takové štěstí nemělo, protože znak „*ö*“ není součástí kódování Latin-1, a proto by jej nešlo úspěšně zakódovat. Obě jména lze samozřejmě zakódovat pomocí znakové sady Unicode. Všimněte si však, že u kódování UTF-16 představují první dva bajty označení pořadí bajtů. Tyto bajty pak použije dekódovací funkce ke zjištění, zda jsou data ve formátu Big nebo Little Endian.

O metodě `str.encode()` si musíme říci ještě několik dalších věcí. U prvního argumentu (název kódování) se rozlišuje velikost písmen, přičemž pomlčky a podtržítka jsou považovány za totéž, takže například „us-ascii“ a „US_ASCII“ jsou stejné názvy. Dále je k dispozici řada aliasů – kupříkladu „latin“, „latin1“, „latin_1“, „ISO-8859-1“, „CP819“ a některé další představují „Latin-1“. Tato metoda může dále přijímat volitelný druhý argument, který metodě říká, jak má zpracovat chyby. Můžeme tak například zakódovat libovolný řetězec do kódování ASCII, pokud jako druhý argument zadáme „ignore“ nebo „replace“, což ovšem znamená možnost ztráty dat. Nechceme-li o data přijít, můžeme zadat „backslashreplace“, což způsobí nahrazení všech znaků nespadajících do znakové sady ASCII příslušnými bajtovými kódy s prefixem „\x“, „\u“ nebo „\U“. Například volání `artist.encode("ascii", "ignore")` vrátí `b'Tage sn'` a volání `artist.encode("ascii", "replace")` vrátí `b'Tage ?s?n'`, zatímco volání `artist.encode("ascii", "backslashreplace")` vrátí `b'Tage \xc5s\xe9n'`. (Řetězec v kódování ASCII můžeme získat také pomocí výrazu `"{0!a}".format(artist)`, který se vyhodnotí na `'Tage \xc5s\xe9n'`.)

Opakem metody `str.encode()` je metoda `bytes.decode()` (a také metoda `bytearray.decode()`), která vrací řetězec s bajty dekódovanými pomocí zadaného kódování:

```
>>> print(b"Tage \xc3\x85s\xc3\xa9n".decode("utf8"))
Tage Åsén
>>> print(b"Tage \xc5s\xe9n".decode("latin1"))
Tage Åsén
```

Rozdíly mezi 8bitovými kódováními Latin-1, CP850 (kódování pro IBM PC) a UTF-8 dávají jasně najevo, že snaha o hádání kódování s největší pravděpodobností není úspěšnou strategií. Kódování UTF-8 se v oblasti holých textových souborů naštěstí stává de facto standardem, takže pozdější generace již možná ani nebudou vědět, že existují také nějaká jiná kódování.

Soubory `.py` s kódem jazyka Python používají kódování UTF-8, takže Python vždy ví, které kódování má u řetězcových literálů použít. To znamená, že do svých řetězců můžeme zapsat libovolné znaky ze znakové sady Unicode, tedy za předpokladu, že je náš editor podporuje.*

Když Python čte data z externích zdrojů, jako jsou sokety, pak nemůže vědět, které kódování používají, a proto vrátí bajty, které pak můžeme adekvátním způsobem dekódovat. U textových souborů používá Python jemnější způsob spočívající v použití místního kódování, není-li kódování stanoveno explicitně.

* Je možné používat i jiná kódování. Přečtěte si tutoriál k jazyku Python s názvem „Source Code Encoding“.

Naštěstí některé souborové formáty svá kódování uvádějí. Například můžeme předpokládat, že soubor XML používá kódování UTF-8, pokud direktiva `<?xml?>` explicitně nespecifikuje odlišné kódování. Při čtení kódu jazyka XML můžeme tedy extrahovat řekněme prvních 1000 bajtů, podívat se na specifikaci kódování, a pokud ji nalezneme, tak dekódovat soubor pomocí uvedeného kódování; v opačném případě přejdeme zpět k dekódování s použitím UTF-8. Tento postup by měl fungovat u libovolného textového souboru s kódem jazyka XML nebo s holým textem, který používá kterékoli z jednobajtových kódování podporovaných Pythonem, kromě kódování založeného na kódu EBCDIC (CP424, CP500) a několika dalších (CP037, CP864, CP865, CP1026, CP1140, HZ, SHIFT-JIS-2004, SHIFT-JISX0213). Tento postup však nebude fungovat u vícebajtových kódování (jako jsou např. UTF-16 a UTF-32). Na webu Python Package Index (viz *http://pypi.python.org/pypi*) jsou k dispozici nejméně dva balíčky Pythonu pro automatickou detekci kódování souboru.

Příklady

V této části využijeme toho, čemu jsme se věnovali v této a předchozí lekci, abychom si ukázali dva malé, ale zato ucelené prográmky, na nichž si procvičíme vše, co jsme se dosud naučili. První program je malinko matematický, se svými 35 řádky je ale docela krátký. Druhý se věnuje zpracování textu a se svými sedmi funkcemi na přibližně 80 řádcích kódu je tak o něco „výživnější“.

Program quadratic.py

Kvadratické rovnice jsou rovnice popisující parabolu ve tvaru $ax^2 + bx + c = 0$, kde $a \neq 0$. Kořeny takovéto rovnice se odvozují ze vzorce $x = (\frac{-b \pm \sqrt{b^2 - 4ac}}{2a})$. Část $b^2 - 4ac$ se označuje jako *diskriminant*. Je-li diskriminant kladný, pak existují dva reálné kořeny, je-li nulový, existuje jeden reálný kořen, a je-li záporný, pak existují dva komplexní kořeny. Napíšeme program, který přijme koeficienty *a, b* a *c* od uživatele (přičemž *b* a *c* mohou být 0) a poté vypočítá a vypíše kořen nebo kořeny.*

Nejdříve se podíváme na ukázkový běh programu a poté si projede jeho kód.

```
quadratic.py
ax² + bx + c = 0
zadejte a: 2.5
zadejte b: 0
zadejte c: -7.25
2.5x² + 0.0x + -7.25 = 0 → x = 1.70293863659 nebo x = -1.70293863659
```

S koeficienty 1,5, -3 a 6 vypadá výstup takto (některá čísla jsou ořezána):

```
1.5x² + -3.0x + 6.0 = 0 → x = (1+1.7320508j) nebo x = (1-1.7320508j)
```

Výstup není tak pěkný, jak by měl být. Kupříkladu místo `+ -3.0x` bychom raději měli mít `- 3.0x` a činitelé s koeficienty s hodnotou 0 by se neměli vůbec zobrazovat. Tyto problémy budete moci napravit ve cvičeních.

---

* Konzola Windows má jen velmi slabou podporu kódování pro UTF-8, a proto se mohou u několika znaků (² a →) používaných v programu quadratic.py vyskytnout problémy. Z tohoto důvodu je k dispozici též program `quadratic_uni.py`, který zobrazí správné symboly na systémech Linux a Mac OS X a na systémech Windows zobrazí alternativní znaky (^2 and →).

Přejděme nyní ke kódu, který začíná třemi příkazy `import`:

```
import cmath
import math
import sys
```

Potřebujeme matematické knihovny pro práci s typem `float` i `complex`, protože kořenové funkce pro reálná a komplexní čísla jsou odlišné. Dále potřebujeme modul `sys`, jenž obsahuje konstantu `sys.float_info.epsilon`, kterou potřebujeme pro porovnávání čísel s pohyblivou řádovou čárkou s nulou.

Kromě toho potřebujeme funkci, která od uživatele získá číslo s pohyblivou řádovou čárkou:

```
def get_float(msg, allow_zero):
    x = None
    while x is None:
        try:
            x = float(input(msg))
            if not allow_zero and abs(x) < sys.float_info.epsilon:
                print("nulu nelze zadat")
                x = None
        except ValueError as err:
            print(err)
    return x
```

Tato funkce bude procházet cyklem, dokud uživatel nezadá platné číslo typu `float` (např. `0.5`, `-9`, `21`, `4.92`); hodnotu `0` přijme pouze tehdy, má-li parametr `allow_zero` hodnotu `True`.

Jakmile je funkce `get_float()` definována, provede se zbývající část kódu. Projdeme si jej ve třech částech a začneme interakcí s uživatelem:

```
print("ax\N{SUPERSCRIPT TWO} + bx + c = 0")
a = get_float("zadejte a: ", False)
b = get_float("zadejte b: ", True)
c = get_float("zadejte c: ", True)
```

Díky funkci `get_float()` je získání koeficientů a, b a c velice jednoduché. Druhý argument představující pravdivostní hodnotu říká, zda lze zadat nulu.

```
x1 = None
x2 = None
discriminant = (b ** 2) - (4 * a * c)
if discriminant == 0:
    x1 = -(b / (2 * a))
else:
    if discriminant > 0:
        root = math.sqrt(discriminant)
    else:  # diskriminant < 0
```

```
        root = cmath.sqrt(discriminant)
    x1 = (-b + root) / (2 * a)
    x2 = (-b - root) / (2 * a)
```

Kód vypadá trošku jinak než vzorec, protože musíme začít výpočtem diskriminantu. Má-li diskriminant hodnotu 0, pak víme, že máme jedno řešení, které můžeme přímo vypočítat. V opačném případě provedeme reálnou a komplexní odmocninu diskriminantu a vypočítáme dva kořeny.

```
equation = ("{0}x\N{SUPERSCRIPT TWO} + {1}x + {2} = 0"
            " \N{RIGHTWARDS ARROW} x = {3}").format(a, b, c, x1)
if x2 is not None:
    equation += " nebo x = {0}".format(x2)
print(equation)
```

Použití metody str.format() s rozbalením mapování ➤ 86

Neprovádíme žádné efektní formátování, protože výchozí formát čísel s pohyblivou řádovou čárkou v Pythonu je pro účely tohoto příkladu dostatečný. Na druhou stranu používáme pro několik speciálních znaků názvy znaků ze znakové sady Unicode.

Robustnější alternativa oproti pozičním argumentům, jejichž názvy polí jsou definovány pomocí indexů, spočívá v použití slovníku vráceného funkcí `locals()`, což je technika, s níž jsme se seznámili v dřívější části této lekce:

```
equation = ("{a}x\N{SUPERSCRIPT TWO} + {b}x + {c} = 0"
            " \N{RIGHTWARDS ARROW} x = {x1}").format(**locals())
```

3.1

A pokud používáme Python 3.1, pak bychom mohli vynechat názvy polí a nechat Python, aby sám tato pole naplnit pomocí pozičních argumentů předaných metodě `str.format()`.

```
equation = ("{}x\N{SUPERSCRIPT TWO} + {}x + {} = 0"
            " \N{RIGHTWARDS ARROW} x = {}").format(a, b, c, x1)
```

Jedná se o pohodlné řešení, které ale není tak robustní jako pojmenované parametry, ani tak všestranné, pokud bychom potřebovali použít specifikace formátu. Nicméně v řadě jednoduchých případů je tato syntaxe snadná a užitečná.

Program csv2html.py

Jedním z běžných požadavků je vzít datovou sadu a zobrazit ji uživateli pomocí jazyka HTML. V této části napíšeme program, který přečte soubor v jednoduchém formátu CSV (Comma Separated Value – hodnoty oddělené čárkou) a vypíše tabulku HTML obsahující jeho data. Python nabízí výkonný a sofistikovaný modul `csv` pro práci s CSV a podobnými formáty, my si však napíšeme veškerý kód sami.

Formát CSV, který budeme podporovat, má na jednom řádku jeden záznam, přičemž každý záznam je rozdělen do polí pomocí čárek. Každé pole může být buď řetězec, nebo číslo. Řetězce musejí být uzavřené do jednoduchých nebo dvojitých uvozovek a čísla by měla být v uvozovkách jen v případě, že obsahuje čárky. Čárky mohou být též uvnitř řetězců, kde se nesmějí považovat za oddělovače polí. Předpokládáme, že první záznam obsahuje pole popisků. Námi vytvářený výstup bude mít podobu

tabulky HTML s textem zarovnaným vlevo (výchozí zarovnání v jazyku HTML) a s čísly zarovnanými vpravo, kdy na jednom řádku bude jeden záznam a v jedné buňce jedno pole.

Program musí vypsat otevírací značku tabulky jazyka HTML, poté přečíst každý řádek data, u každého z nich vypsat příslušný řádek tabulky HTML a na konec vypsat ukončovací značku tabulky HTML. Barva pozadí prvního řádku (který zobrazí popisky polí) bude světle zelená a na pozadí datových řádků se bude střídat bílá a světle žlutá. Musíme se také ujistit, že se speciální znaky jazyka HTML („&“, „<“ a „>“) náležitým způsobem zakódovaly, a dále chceme, aby se řetězce zobrazily trošku čistším způsobem.

Zde je úryvek s ukázkovými daty:

```
"ZEMĚ","2000","2001",2002,2003,2004
"ANTIGUA A BARBUDA",0,0,0,0,0
"ARGENTINA",37,35,33,36,39
"BAHAMY, THE",1,1,1,1,1
"BAHRAJN",5,6,6,6,6
```

Za předpokladu, že ukázková data jsou uložena v souboru *data/co2-sample.csv* a že uživatel zadal příkaz `csv2html.py < data/co2-sample.csv > co2-sample.html`, bude mít soubor *co2-sample.html* obsah podobný následujícímu:

```
<table border='1'><tr bgcolor='lightgreen'>
<td>Země</td><td align='right'>2000</td><td align='right'>2001</td>
<td align='right'>2002</td><td align='right'>2003</td>
<td align='right'>2004</td></tr>
...
<tr bgcolor='lightyellow'><td>Argentina</td>
<td align='right'>37</td><td align='right'>35</td>
<td align='right'>33</td><td align='right'>36</td>
<td align='right'>39</td></tr>
...
</table>
```

Výstup jsme malinko upravili a vynechali jsme některé řádky, místo kterých je uveden výpustek. Použili jsme velmi jednoduchou verzi jazyka HTML (HTML 4 Transitional) bez šablony stylů. Obrázek 2.7 ukazuje, jak tento výstup vypadá ve webovém prohlížeči.

Země	2000	2001	2002	2003	2004
Antigua a Barbuda	0	0	0	0	0
Argentina	37	35	33	36	39
Bahamy	1	1	1	1	1
Bahrajn	5	6	6	6	6

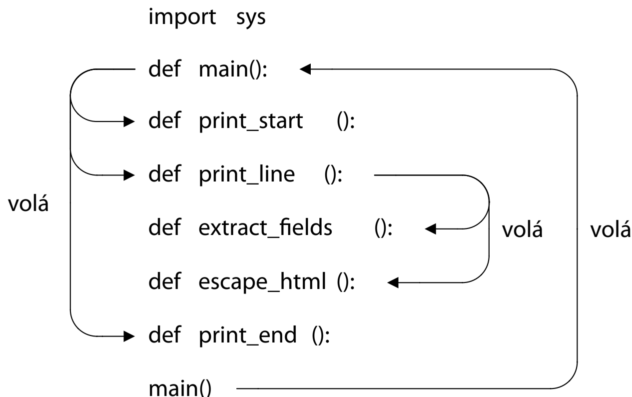
Obrázek 2.7: Tabulka z programu csv2html.py ve webovém prohlížeči

Nyní již tedy víme, jak se program používá a co dělá, takže se můžeme pustit do průzkumu jeho zdrojového kódu. Celý program začíná importem modulu `sys`. My si jej zde neukážeme a od této chvíle si už nebudeme ukazovat žádné importy, pokud nebudou nějak výjimečné nebo hodné výkladu. Posledním příkazem v programu je jediné volání funkce:

```
main()
```

I když Python na rozdíl od některých jiných jazyků žádný vstupní bod nepotřebuje, je docela běžné, že programátoři v Pythonu ve svých programech vytvářejí funkci s názvem `main()`, kterou pak volají pro zahájení činnosti programu. Žádnou funkci nelze zavolat před jejím vytvořením, a proto je nutné zajistit, aby se funkce `main()` zavolala až poté, kdy budou definovány všechny funkce, na kterých závisí. Na pořadí, ve kterém jsou funkce v souboru uvedeny (tj. pořadí, ve kterém jsou vytvářeny), vůbec nezáleží.

V programu csv2html.py je první volanou funkcí funkce `main()`, která zase volá funkce `print_start()` a `print_line()`, přičemž funkce `print_line()` volá funkce `extract_fields()` a `escape_html()`. Struktura našeho programu je znázorněna na obrázku 2.8.



Obrázek 2.8: Struktura programu csv2html.py

Když Python čte soubor, začíná odshora. V tomto příkladu tedy začíná provedením importu, poté vytvoří funkci `main()` a poté vytvoří ostatní funkce v pořadí, ve kterém jsou v souboru uvedeny. Když Python nakonec dospěje k volání funkce `main()` na konci souboru, budou všechny funkce, které `main()` volá (a všechny funkce, které tyto funkce volají), již existovat. Provádění tak, jak o něm běžně uvažujeme, začíná v okamžiku zavolání funkce `main()`.

Nyní se postupně podíváme na každou z funkcí, přičemž začneme funkcí `main()`:

```
def main():
    maxwidth = 100
    print_start()
    count = 0
    while True:
        try:
            line = input()
```

```
            if count == 0:
                color = "lightgreen"
            elif count % 2:
                color = "white"
            else:
                color = "lightyellow"
            print_line(line, color, maxwidth)
            count += 1
        except EOFError:
            break
    print_end()
```

Proměnná `maxwidth` slouží k omezení počtu znaků v buňce. Je-li pole větší než tato hodnota, tak jej ořežeme a za oříznutý text přidáme výpustek. Na funkce `print_start()`, `print_line()` a `print_end()` se podíváme vzápětí. Cyklus `while` prochází přes všechny řádky vstupu, který sice může pocházet i od uživatele píšícího na klávesnici, my však očekáváme, že se bude jednat o přesměrovaný soubor. Nastavíme barvu, kterou chceme použít, a pomocí funkce `print_line()` vypíšeme řádek jako řádek tabulky HTML:

```
def print_start():
    print("<table border='1'>")

def print_end():
    print("</table>")
```

Tyto dvě funkce bychom ani vytvářet nemuseli. Stačilo by jen umístit příslušná volání funkce `print()` do těla funkce `main()`. Naším cílem je však oddělovat logiku programu, což je flexibilnější, i když v takto malém příkladě to nehraje žádnou roli.

```
def print_line(line, color, maxwidth):
    print("<tr bgcolor='{0}'>".format(color))
    fields = extract_fields(line)
    for field in fields:
        if not field:
            print("<td></td>")
        else:
            number = field.replace(",", "")
            try:
                x = float(number)
                print("<td align='right'>{0:d}</td>".format(round(x)))
            except ValueError:
                field = field.title()
                field = field.replace(" A ", " a ")
                if len(field) <= maxwidth:
                    field = escape_html(field)
                else:
                    field = "{0} ...".
```

```
                        format(escape_html(field[:maxwidth]))
                print("<td>{0}</td>".format(field))
    print("</tr>")
```

K rozdělení každého řádku do polí nemůžeme použít příkaz `str.split(",")`, protože čárky se mohou vyskytovat také uvnitř řetězců uzavřených do uvozovek. Naše řešení jsme tedy umístili do funkce `extract_fields()`. Jakmile máme seznam polí (jako řetězců bez obklopujících uvozovek), můžeme je projít a pro každé vytvořit buňku tabulky.

Je-li pole prázdné, vypíšeme prázdnou buňku. Je-li pole uzavřené do uvozovek, pak by mohlo jít o řetězec nebo číslo, které bylo uzavřeno do uvozovek kvůli interním čárkám (např. `"1,566"`). Tuto situaci vyřešíme tak, že vytvoříme kopii pole s odstraněnými čárkami a toto pole se pokusíme převést na hodnotu typu `float`. Je-li převod úspěšný, vypíšeme vpravo zarovnanou buňku s polem zaokrouhleným na nejbližší celé číslo, které vypíšeme jako celé číslo. Pokud převod selže, vypíšeme pole jako řetězec. V takovém případě použijeme metodu `str.title()` k úpravě velikosti písmen a dále nahradíme osamělá písmena `"A"` písmeny `"a"`. Pokud pole není příliš dlouhé, použijeme jej celé. V opačném případě jej ořízneme na velikost definovanou proměnnou `maxwidth` a přidáme výpustek označující oříznutí. V obou případech zakódujeme speciální znaky jazyka HTML, které by mohlo dané pole obsahovat.

```
def extract_fields(line):
    fields = []
    field = ""
    quote = None
    for c in line:
        if c in "\"'":
            if quote is None: # začátek řetězce uzavřeného do uvozovek
                quote = c
            elif quote == c:  # konec řetězce uzavřeného do uvozovek
                quote = None
            else:
                field += c    # další uvozovku uvnitř řetězce v uvozovkách
            continue
        if quote is None and c == ",": # konec pole
            fields.append(field)
            field = ""
        else:
            field += c        # akumulace pole
    if field:
        fields.append(field)  # přidání posledního pole
    return fields
```

Tato funkce přečte zadaný řádek znak po znaku a postupně vytvoří seznam polí, z nichž každé je řetězcem bez uzavíracích uvozovek. Funkce si poradí i s poli, která nejsou uzavřené do uvozovek, dále s poli, která jsou uzavřená do jednoduchých či dvojitých uvozovek, a správně zpracuje také čárky a uvozovky (jednoduché uvozovky v řetězcích uzavřených do dvojitých uvozovek a dvojité uvozovky v řetězcích uzavřených do jednoduchých uvozovek).

```
def escape_html(text):
    text = text.replace("&", "&amp;")
    text = text.replace("<", "&lt;")
    text = text.replace(">", "&gt;")
    return text
```

Tato funkce jednoduše nahradí každý speciální znak jazyka HTML příslušnou entitou jazyka HTML. Nejdříve je samozřejmě nutné nahradit ampersand, ačkoliv na pořadí ostrých závorek už nezáleží. Standardní knihovna Pythonu obsahuje malinko sofistikovanější verzi této funkce. Budete mít šanci se s ní seznámit ve cvičeních a opět se s ní setkáte v lekci 7.

Shrnutí

Na začátku této lekce jsme si ukázali seznam klíčových slov jazyka Python a popsali jsme si pravidla, která Python aplikuje na identifikátory. Ty nejsou díky podpoře znakové sady Unicode v jazyku Python omezeny pouze na určitou podmnožinu znaků z malé znakové sady, jako je ASCII nebo Latin-1.

Dále jsme si popsali datový typ `int` jazyka Python, který se od podobných typů ve většině ostatních jazyků liší v tom, že nemá žádné skutečné omezení velikosti. Celá čísla v jazyku Python mohou být tak velká, jak dovoluje paměť počítače, a není vůbec žádný problém pracovat s čísly se stovkami cifer. Všechny nejzákladnější datové typy jazyka Python jsou neměnitelné, což lze jen zřídkakdy postřehnout, protože díky operátorům rozšířeného přiřazení (`+=`, `*=`, `-=`, `/=` a další) můžeme používat velice přirozenou syntaxi, zatímco interpret jazyka Python vytváří v pozadí výsledné objekty, s nimiž opětně svazuje naše proměnné. Celočíselné literály se obvykle zapisují jako desítková čísla, ovšem pomocí prefixu `0b` lze psát binární literály, pomocí prefixu `0o` osmičkové literály a pomocí prefixu `0x` šestnáctkové literály.

Při dělení dvou celých čísel pomocí operátoru `/` je výsledek vždy typu `float`. To je sice od mnoha jiných běžně používaných jazyků odlišné, ale pomáhá to předcházet některých docela zákeřným chybám, k nimž může docházet, když dělení výsledek tiše ořeže. (Chceme-li přesto použít celočíselné dělení, můžeme sáhnout po operátoru `//`.)

Jazyk Python nabízí datový typ `bool`, který umí uchovávat buď `True`, nebo `False`. Dále Python nabízí logické operátory `and`, `or` a `not`, z nichž oba dva binární operátory (`and` a `or`) používají logiku zkráceného vyhodnocování.

K dispozici jsou tři druhy čísel s pohyblivou řádovou čárkou: `float`, `complex` a `decimal.Decimal`. Nejčastěji se používá typ `float`, který reprezentuje s dvojitou přesností číslo s pohyblivou řádovou čárkou, jehož přesné číselné charakteristiky závisejí na knihovně jazyka C, C# nebo Java, s níž byl Python sestaven. Komplexní čísla jsou reprezentována jako dvě čísla typu `float`. Jedno uchovává reálnou a druhé imaginární hodnotu. Typ `decimal.Decimal` poskytuje modul `decimal`. Čísla tohoto typu mají ve výchozím nastavení 28 desetinných míst. Tuto přesnost lze ale dále zvýšit nebo snížit tak, aby vyhovovala aktuálním potřebám.

Všechny tři typy s pohyblivou řádovou čárkou lze použít s příslušnými vestavěnými matematickými operátory a funkcemi. Kromě toho modul `math` nabízí pestrou škálu trigonometrických, hyper-

bolických a logaritmických funkcí, které lze použít s čísly typu `float`, přičemž modul `cmath` nabízí podobnou skupinu funkcí pro komplexní čísla.

Větší část lekce byla věnována řetězcům. Řetězcové literály jazyka Python lze vytvářet pomocí jednoduchých či dvojitých uvozovek. Můžeme také použít řetězce s trojitými uvozovkami, chceme-li do řetězce přímo zahrnout znaky nových řádků a uvozovky. K vkládání speciální znaků (jako je např. tabulátor a nový řádek) lze použít speciální posloupnosti znaků (`\t` a `\n`). Dále je možné pomocí šestnáctkových kódů nebo specifických názvů vkládat znaky ze znakové sady Unicode. Ačkoliv řetězce podporují stejné porovnávací operátory jako ostatní typy jazyka Python, je třeba si uvědomit, že řazení řetězců obsahujících neanglické znaky může být problematické.

Řetězce jsou posloupnosti, a proto lze pomocí velmi jednoduché, přesto však výkonné syntaxe řezacího operátoru (`[]`) řetězce řezat a krokovat. Řetězce lze také operátorem + spojovat a operátorem * replikovat. Dále můžeme použít verze těchto operátorů s rozšířeným přiřazením (+= a *=), i když častěji se pro spojování řetězců používá metoda `str.join()`. Řetězce mají spoustu dalších metod, včetně metod pro testování vlastností řetězce (např. `str.isspace()` a `str.isalpha()`), pro změnu velikosti písmen (např. `str.lower()` a `str.title()`), pro vyhledávání (např. `str.find()` a `str.index()`) a řady dalších.

Podpora řetězců v jazyku Python je skutečně excelentní; díky niž můžeme texty snadno vyhledávat a extrahovat nebo celé řetězce či části řetězců porovnávat, nahrazovat znaky či podřetězce, rozdělovat řetězce na seznam podřetězců a spojovat seznamy řetězců do jediného řetězce.

Pravděpodobně nejvšestrannější metodou je metoda `str.format()`. Tato metoda se používá pro vytváření řetězců pomocí nahrazovacích polí a proměnných, které se dosadí do těchto polí, a pomocí specifikací formátu, které přesně definují charakteristiky každého pole nahrazovaného nějakou hodnotu. Syntaxe pro názvy nahrazovacích polí nám umožňuje přistupovat k argumentům metody podle jejich pozice nebo jména (pro klíčované argumenty) a pomocí názvu indexu, klíče nebo atributu přistupovat k položce nebo atributu argumentu. Díky specifikaci formátu můžeme stanovit výplňový znak, zarovnání a minimální šířku pole. Kromě toho u čísel můžeme nastavovat, jak se zobrazí znaménko, a u čísel s pohyblivou řádovou čárkou můžeme určit počet desetinných míst a také to, zda se má použít standardní nebo exponenciální notace.

Probírali jsme také spletité téma kódování znaků. Soubory `.py` s kódem jazyka Python používají standardně kódování UTF-8 znakové sady Unicode, a proto mohou mít komentáře, identifikátory a data zapsaná v téměř libovolném lidském jazyku. Pomocí metody `str.encode()` můžeme převést řetězec na posloupnost bajtů s použitím určitého kódování a metodou `bytes.decode()` můžeme posloupnost bajtů v určitém kódování převést zpět na řetězec. Práce s pestrou škálou kódování znaků, které se v současné době používají, může být velmi nepohodlná, avšak kódování UTF-8 se v oblasti holých textových souborů rychle stává de facto standardem (a již nyní je standardem pro soubory XML), takže tento problém by měl v následujících letech slábnout.

Kromě datových typů probíraných v této lekci nabízí jazyk Python dva další vestavěné datové typy `bytes` a `bytearray`, kterým se budeme věnovat v lekci 7. Jazyk Python dále nabízí několik datových typů představujících kolekce, z nichž některé jsou vestavěné a jiné jsou součástí standardní knihovny. V následující lekci se tedy podíváme na nejdůležitější datové typy jazyka Python určené pro práci s kolekcemi.

Cvičení

1. Upravte program print_unicode.py tak, aby uživatel mohl na příkazovém řádku zadat několik samostatných slov a aby se vytiskly pouze ty řádky, jejichž název znaku znakové sady Unicode obsahuje všechna slova zadaná uživatelem. To znamená, že můžeme napsat příkazy, jako je tento:

   ```
   print_unicode_ans.py greek symbol
   ```

   Jeden ze způsobů, jak to provést, spočívá v nahrazení proměnné word (která obsahovala hodnotu 0, None nebo řetězec) seznamem words. Nezapomeňte aktualizovat kromě kódu také informace o použití. Změny zahrnují přidání méně než deseti řádků kódu a změnu maximálně deseti dalších. Řešení najdete v souboru print_unicode_ans.py. (Uživatelé systému Windows by měli upravovat soubor print_unicode_uni.py; výsledek naleznou v souboru print_unicode_uni_ans.py.)

2. Upravte program quadratic.py tak, aby se koeficienty s hodnotou 0,0 nevypisovaly a aby se záporné koeficienty vypisovaly jako - n a ne + -n. To zahrnuje nahrazení posledních pěti řádků zhruba patnácti řádky. Řešení najdete v souboru quadratic_ans.py. (Uživatelé systému Windows by měli upravovat soubor quadratic_uni.py, výsledek naleznou v souboru quadratic_uni_ans.py.)

3. V programu csv2html.py vymažte funkci escape_html() a použijte místo ní funkci xml.sax.saxutils.escape() z modulu xml.sax.saxutils. To je snadné, stačí jen jeden nový řádek (příkaz import), pět řádků vymazat (nechtěná funkce) a jeden řádek změnit (použití funkce xml.sax.saxutils.escape() místo escape_html()). Řešení je k dispozici v souboru csv2html1_ans.py.

4. Znovu upravte soubor csv2html.py a tentokrát přidejte novou funkci s názvem process_options(). Tato funkce by se měla volat z funkce main() a měla by vracet n-tici dvou hodnot: maxwidth (typu int) a format (typu str). Funkce process_options() by měla při zavolání nastavit výchozí hodnotu proměnné maxwidth na 100 a výchozí hodnotu proměnné format na ".0f" – tato hodnota se pak použije jako specifikátor formátu při vypisování čísel.

   Pokud uživatel na příkazovém řádku zadá „-h“ nebo „--help“, vypíše se zpráva s informacemi o použití a vrátí dvojice (None, None). (V tomto případě by funkce main() neměla provést vůbec nic.) V opačném případě by tato funkce měla přečíst argumenty zadané na příkazovém řádku a provést příslušná přiřazení. Když například uživatel zadá „maxwidth=n“, pak se do proměnné maxwidth přiřadí hodnota n a podobně se nastaví proměnná formát při zadání „format=s“. Zde je ukázkový běh zobrazující výpis informací o použití:

   ```
   csv2html2_ans.py -h
   použití:
   csv2html.py [maxwidth=int] [format=str] < infile.csv > outfile.html

   Parametr maxwidth je volitelné celé číslo;
   je-li zadáno, nastaví maximum počet znaků,
   které lze vypsat u řetězcových polí,
   ```

```
jinak se tato pole ořežou na 100 znaků.

Parametr format je formát pro čísla;
není-li zadán, použije se ".0f".
A zde je příkazový řádek s nastavením obou voleb:
csv2html2_ans.py maxwidth=20 format=0.2f < mydata.csv > mydata.html
```

Nezapomeňte upravit funkci `print_line()` tak, aby pro výpis čísel používala nastavený formát. Bude třeba jí předat další argument, přidat jeden řádek a upravit jiný řádek. To bude mít také malý vliv na funkce `main()`. Funkce `process_options()` by měla mít přibližně dvacet pět řádků (včetně asi tak devíti pro zprávu s informacemi o použití). Toto cvičení může být pro nezkušené programátory obtížnější.

K dispozici máte dva soubory s testovacími daty: `data/co2-sample.csv` a `data/co2-fromfossilfuels.csv`. Řešení je k dispozici v souboru `csv2html2_ans.py`. V lekci 5 uvidíte, jak pomocí modulu `optparse` zjednodušit zpracování argumentů na příkazovém řádku.

LEKCE 3

Datové typy představující kolekce

V této lekci:

- Typy představující posloupnost
- Množinové typy
- Typy představující mapování
- Procházení a kopírování kolekcí

V předchozí lekci jsme se seznámili s nejdůležitějšími základními datovými typy jazyka Python. V této lekci naše programovací dovednosti rozšíříme o možnost shromažďovat datové prvky pomocí datových typů jazyka Python představující kolekce. Budeme se věnovat n-ticím a seznamům a představíme si také nové datové typy představující kolekce, mezi něž patří množiny a slovníky, a všechny si je projdeme pořádně do hloubky.*

Kromě kolekcí se podíváme také na to, jak vytvářet datové prvky, které jsou seskupením jiných datových prvků (podobně jako v případě typu `struct` v jazycích C a C++ nebo `record` v jazyku Pascal); s takovými prvky lze pak v případě potřeby pracovat jako s jednotlivým elementem, zatímco v nich obsažené prvky zůstávají samostatně přístupné. Seskupené prvky můžeme pochopitelně umisťovat do kolekcí jako jakékoli jiné prvky.

Díky možnosti vkládat datové prvky do kolekce je pak mnohem snazší provádět operace, které je třeba aplikovat na všechny tyto prvky. Kromě toho se pak kolekce prvků snadněji čtou ze souborů. V této lekci se budeme průběžně věnovat také úplným základům práce s textovými soubory, přičemž většinu podrobností (včetně ošetřování chyb) si vysvětlíme v lekci 7.

Jakmile probereme jednotlivé datové typy představující kolekce, podíváme se na to, jak kolekce procházet, protože stejná syntaxe se používá pro všechny kolekce jazyka Python. Rozebereme si též problémy a techniky související s kopírováním kolekcí.

Typy představující posloupnost

Typ představující posloupnost je takový typ, který podporuje operátor příslušnosti (`in`), funkci zjišťující velikost (`len()`) a řezání (`[]`) a který je iterovatelný. Jazyk Python nabízí pět vestavěných typů představujících posloupnost: `bytearray` (pole bajtů), `bytes` (bajty), `list` (seznam), `str` (řetězec) a `tuple` (n-tice). První dva typy budeme probírat samostatně v lekci 7. Standardní knihovna poskytuje ještě několik dalších typů představujících posloupnost, z nichž nejpozoruhodnější je typ `collections.namedtuple`. Při procházení vracejí všechny tyto posloupnosti své prvky v určitém pořadí.

Řetězce ➤ 71

V předchozí lekci jsme se věnovali řetězcům. V této části se zaměříme na n-tice, pojmenované n-tice a seznamy.

N-tice

Řezání a krokování řetězců ➤ 74

N-tice je uspořádaná posloupnost nula nebo více odkazů na objekty. N-tice podporují stejnou syntaxi pro řezání a krokování jako řetězce. Díky tomu je extrahování prvků z n-tice velice snadné. Podobně jako řetězce jsou také n-tice neměnitelné, takže nemůžeme nahradit nebo vymazat žádný z jejich prvků. Pokud chceme mít možnost modifikovat uspořádanou posloupnost, musíme místo n-tice použít seznam. Pokud už n-tici máme a chceme ji upravit, pak ji můžeme pomocí převodní funkce `list()` převést na seznam a poté aplikovat změny na výsledný seznam.

Mělké a hloubkové kopírování ➤ 146

Datový typ `tuple` lze zavolat i jako funkci `tuple()`. Bez argumentů vrátí prázdnou n-tici, s argumentem typu `tuple` vrátí jeho mělkou kopii a argument jakéhokoliv jiného typu se pokusí převést na n-tici. Funkci `tuple()` nelze předat více než jeden argument. N-tice je možné vytvářet i bez funkce `tuple()`. Prázdnou n-tici vytvoříme pomocí prázdných závorek `()` a n-tici s jedním nebo více prvky

* Definice v této lekci týkající se toho, co je typ představující posloupnost nebo typ představující mapování, jsou sice praktické, ale neformální. S formálnějšími definicemi se seznámíme v lekci 8.

vytvoříme pomocí čárek. Někdy je nutné n-tice uzavřít do závorek, abychom předešli syntaktické nejednoznačnosti. Kupříkladu k předání n-tice `1, 2, 3` nějaké funkci musíme napsat `function((1, 2, 3))`.

Obrázek 3.1 ukazuje n-tici `t = "venuše", -28, "zelená", "21", 19.74` a indexové pozice jednotlivých prvků uvnitř této n-tice. Řetězce se indexují podobným způsobem, avšak zatímco řetězce mají na každé pozici znak, n-tice mají na každé pozici odkaz na objekt.

t[-5]	t[-4]	t[-3]	t[-2]	t[-1]
'venuše'	-28	'zelená'	'21'	19.74
t[0]	t[1]	t[2]	t[3]	t[4]

Obrázek 3.1 Indexové pozice v n-tici

N-tice nabízejí jen dvě metody. Metoda `t.count(x)` vrací počet výskytů objektu `x` v n-tici `t` a metoda `t.index()` vrací pozici nejlevějšího výskytu objektu `x` v n-tici `t` nebo vyvolá výjimku `ValueError`, pokud v n-tici žádný objekt `x` není. (Tyto metody jsou k dispozici také u seznamů.)

Kromě toho lze n-tice použít s operátory + (spojení), * (replikace) a `[]` (řezání) a také s operátory `in` a `not in` pro testování příslušnosti. Ačkoliv n-tice jsou neměnné, můžeme použít i operátory rozšířeného přiřazení += a *=, což je možné díky tomu, že Python v pozadí vytvoří novou n-tici pro uchování výsledku a nastaví odkaz na objekt na levé straně tak, aby ukazoval na tuto nově vytvořenou n-tici. Stejná technika se používá i při aplikaci těchto operátorů na řetězce. N-tice lze porovnávat pomocí standardních porovnávacích operátorů (<, <=, ==, !=, >=, >), přičemž porovnávání se aplikuje na jednotlivé prvky (a rekurzivně pro vnořené prvky, jako jsou n-tice uvnitř n-tic).

Pojďme se podívat na několik příkladů řezání. Začneme extrahováním jednoho prvku a řezu prvků:

```
>>> hair = "černá", "hnědá", "blond", "červená"
>>> hair[2]
'blond'
>>> hair[-3:]    # stejné jako: hair[1:]
('hnědá', 'blond', 'červená')
```

Tyto příkazy fungují stejně u řetězců, seznamů a u jakýchkoli jiných typů představujících posloupnost.

```
>>> hair[:2], "šedá", hair[2:]
(('černá', 'hnědá'), 'šedá', ('blond', 'červená'))
```

Zde jsme se pokusili vytvořit novou pětici, ale obdrželi jsme trojici, která obsahuje dvě dvojice. Příčinou je to, že jsme použili operátor čárka se třemi prvky (n-tice, řetězec a n-tice). Pro získání jediné n-tice se všemi prvky musíme jednotlivé n-tice spojit:

```
>>> hair[:2] + ("šedá",) + hair[2:]
('černá', 'hnědá', 'šedá', 'blond', 'červená')
```

K vytvoření n-tice s jedním prvkem stačí napsat čárku, avšak v tomto případě bychom při vynechání závorek obdrželi chybu `TypeError` (poněvadž Python by si myslel, že se pokoušíme spojit řetězec s n-ticí).

V této knize (od tohoto okamžiku) budeme pro zápis n-tic používat určitý styl. Pokud budeme mít n-tice na levé straně binárního operátoru nebo na pravé straně unárního příkazu, pak závorky vynecháme; ve všech ostatních případech budeme závorky uvádět. Zde je několik příkladů:

```
a, b = (1, 2)                      # levá strana binárního operátoru

del a, b                           # pravá strana unárního operátoru

def f(x):
    return x, x ** 2               # pravá strana unárního příkazu

for x, y in ((1, 1), (2, 4), (3, 9)):  # levá strana binárního operátoru
    print(x, y)
```

Nejedná se o styl programování, který by bylo nutné bezpodmínečně dodržovat. Například někteří programátoři raději píší závorky pokaždé, což je stejné jako reprezentační forma n-tice, zatímco jiní je používají pouze tam, kde jsou nezbytně nutné.

```
>>> eyes = ("hnědá", "ořechová", "jantarová", "zelená", "modrá", "šedá")
>>> colors = (hair, eyes)
>>> colors[1][3:-1]
('zelená', 'modrá')
```

Zde jsme vnořili dvě n-tice do jiné n-tice. Tímto způsobem lze přímo vytvářet vnořené kolekce libovolné hloubky. Na řez můžeme dle potřeby opakovaně aplikovat řezací operátor `[]`:

```
>>> things = (1, -7.5, ("hrách", (5, "Xyz"), "fronta"))
>>> things[2][1][1][2]
'z'
```

Rozeberme si tento kód krok za krokem. Výraz `things[2]` se vyhodnotí na třetí prvek n-tice (neboť první prvek má index `0`), což je n-tice `("hrách", (5, "Xyz"), "fronta")`. Výraz `things[2][1]` nám dá druhý prvek n-tice `things[2]`, což je opět n-tice `(5, "Xyz")`. A výraz `things[2][1][1]` se vyhodnotí na druhý prvek této n-tice, což je řetězec `"Xyz"`. Nakonec nám výraz `things[2][1][1][2]` vrátí třetí prvek (znak) v řetězci, což je `"z"`.

N-tice jsou schopné uchovávat libovolné prvky libovolného datového typu, včetně typů představujících kolekce, jako jsou n-tice a seznamy, protože ve skutečnosti se neuchovávají samotné objekty, ale odkazy na objekty. Použití podobně složité zanořené datové struktury může být matoucí. Jedno z řešení spočívá v přiřazení jmen určitým indexovým pozicím:

```
>>> MANUFACTURER, MODEL, SEATING = (0, 1, 2)
>>> MINIMUM, MAXIMUM = (0, 1)
>>> aircraft = ("Airbus", "A320-200", (100, 220))
```

```
>>> aircraft[SEATING][MAXIMUM]
220
```

Tento kód je jistě mnohem smysluplnější než zápis `aircraft[2][1]`, vyžaduje však vytvoření spousty proměnných a nevypadá hezky. V následující části se seznámíme s elegantním řešením tohoto problému.

Na prvních dvou řádcích výše uvedeného úryvku kódu jsme v obou příkazech provedli přiřazení do n-tic. Máme-li na pravé straně přiřazení posloupnost (zde máme n-tici) a na levé straně n-tici, pak říkáme, že pravá strana byla *rozbalena* (unpack). Rozbalení posloupnosti lze použít k prohození hodnot:

```
a, b = (b, a)
```

Přesně řečeno, závorky na pravé straně nejsou nutné, ale jak jsme si řekli již dříve, v této knize budeme psát kód takovým stylem, že závorky vynecháme u operandů na levé straně binárních operátorů a u operandů na pravé straně unárních příkazů, přičemž všude jinde budeme závorky psát.

Již jsme se setkali s příklady rozbalení posloupnosti v kontextu cyklů `for ... in`:

```
for x, y in ((-3, 4), (5, 12), (28, -45)):
    print(math.hypot(x, y))
```

Zde procházíme přes n-tici dvojic, které postupně rozbalujeme do proměnných `x` a `y`.

Pojmenované n-tice

Pojmenovaná n-tice se chová stejně jako holá n-tice a vykazuje stejné výkonové charakteristiky. To, co přidává, je možnost odkazovat na prvky v n-tici kromě indexové pozice také jménem, díky čemuž můžeme vytvářet seskupení datových prvků.

Modul `collections` nabízí funkci `namedtuple()`, která se používá k vytváření vlastních datových typů představujících n-tice:

```
Sale = collections.namedtuple("Sale",
                              "productid customerid date quantity price")
```

Prvním argumentem funkce `collections.namedtuple()` je název datového typu n-tice, který chceme vytvořit. Druhý argument je řetězec mezerou oddělených jmen, jedno pro každý prvek, pod kterými budou vystupovat prvky našich vlastních n-tic. První argument a jména ve druhém argumentu musejí být platnými identifikátory jazyka Python. Funkce vrátí naši novou třídu (datový typ), jejímž prostřednictvím můžeme vytvářet pojmenované n-tice. V tomto případě můžeme tedy pracovat s výrazem `Sale` jako s jakoukoli jinou třídou jazyka Python (jako je např. `tuple`) a vytvářet objekt typu `Sale`. (V řeči objektově orientovaného programování je každá takto vytvářená třída podtřídou třídy `tuple`. Objektově orientovanému programování včetně odvozování nových tříd se budeme věnovat v lekci 6.)

Zde je příklad:

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 79.9))
sales.append(Sale(419, 874, "2008-09-15", 1, 184.9))
```

Zde jsme vytvořili seznam prvků dvou prvků typu `Sale`, tedy, dvou našich vlastních n-tic. Na prvky v těchto n-ticích se můžeme odkazovat pomocí indexových pozic, takže například hodnota prvního prvku je `sales[0][-1]` (tj. `79.9`). Kromě toho však můžeme použít také jména, díky kterým je výsledný kód daleko čistší:

```
total = 0
for sale in sales:
    total += sale.quantity * sale.price
print("Celkem {0:.2f} Kč".format(total))  # vypíše: Celkem 424.6 Kč
```

Čitelnost a pohodlnost nabízená pojmenovanými n-ticemi je často velice užitečná. Vraťme se nyní k příkladu s letadlem z předchozí části (strana 112), který můžeme přepsat do mnohem pěknějšího tvaru:

```
>>> Aircraft = collections.namedtuple("Aircraft",
                                      "manufacturer model seating")
>>> Seating = collections.namedtuple("Seating", "minimum maximum")
>>> aircraft = Aircraft("Airbus", "A320-200", Seating(100, 220))
>>> aircraft.seating.maximum
220
```

Co se týče extrahování prvků z pojmenované n-tice do řetězců, můžeme použít jednu ze tří hlavních technik.

