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

WWW

Ke stažení zdrojové kódy příkladů z knihy

computer press

Addison Wesley

Python 3

Vyšlo také v tištěné verzi

Objednat můžete na
www.computerpress.cz
www.albatrosmedia.cz

Mark Summerfield
Python 3 – e-kniha

Mark Summerfield

Python 3
Výukový kurz

Computer Press
Brno
2021

Stručný obsah

Obsah

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 knihy na *www.albatrosmedia.cz*.

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`.

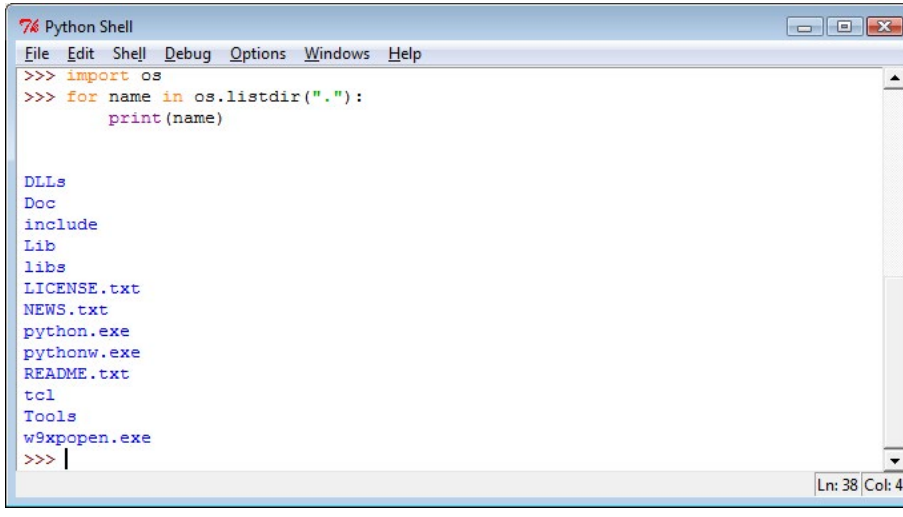
Řada výkonných editorů holého textu, jako je Vim nebo Emacs, obsahuje vestavěnou podporu pro úpravu programů psaných v jazyku Python. Tato podpora obvykle zahrnuje barevné zvýrazňování syntaxe a správné odsazování řádků. Alternativou je pak programovací prostředí Pythonu s názvem IDLE. V systémech Windows 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
21062458333711437339583605536734086463779019080109822250862195507 2
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.

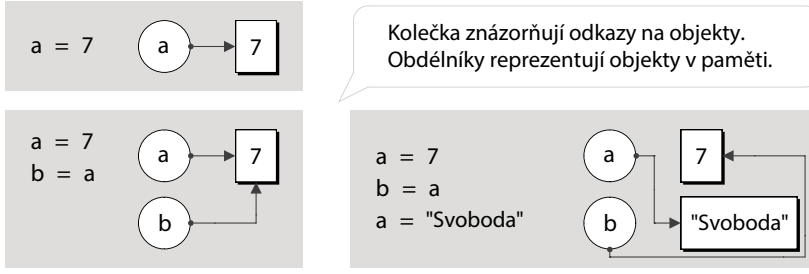
Nyní budeme pokračovat s naším příkladem a provedeme několik opětovných svázání. Jak jsme si řekli již dříve, komentáře začínají znakem # a pokračují až do konce řádku:

```
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

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.

Identifikátory a klíčová slova ➤ 58

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 ntice, 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
```

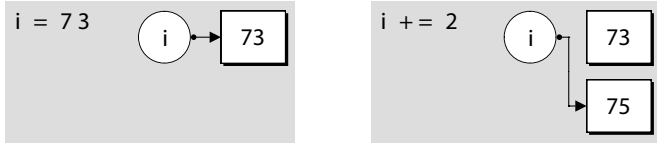
Operátor dělení nevrací celé, ale desetinné číslo. Většina ostatních jazyků vrátí celé číslo vytvořené ořezáním desetinné části. Pokud potřebujeme celočíselný výsledek, můžeme jej vždy převést pomocí `int()` (nebo pomocí ořezávacího operátoru dělení `//`, k němuž se dostaneme později).

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.

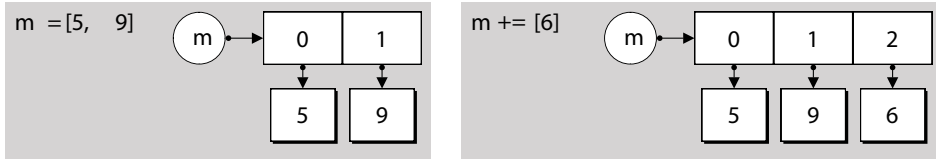
Jazyk Python přetěžuje (tj. opětovně používá pro jiný datový typ) operátory + a += pro řetězce i seznamy, přičemž první znamená zřetězení a druhý připojení pro řetězce a rozšíření (připojení dalšího seznamu) pro seznamy:

```
>>> 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).

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

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

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: