Učebnice jazyka Python (aneb Létající cirkus)

Release 2.2

Jan Švec

16. prosince 2002

PyCZ
Email: honza@py.cz

Překlad z originálního anglického dokumentu "Python Tutorial" autorů Guida van Rossuma a Freda L. Drakea. Originální copyright:

Abstrakt

Python je vysoce výkonný programovací jazyk používající efektivní vysokoúrovňové datové typy, přičemž jednoduše a elegantně řeší otázku objektově orientovaného programování. Jeho syntaxe a dynamické typy spolu s interpretováním kódu dotváří pověst ideálního nástroje pro psaní skriptů a rychlý vývoj aplikací (Rapid Application Development, RAD). Samotný interpretr jazyka je spustitelný na velkém množství platforem včetně Linuxu, Windows, MacOS a DOS.

Zdrojové kódy interpretru Pythonu a standardních knihoven jsou volně ke stažení z domovské stránky Pythonu (http://www.python.org/) a je možné je dále volně modifikovat a distribuovat. Na této stránce také najdete předkompilované instalační balíčky pro většinu podporovaných platforem a nechybí ani množství odkazů na další moduly, programy a nástroje určené uživatelům Pythonu.

Prostředí jazyka je snadno rozšiřitelné pomocí funkcí a datových typů napsaných v jazycích C nebo C++. Python lze také použít jako skriptovací jazyk pro aplikace v jiných jazycích.

Tato učebnice jazyka Python vás zasvětí do logiky jazyka a jeho základních vlastností. Rovněž se dozvíte více o běhovém systému, přičemž je ideální mít tento systém přímo nainstalovaný na vašem systému a veškeré příklady si zkoušet přímo v něm. Každá ukázka je ale psána s ohledem na názornost, proto by měla být snadno pochopitelná i pro začátečníka, který nemá přímo možnost si ji hned vyzkoušet.

Pro podrobnější popis standardních objektů a modulů nahlédněte do dokumentu "Python Library Reference", zatímco v dokumentu "Python Reference Manual" naleznete formální definici jazyka (všechny zde uváděné dokumenty jsou součástí distribučního balíčku zdrojových kódů). Těm, kteří potřebují Python rozšiřovat o své vlastní doplňky v jazycích C nebo C++, se budou hodit dokumenty "Extending and Embedding the Python Interpretr" a "Python/C API Reference". Tyto dokumenty můžete zároveň najít na stránkách http://www.python.org.

Tato publikace se nesnaží být vyčerpávající, vysvětluje pouze nejzákladnější vlastnosti jazyka Python. S její pomocí si ale můžete vytvořit představu o tom, jak samotný jazyk vypadá. Po jejím přečtení dokážete pochopit strukturu již existujícího kódu. Zároveň se naučíte potřebné základy pro psaní nových programů v jazyce Python. Po zvládnutí této učebnice můžete pokračovat studováním dokumentace k mnoha modulům popsaných dokumentem "Python Library Reference".

OBSAH

KAPITOLA

PRVNÍ

Proč používat zrovna Python?

Jestliže jste již někdy napsali opravdu velký shellovský skript, jistě znáte tuto situaci: chcete do programu přidat nějakou novou funkci, ale ten je již příliš pomalý, rozsáhlý a komplikovaný, případně tato nová vlastnost vyžaduje volat jinou funkci, které je přístupná pouze z jazyka C . . . Obyčejně tento problém není natolik podstatný, abyste se rozhodli celý skript přepsal do C, třeba používáte řetězce proměnné délky nebo další vysokoúrovňové datové typy (jako třeba seznamy nebo asociativní pole), jež můžete snadno používat v shellu ale již ne v C. Jistě, všechny tyto typy lze implementovat v C, ale zabralo by to pravděpodobně mnoho času a nejspíš si na to ani netroufáte.

Stejně tak jiná situace: pracujete na projektu, který používá množství C knihoven a cyklus napiš/zkompiluj/otestuj je již příliš zdlouhavý. Vy ale potřebujete pravý opak - rychlý a efektivní vývoj. Třeba dokonce chcete napsat program, který bude používat skriptovací jazyk, ale nechcete navrhovat a ladit vlastní interpretr.

V těchto případech je právě Python vhodným jazykem pro vás. Lze se jej velice snadno naučit (mluví se o několika málo dnech pro získání základních znalostí), přesto se jedná o skutečný programovací jazyk nabízející mnoho typů a struktur. Nechají se v něm napsat opravdu velké a rozsáhlé projekty. Také je více odolný proti chybám programátora než obyčejné C, přičemž těží ze všech výhod vysokoúrovňového jazyka, má vestavěné vysokoúrovňové datové typy jako seznamy nebo asociativní pole, díky čemuž ušetříte mnoho času, který byste jinak věnovali jejich implementování v C. Kvůli těmto obecnějším datovým typům je Python vhodnější pro mnohem více problémů než jazyky *Awk* nebo *Perl*, mnoho úkolů se za použití Pythonu řeší dokonce snadněji než v těchto jazycích.

Python umožňuje rozdělit vaše programy do samostatných modulů, které mohou být snadno použity i v jiných programech a projektech. Již základní distribuce Pythonu obsahuje velké množství standardních modulů, které můžete použít jako základ vašich programů, případně se z nich můžete přiučit hodně běžně programátorských obratů. Mezi nimi najdete také moduly pro práci se soubory, systémovými voláními, sokety a také moduly sloužící jako rozhraní ke grafickému uživatelskému rozhraní Tk.

Python je interpretovaný jazyk, čímž programátorovi šetří množství času. Již žádné kompilování a linkování programů. Interpretr může být použit interaktivně, což vám umožní snadné experimentování s jazykem samotným stejně jako s jednotlivými moduly. Jednoduše takto lze testovat také vaše uživatelské funkce a třídy. S trochou zručnosti je použitelný i jako výkonný kalkulátor obohacený o mnoho matematických funkcí.

Programy v Pythonu jsou velice kompaktní a snadno pochopitelné. Zároveň jsou typicky mnohem kratší než ekvivalentní kód implementovaný v C nebo C++ a to z mnoha důvodů:

- vysokoúrovňové datové typy umožňující rychlé a komplexní operace v jediném výrazu;
- seskupování výrazů se děje pomocí odsazení narozdíl od používání otevírací a uzavírací složené závorky v C a C++;
- není nutné deklarovat proměnné a argumenty funkcí, jazyk dokonce nerozlišuje ani jejich typ;

Python je *rozšiřitelný*: umíte-li programovat v jazyce C pak pro vás bude hračkou přidávání nových interních funkcí nebo modulů pro zajištění maximální rychlosti časově náročných operací, případně takto můžete interpretovanému kódu zajistit přístup ke knihovnám, které jsou distribuované pouze v binární formě (např. knihovny od výrobců hard-

ware apod.). Můžete také přilinkovat interpretr k vaší aplikaci napsané v C a naplno tak využít potenciálu tohoto jazyka, který je jako stvořený pro úlohu ideálního skriptovacího jazyka.

Jazyk Python je pojmenovaný podle pořadu společnosti BBC "Monty Python's Flying Circus" a jeho název tedy nemá nic společného s hady.[1] Guido van Rossum je vášnivým fanouškem tohoto pořadu a při práci s Pythonem se doslova na každém rohu setkáte s proprietami majícími svůj původ v tomto pořadu.

1.1 Obsah této učebnice

Jestliže jste dočetli až sem, jistě jste již poznali, že Python není jen jedním z řady interpretovaných jazyků. Chcete poznat tento skvělý jazyk více do hloubky? Pak pravděpodobně nejlepší cestou, jak se ho naučit, je přečtení této učebnice spolu s jeho používáním.

V další kapitole této učebnice již získáte základní znalosti pro ovládání samotného interpretru - běhového prostředí jazyka. Tato kapitola se pohybuje spíše v teoretické rovině, ale po jejím přečtení již budete připraveni začít s jazykem pracovat naplno.

Zbytek učebnice na jednoduchých příkladech ukazuje, jak používat jazyk co nejefektivněji, přičemž vysvětluje i nezbytně nutnou teorii jako jsou jednoduché výrazy, příkazy a datové typy, poté budou následovat výklad funkcí a modulů a nakonec nastíníme problematiku pokročilejších vlastností jazyka jako jsou výjimky a třídy včetně implementace objektově orientovaného programování v jazyce Python.

[1] "Python" v angličtině znamená hroznýš

KAPITOLA

DRUHÁ

Používáme interpretr jazyka Python

2.1 Spuštění běhového systému

Interpretr jazyka Python je většinou nainstalován jako soubor ‘/usr/local/bin/python’. Máte-li zařazen adresář ‘/usr/local/bin’ do cesty, v níž shell vyhledává spustitelné soubory, můžete prostředí jazyka spustit přímo zapsáním příkazu

```
python
```

Jméno adresáře, v němž je Python nainstalován, je závislé na volbách nastavených při jeho překladu, proto je možné, že ve vašem systému tomu bude jinak. Více informací vám jistě sdělí správce vašeho systému.

Pro ukončení interpretru stiskněte znak konce souboru (EOF, `Control-D` v systému UNIX, `Control-Z` v systémech Dos a Windows). V tomto případě předá prostředí jazyka shellu návratovou hodnotu 0. Nelze-li interpretr takto opustit, lze použít následující příkaz: ‘`import sys; sys.exit()`’.

Editování příkazového řádku v interpretru jazyka Python není příliš pohodlné. Proto lze interpretr v UNIXovém prostředí zkompilovat s podporou knihovny GNU Readline, která zpřístupní historii příkazů i doplňování jmen příkazů a proměnných. Velice rychlý způsob, jak zjistit, zda váš interpretr podporuje tuto knihovnu, je zapsání znaku `Control-P` po první výzvě, kterou dostanete po spuštění Pythonu. Jestliže se ozve pípnutí, má interpretr pravděpodobně podporu knihovny Readline zakompilovánu. Jestliže se objeví znaky `^P`, pak nejsou rozšířené úpravy příkazového řádku podporovány a vy můžete používat pouze klávesu `Backspace` pro smazání předchozího znaku. Pro další informace o konfiguraci součinnosti interpretru a knihovny GNU Readline se podívejte do Dodatku A.

Interpretr pracuje podobně jako shell systému UNIX: když je spuštěn a jeho standardní vstup je spojen s terminálovým zařízením, načítá a spouští příkazy interaktivně. Je-li mu při spuštění předáno jako argument jméno souboru, případně je standardní vstup přesměrován z nějakého souboru, čte a spouští příkazy přímo z tohoto souboru - *skriptu*.

Třetí způsob, jak lze provést libovolný kód jazyka Python, je spuštění příkazu ‘**python -c** *příkazy* `[argumenty] ...`’, jenž rovnou vykoná *příkazy*. Jde o obdobu stejnojmenné volby unixového shellu. Jelikož příkazy jazyka Python často obsahují mezery nebo jiné speciální znaky musí se uvodit nejlépe dvojitými uvozovkami, aby se zabránilo jejich interpretování shellem.

Pamatujte, že je rozdíl mezi spuštěním příkazu ‘`python soubor`’ a ‘`python < soubor`’. V případě prvním se nejprve načte a zkontroluje celý soubor, na případné chyby v syntaxi se tudíž přijde ihned, ještě před spuštěním prvního příkazu. Ve druhém případě se načítají příkazy jeden po druhém, následně jsou kontrolovány a spouštěny, je-li některý příkaz syntakticky chybně, zjistí se to až těsně před jeho spuštěním. Pozornost je třeba také věnovat příkazům, které čtou vstup od uživatele, přesměrování standardního vstupu totiž platí i pro ně a ty tudíž (nechtěně) načtou následující kód programu.

Velice častým požadavkem je po vykonání skriptu spustit interaktivní mód a tím pádem předat řízení programu uživateli, čehož lze s výhodou využít při ladění programů apod. To zajistí volba **-i** předaná příkazu **python** (tato volba

nefunguje správně v případě přesměrovaného standardního vstupu).

2.1.1 Předávání argumentů

Jméno skriptu a další argumenty předané z příkazového řádku jsou uloženy v proměnné `sys.argv`. Jedná se o seznam řetězců obsahující minimálně jeden prvek - jméno skriptu. Pouze je-li aktivní interaktivní mód, pak je prvek `sys.argv[0]` roven prázdnému řetězce. Jde-li o skript čtený ze standardního vstupu, je `sys.argv[0]` nastaven na hodnotu `'-'`. Při použití volby **-c** *příkazy* odpovídá nultý prvek seznamu `sys.argv` hodnotě `'-c'`. Zde je důležité podotknout, že všechny argumenty předané za **-c** *příkazy* již nejsou interpretrem zpracovány, dojde pouze k jejich uložení do seznamu `sys.argv`.

2.1.2 Interaktivní mód

Jsou-li příkazy čteny z terminálu, říkáme, že interpretr je v *interaktivním módu*. V tomto módu se nejprve zobrazí uvítací zpráva obsahující informace o verzi a autorských právech. Následně běhové prostředí vytiskne *primární výzvu* (většinou '`>>> `') a čeká na zadání příkazu. Pro vstup složitějších příkazů rozložených přes více řádků se používá i *sekundární výzva* (implicitně '`... `').

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Typickým příkladem víceřádkového příkazu jsou konstrukce pro řízení toku programu. Jako příklad si uvedeme konstrukci `if`:

```
>>> zeme_je_placka = 1
>>> if zeme_je_placka:
...     print "Bacha, at' z ní nespadnete!"
...
Bacha, at' z ní nespadnete!
```

2.2 Běhové prostředí jazyka Python

2.2.1 Obsluha chyb

Při výskytu chyby vytiskne interpretr chybovou zprávu obsahující výpis volaných funkcí a popis chyby samotné. Dojde-li k chybě v interaktivním módu, vrátí se řízení zpět k primární výzvě, při vzniku chyby ve skriptu se nejprve vytiskne hlášení o chybě a poté se skript ukončí s nenulovým návratovým kódem (v tomto případě nejsou výjimky obsloužené větví `except` příkazu `try` brány jako chyby). Všechna chybová hlášení jsou vypisována na standardní chybový výstup, čímž se zabrání pomíchání chybových výpisů s výstupem samotného programu (ten je vypisován na standardní výstup). Fatální chyby jako vnitřní nekonzistence datových struktur či některé případy vyčerpání paměti můžou způsobit rovnou ukončení interpretru s nenulovým návratovým kódem.

Stisk kombinace `Control-C` (případně `DEL`) vyvolá přerušení. To v interaktivním módu způsobí okamžitý návrat k primární výzvě. [1] Přerušení vzniklé za běhu příkazu ho ukončí a vyvolá výjimku `KeyboardInterrupt`, kterou lze

[1] Chybně nakonfigurovaná knihovna GNU Readline může tento stisk odchytit, k přerušení pak nedojde

odchytit obvyklým způsobem příkazem `try`. [2]

2.2.2 Spustitelné skripty

Na BSD kompatibilních systémech UNIX můžete libovolný skript jazyka Python učinit spustitelným souborem uvedením této "magické" sekvence na začátku souboru:

```
#!/usr/bin/env python
```

(Stejný postup jistě znají ti, kteří píší skriptu unixového shellu.)

Pokud má skript nastaven executable bit (viz. příkaz **chmod**(1), více informací v manuálových stránkách) a spustitelný soubor interpretru se nachází v uživatelově implicitní cestě (tj. cestě, kde shell vyhledává spustitelné soubory a příkazy, určuje jí proměnná prostředí PATH), lze tento skript spustit přímo z příkazové řádky shellu. Je důležité, aby znaky '`#!`' byly první dva znaky celého souboru. Všimněte si, že znak '#' je v Pythonu použit jako začátek komentáře, tudíž sekvence '`#!`' je interpretrem považována za obyčejný komentář.

2.2.3 Soubory načítané při startu interpretru

Používáte-li Python interaktivně, často potřebujete spouštět některé příkazy při každém startu prostředí. To lze snadno zajistit nastavením proměnná prostředí PYTHONSTARTUP tak, aby ukazovala na jméno souboru obsahujícího požadované příkazy. Tento soubor je obdobou souboru '.profile' v unixových shellech.

Podobně jako '.profile' je i tento soubor čten pouze v případě interaktivního sezení, při spouštění skriptů se neuplatní! Všechny příkazu načítané na začátku interaktivního sezení jsou spouštěny ve stejném prostoru jmen, ve kterém následně budou interaktivně spouštěny příkazy. Tudíž ty objekty, které tyto příkazy definují (nebo importují) lze používat přímo v příkazech, které spouštíme v interaktivním módu. V případě potřeby můžete v tomto souboru nastavit i výzvy interpretru (primární výzvu reprezentuje proměnná `sys.ps1`, sekundární `sys.ps2`).

Pokud potřebujete načítat i soubor, který je uložen v pracovním adresáři (pro nastavení různých voleb pro různé projekty apod.), musíte v hlavním souboru, na nějž ukazuje proměnná PYTHONSTARTUP, zapsat příkazy na způsob:

```
if os.path.isfile('.pythonrc.py'):
    execfile('.pythonrc.py')
```

Přejete-li si načítat soubor uvedený v proměnné PYTHONSTARTUP i v případě skriptů, musíte tento soubor spustit sami. To učiní tyto příkazy uvedená na začátku skriptu:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

[2] Více o výjimkách viz. kapitola 8.

KAPITOLA

TŘETÍ

Úvod do jazyka Python

Ve všech příkladech v této knize, které uvádějí ukázky vzorového sezení je vstup pro interpretr označen uvedením výzvy (primární ‘`>>> `’ a sekundární ‘`... `’). Výstup z programu je uváděn na samotném řádku přesně tak, jak jej příkaz `print` vytiskl. Sekundární výzva uvedená na samostatném řádku znamená prázdný řádek a je použita k ukončení víceřádkového příkazu.

Mnoho příkladů v této publikaci, přestože jsou určeny pro zápis v interaktivním módu, obsahuje komentáře. Každý komentář v jazyce Python začíná znakem křížek ‘#’ a pokračuje do konce fyzického řádku. Komentář se může objevit jak na začátku řádky, tak může následovat po "bílých znacích" [1] nebo kódu. Znak ‘#’ uvedený uvnitř řetězce *neznamená* komentář, je považován za znak řetězce:

```
# toto je první komentář
SPAM = 1                 # a toto je druhý komentář
                         # ... a nyní třetí!
STRING = '# Toto není komentář.'
```

3.1 Python jako kalkulátor

Nyní si vyzkoušíme několik jednoduchých příkazů jazyka Python. Spusťte si interpretr a počkejte na zobrazení primární výzvy ‘`>>> `’.

3.1.1 Čísla

Interpretr se chová jako jednoduchý kalkulátor. Můžete zapsat libovolný výraz a on vypíše jeho hodnotu. Zápis výrazů je jednoduchý: operátory `+`, `-`, `*` a `/` fungují stejně jako v jiných jazycích (například Pascal nebo C), rovněž můžete používat závorky pro změnu priority výrazů. Například:

[1] Tzv. bílé znaky jsou všechny znaky, které reprezentují bílá místa v textu, čili mezery, tabulátory, konce řádků apod.

```
>>> 2+2
4
>>> # Toto je komentář
... 2+2
4
>>> 2+2  # a toto je komentář na stejném řádku jako kód
4
>>> (50-5*6)/4
5
>>> # Dělení celých čísel vrátí celé číslo:
... 7/3
2
>>> 7/-3
-3
```

Stejně jako v C je znak rovnítko (‘`=`’) určen pro přiřazení hodnoty proměnné. Hodnota proměnné po přiřazení již není interaktivním interpretrem vypsána:

```
>>> vyska = 20
>>> sirka = 5*9
>>> vyska * sirka
900
```

Hodnota může být přiřazena i více proměnným najednou:

```
>>> x = y = z = 0  # Vynuluj x, y a z
>>> x
0
>>> y
0
>>> z
0
```

Python plně podporuje operace v plovoucí řádové čárce (tj. desetinná čísla). Operátor pracující s různými typy operandů si nejprve zkonvertuje celá čísla na čísla v plovoucí řádové čárce a následně provede výpočet (obdobné chování možná znáte z jazyka C):

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Python také plně podporuje komplexní čísla, přičemž imaginární číslo je zapisováno s příponou ‘`j`’ nebo ‘`J`’. Komplexní čísla zapisujeme ve tvaru ‘`(`*Re* `+` *Im*`j)`’ nebo je můžeme vytvořit pomocí interní funkce ‘`complex(`*Re*`,` *Im*`)`’:

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Komplexní čísla jsou vždy reprezentována dvojicí desetinných čísel, reálnou a imaginární částí. Chceme-li získat velikosti těchto částí čísla *z*, použijeme zápisu *z*`.real` a *z*`.imag`:

```
>>> z=1.5+0.5j
>>> z.real
1.5
>>> z.imag
0.5
```

Poněvadž v matematice neexistuje způsob, jak převést komplexní číslo na reální, ani Python nedovoluje použití konverzních funkcí `float()`, `int()` a `long()` s komplexním argumentem. Raději použijte funkci `abs(`*z*`)` pro získání absolutní hodnoty komplexního čísla, nebo zápis `z.real` reprezentují reálnou část čísla:

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Pokud používáte Python jako stolní kalkulátor, pak se můžete velice snadno vrátit k předchozímu výsledku — ten reprezentuje proměnná _, například:

```
>>> urok = 12.5 / 100
>>> penize = 100.50
>>> penize * urok
12.5625
>>> penize + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Hodnota proměnné _ by nikdy neměla být modifikována uživatelem. Pokud byste jí přiřadili hodnotu, vytvořili byste

nezávislou lokální proměnnou se stejným jménem, která by zakryla interní proměnnou s tímto chováním.

3.1.2 Řetězce

Podobně jako s čísly můžete v Python pracovat i s řetězci. Ty mohou být zapsány mnoha způsoby, především je možné je uvodit jak jednoduchými, tak i dvojitými uvozovkami:

```
>>> 'houby s voctem'
'houby s voctem'
>>> 'rock\'n\'roll'
"rock'n'roll"
>>> "rock'n'roll"
"rock'n'roll"
>>> '"To je vražda," napsala.'
'"To je vražda," napsala.'
>>> "\"To je vražda,\" napsala."
'"To je vražda," napsala.'
>>> '"To je rock\'n\'roll," řekla.'
'"To je rock\'n\'roll," řekla.'
```

Mnohdy programátor potřebuje řetězec, který je rozložen přes více řádků. Dosáhnout toho opět můžeme několika způsoby. První z nich, který se neváže pouze na řetězce, je spojení dvou po sobě jdoucích řádků znakem zpětného lomítka:

```
hello = 'Toto je dlouhý řetězec obsahující mnoho\n\
řádek textu, stejně jej zapisujete i v C.\n\
    "Bílé" znaky na začátku řádku se samozřejmě\
 berou v úvahu.'

print hello
```

Nevýhodou tohoto přístupu je nutnost všechny řádky ukončit vložením znaků `\n`. Zpětné lomítko způsobí ignorování následujícího znaku nového řádku, řetězec tak může pokračovat na dalším řádku, aniž by bylo nutné ho ukončit uvozovkami. To demonstruje předchozí příklad, jehož výstupem je tento text:

```
Toto je dlouhý řetězec obsahující mnoho
řádek textu, stejně jej zapisujete i v C.
    "Bílé" znaky na začátku řádku se samozřejmě berou v úvahu.
```

Python podporuje i tzv. raw řetězce (cosi jak ryzí, syrové řetězce), u nichž se řídící (escape) sekvence nepřevádí na odpovídající znaky.[2]. Raw řetězce charakterizuje předpona *r*. Potom řetězec `r'\n'` odpovídá dvěma znakům - zpětnému lomítku a znaku n, kdežto řetězec `'\n'` je jediný znak nového řádku:

```
hello = r'Toto je dlouhý řetězec obsahující mnoho\n\
řádek textu, stejně jej zapisujete i v C.'

print hello
```

Tento příklad vytiskne text:

[2]Escape sekvencí je například `\n` - znak nového řádku nebo `\t` - tabulátor

```
Toto je dlouhý řetězec obsahující mnoho\n\
řádek textu, stejně jej zapisujete i v C.
```

Další možností, jak vytvořit víceřádkový řetězec je jeho uzavření mezi odpovídající pár trojitých uvozovek (`"""` nebo `'''`). To má tu výhodu, že nemusíme explicitně zapisovat konce řádků, ty bude řetězec obsahovat přesně tak, jak jsou zapsány ve zdrojovém kódu:

```
print """
Použití: nakladač [VOLBY]
     -h                        Zobraz tuto zprávu
     -H hostname               Připoj se na tento počítač
"""
```

Tato ukázka vytiskne následující výstup:

```
Použití: nakladač [VOLBY]
     -h                        Zobraz tuto zprávu
     -H hostname               Připoj se na tento počítač
```

Abychom věděli přesnou podobu řetězce, vytiskne jej interpretr stejným způsobem jako jej zapisujeme, přičemž i všechny speciální znaky jsou uvozeny zpětným lomítkem. (Pokud chceme zobrazit "hodnotu" řetězce, tj. to, co skutečně obsahuje, můžeme použít příkaz `print` popsaný později. Ten řetězec vytiskne bez uvozovek a se všemi speciálními znaky.)

Řetězce můžeme spojovat pomocí operátoru `+`, dokonce je lze opakovat operátorem `*`:

```
>>> slovo = 'Help' + 'A'
>>> slovo
'HelpA'
>>> '<' + slovo*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Nalezne-li interpretr v kódu dva zápisy řetězců bezprostředně za sebou, spojí je dohromady jako by mezi nimi ležel operátor `+`. První řádek příkladu mohl být tedy zapsán jako `slovo = 'Help' 'A'`. Ale pozor, takto lze spojovat pouze zápisy řetězců, pro spojení řetězce a výrazu *musíme* použít operátor `+`!

```
>>> import string
>>> 'str' 'ing'                   #  <-  Správně
'string'
>>> string.strip('str') + 'ing'   #  <-  Správně
'string'
>>> string.strip('str') 'ing'     #  <-  CHYBNĚ!!!
  File "<stdin>", line 1, in ?
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

Řetězce můžeme (podobně jako v jazyce C) indexovat. První znak řetězce pak má index 0. Poněvadž je Python velice uživatelsky přítulný, nekomplikuje život programátora speciálním typem určeným pro jediný znak — každý znak řetězce je opět řetězec s délkou 1. Na získání podřetězce nepotřebujeme žádné speciální funkce, samotný jazyk (podobně jako jazyk Icon) podporuje *indexování subsekvencí*[3]. Subsekvenci indexujeme podobně jako jednotlivé znaky, pouze potřebuje dva indexy (začátek a konec subsekvence), které oddělíme dvojtečkou:

```
>>> slovo[4]
'A'
>>> slovo[0:2]
'He'
>>> slovo[2:4]
'lp'
```

Mezi řetězci v C a v Pythonu ale existuje obrovský rozdíl. Řetězce v jazyce Python *nelze* měnit. Jde o celkem logický závěr, řetězec `'ahoj'` vždy bude řetězec `'ahoj'`, proč bychom ho tedy měli měnit?[4]. Pokusíme-li se změnit určitou pozici v řetězci, dojde k chybě:

```
>>> slovo[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> slovo[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Proto jedinou cestou, jak vytvářet nové řetězce, je jejich kombinování, které je velice jednoduché a přitom efektivní:

```
>>> 'x' + slovo[1:]
'xelpA'
>>> 'Splat' + slovo[4]
'SplatA'
```

Slice indexy mají ještě další specifické vlastnosti. Vynecháme-li první index, je za něj automaticky dosazena nula (začátek řetězce). Při neuvedení druhého indexu se použije délka řetězce (čili konec řetězce): [5].

```
>>> slovo[:2]    # První dva znaky
'He'
>>> slovo[2:]    # Vše s výjimkou prvních dvou znaků
'lpA'
```

Kód ve tvaru `s[:i] + s[i:]` je samozřejmě vyhodnocen jako `s`:

```
>>> slovo[:2] + slovo[2:]
'HelpA'
>>> slovo[:3] + slovo[3:]
'HelpA'
```

Další vlastností slice indexů je jejich automatické "zarovnávání" na rozměr řetězce. Je-li totiž index použitý ve slice

[3] V originální dokumentaci *slice*, v této publikaci budeme někdy používat i výraz slice operace apod.
[4] Obdobně číslo 1 vždy je číslo 1, také se nikdo nezamýšlí nad tím, jestli jde jeho hodnota změnit.
[5] Ve skutečnosti je druhý index nahrazen velikostí proměnné *sys.maxint* určující maximální velikost celého čísla na dané platformě.

konstrukci příliš velký, je nahrazen délkou řetězce. Podobně — pokud je dolní index větší než horní, je výsledkem prázdný řetězec:

```
>>> slovo[1:100]
'elpA'
>>> slovo[10:]
''
>>> slovo[2:1]
''
```

Pokud jsou indexy záporná čísla, dojde k počítání od konce řetězce. Názorně to ukazuje následující příklad i s komentáři:

```
>>> slovo[-1]     # Poslední znak
'A'
>>> slovo[-2]     # Předposlední znak
'p'
>>> slovo[-2:]    # Poslední dva znaky
'pA'
>>> slovo[:-2]    # Vše kromě posledních dvou znaků
'Hel'
```

Pozor ale, -0 je totéž co 0, k žádnému indexování od konce řetězce tím pádem nedojde:

```
>>> slovo[-0]     # (-0 je totéž co 0)
'H'
```