```
>>> print("{0} {1}".format(aircraft.manufacturer, aircraft.model))
Airbus A320-200
```

Zde přistupujeme ke každému prvku n-tice, který nás zajímá, pomocí přístupu k atributům pojmenované n-tice. Díky tomu obdržíme nejkratší a nejjednodušší formátovací řetězec. (A v Pythonu 3.1 bychom jej mohli ještě zmenšit jen na `"{} {}"`.) Avšak tento přístup znamená, že musíme sledovat argumenty předávané metodě `str.format()`, abychom měli přehled o tom, jaké nahrazovací texty se použijí. To je méně čisté ve srovnání s pojmenovanými poli uvedenými přímo ve formátovacím řetězci.

```
"{0.manufacturer} {0.model}".format(aircraft)
```

Zde jsme použili jediný poziční argument a názvy atributů pojmenované n-tice jako názvy polí ve formátovacím řetězci. To je sice ve srovnání se samotnými pozičními argumenty mnohem čistší, je ale škoda, že musíme specifikovat poziční hodnotu (a to i v Pythonu 3.1). Naštěstí existuje ještě hezčí způsob.

Pojmenované n-tice mají několik soukromých metod, což jsou metody, jejichž název začíná podtržítkem. Jedna z nich (`namedtuple._asdict()`) je tak užitečná, že si ji ukážeme v akci.*

```
"{manufacturer} {model}".format(**aircraft._asdict())
```

Soukromá metoda `namedtuple._asdict()` vrací mapování dvojic klíč-hodnota, kde každý klíč je název elementu n-tice a každá hodnota je odpovídající hodnotou. Pomocí rozbalování mapování jsme převáděli mapování do argumentů ve tvaru klíč-hodnotu pro metodu `str.format()`.

Použití metody `str.format()` s rozbalením mapování ➢ 86

I když pojmenované n-tice mohou být velmi užitečné, v lekci se seznámíme s objektově orientovaným programováním, posuneme se dál od jednoduchých pojmenovaných n-tic a naučíme se, jak vytvářet vlastní datové typy, které uchovávají datové prvky a které mají také své vlastní, námi definované metody.

Seznamy

Seznam je uspořádaná posloupnost nula nebo více odkazů na objekty. Seznamy podporují stejnou syntaxi pro řezání a krokování jako řetězce a n-tice, díky které lze snadno extrahovat prvky ze seznamu. Na rozdíl od řetězců a n-tic jsou seznamy měnitelné, takže můžeme nahrazovat a mazat kterýkoli z jejich prvků. Kromě toho je možné vkládat, nahrazovat a mazat řezy v seznamu.

Řezání a krokování řetězců ➢ 74

Datový typ `list` lze zavolat i jako funkci `list()`. Bez argumentů vrátí prázdný seznam, s argumentem typu `list` vrátí jeho mělkou kopii a argument jakéhokoliv jiného typu se pokusí převést na seznam. Funkci `list()` nelze předat více než jeden argument. Seznamy je možné vytvářet i bez funkce `list()`. Prázdný seznam vytvoříme pomocí prázdných hranatých závorek `[]` a seznam s jedním nebo více prvky vytvoříme pomocí posloupnosti prvků oddělených čárkou uvnitř hranatých závorek. Další možnost, jak vytvořit seznam, spočívá v použití seznamové komprehenze, což je téma, k němuž se dostaneme za chvíli.

Mělké a hloubkové kopírování ➢ 146

Seznamové komprehenze ➢ 120

Všechny prvky v seznamu jsou skutečně odkazy na objekty, a proto mohou seznamy, stejně jako n-tice, uchovávat prvky libovolného datového typu, včetně typů představujících kolekce, jako jsou seznamy a n-tice. Seznamy lze porovnávat pomocí standardních porovnávacích operátorů (`<`, `<=`, `==`, `!=`, `>=`, `>`), přičemž porovnávání se aplikuje na jednotlivé prvky (a rekurzivně pro vnořené prvky, jako jsou seznamy nebo n-tice uvnitř seznamů).

Po přiřazení `L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]` obdržíme seznam znázorněný na obrázku 3.2.

L[-6]	L[-5]	L[-4]	L[-3]	L[-2]	L[-1]
-17.5	'kilo'	49	'V'	['ram', 5, 'echo']	7
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

Obrázek 3.2 Indexové pozice v seznamu

* U soukromých metod, jako je `namedtuple._asdict()`, není zaručeno, že budou k dispozici ve všech verzích Pythonu 3.x, i když metoda namedtuple._asdict() je dostupná v obou verzích Pythonu 3.0 a 3.1.

Zůstaneme u seznamu `L`, na který můžeme použít řezací operátor (klidně i opakovaně) pro přístup k prvkům v seznamu, jak ukazují následující rovnosti:

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k'
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] == 'echo'
L[4][2][1] == L[4][2][-3] == L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

Seznamy lze vnořovat, procházet a řezat, stejně jako n-tice. Všechny příklady s n-ticemi, které jsme si v předchozí části ukázali, by měly ve skutečnosti pracovat naprosto stejně, pokud bychom místo n-tic použili seznamy. Seznamy podporují testování příslušnosti pomocí operátorů `in` a `not int`, spojování s operátorem +, rozšiřování s operátorem += (např. připojení všech prvků v pravém operandu) a replikaci s operátorem * a *=. Seznamy lze použít také s vestavěnou funkcí `len()` a s příkazem `del`, který si popíšeme v panelu „Mazání prvků pomocí příkazu del" (strana 117). Kromě toho seznamy nabízejí metody uvedené v tabulce 3.1.

Tabulka 3.1 Metody seznamu

Syntaxe	Popis
L.append(x)	Připojí prvek `x` na konec seznamu `L`.
L.count(x)	Vrátí počet výskytů prvku `x` v seznamu `L`.
L.extend(m) L += m	Připojí všechny prvky iterovatelného objektu `m` na konec seznamu `L`. Totéž provádí operátor +=.
L.index(x, *začátek*, *konec*)	Vrátí indexovou pozici nejlevějšího výskytu prvku `x` v seznamu `L` (nebo v řezu *začátek*:*konec* seznamu `L`). V opačném případě vrátí výjimku `ValueError`.
L.insert(i, x)	Vloží prvek `x` do seznamu `L` na indexovou pozici `i`.
L.pop()	Vrátí a odstraní nejpravější prvek seznamu `L`.
L.pop(i)	Vrátí a odstraní prvek na indexové pozici `i` v seznamu `L`.
L.remove(x)	Odstraní nejlevější výskyt prvku `x` ze seznamu L nebo vyvolá výjimku `ValueError`, pokud prvek `x` nenalezne.
L.reverse()	Obrátí seznam `L` na místě.
L.sort(...)	Seřadí seznam `L` na místě. Tato metoda přijímá stejné volitelné argumenty *klíč* a *obrátit* jako vestavěná metoda `sorted()`.

sorted() ➤ 138, 144

Toto je pouze náhled elektronické knihy. Zakoupení její plné verze je možné v elektronickém obchodě společnosti eReading.