Záporné indexy při indexaci podřetězců jsou zarovnány na velikost řetězce, proto není chybou, sahají-li indexy mimo řetězec. To platí ale pouze pro slice indexy, indexy, které vystupují samostatně jsou ponechány tak, jak jsou. Pokud je řetězec kratší a index padne mimo něj, dojde k chybě:

```
>>> slovo[-100:]
'HelpA'
>>> slovo[-10]    # CHYBNĚ!!!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Pokud i přesto, že jsme se podsekvencím tolik věnovali, nechápete, jak jednotlivé znaky a podsekvence z nich složené získat, možná vám přijde vhod následující schema.

Důležité je si zapamatovat, že slice indexy ukazují *mezi* znaky, přičemž levá hrana prvního znaku má číslo 0 a pravá hrana posledního znaku řetězce o *n* znacích má index *n*:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Na prvním řádku jsou uvedeny všechny možné slice-indexy 0...5 v řetězci 'HelpA', na druhém pak odpovídající záporné hodnoty. Řez od *i* do *j* je tedy tvořen všemi znaky mezi hranami označenými mezi hranami označenými *i* a *j*.

Pokud potřebujete zjistit délku určitého řetězce, jistě využijete interní funkci `len()`:

```
>>> s = 'supercalifragilisticexpialidociální'
>>> len(s)
35
```

Pro nezáporné slice-indexy je délka řezu rozdílem slice-indexů pokud oba "padnou" dovnitř řetězce. Například, délka řezu `word[1:3]` je 2.

3.1.3 Řetězce Unicode

Vydáním Pythonu 2.0 se jazyk dostal mezi skupinku jazyků podporujících Unicode. Od té doby Python umí pracovat s Unicode řetězci (viz http://www.unicode.org/) úplně stejným způsobem jako s obyčejnými řetězci. To umožňuje snadnou integraci Unicode do již existujících aplikací. Dokonce je možné díky konverzním funkcím snadno převádět obyčejné řetězce na Unicode a zpět.

Čím je Unicode tak pokrokové? Především v tom, že každému znaku libovolného jazyka přiřazuje jedinečný index. Tím se liší od dříve používaného schematu, kdy se používalo pouze 256 indexů a několik kódových tabulek, takže jednomu indexu odpovídalo více znaků (každý v jiné tabulce). To vedlo k velkým zmatkům, rovněž bylo nutné respektovat internacionalizaci[6] programů, což je zajištění správné funkce programu v různých národních prostředích (program akceptuje národní znaky, správně provádí třídění, konverzi řetězců apod.). Unicode proto definuje pouze jedinou kódovou stránku pro všechny jazyky. Program pak zachází se všemi znaky Unicode stejným způsobem a nepotřebuje rozlišovat mezi různými jazyky a kódovými stránkami.

Unicode řetězce můžeme zapisovat přímo ve zdrojovém kódu programu. Pouze před samotný řetězec vložíme prefix *u* (podobně jako u raw řetězců prefix r):

```
>>> u'Hello World !'
u'Hello World !'
```

Jak vidíme, Unicode řetězec se bez problémů vytvořil. Že se jedná o Unicode řetězec snadno poznáme podle malého písmena 'u' před řetězcem. Chcete-li do řetězce vložit speciální znak, můžete tak učinit díky *Unicode-Escape* módu. Nejlépe to uvidíte na následující ukázce:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

Escape sekvence `\u0020` znamená vložení Unicode znaku s hodnotou 0x0020 (znak mezera) na dané místo v řetězci.

Všechny ostatní znaky jsou interpretovány za použití jejich odpovídajících hodnot v kódové stránce Unicode. Jste-li obeznámeni s převodními tabulkami mezi jednotlivými národními kódovými stránkami a kódovou stránkou Unicode, jistě jste si všimli, že prvních 256 znaků Unicode přesně odpovídá všem 256 znakům kódové stránky Latin-1.

I s Unicode řetězci je možné používat raw mód s podobnou funkcí jako raw mód obyčejných řetězců. Tento mód aktivujeme použitím prefixu *ur* namísto standardního *u*. Python pak začne používat tzv. Raw-Unicode-Escape mód. V tomto módu Python nahrazuje pouze sekvence typu `\uXXXX` a ostatní (třeba `\n`) nechává tak jak jsou:

[6] Anglicky internationalization, často psáno jako '`i18n`' — '`i`' + 18 znaků + '`n`'. Vedle toho existuje i pojem lokalizace, localization ('`l10n`'), proces překladu původních textů programu do nového jazyka.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

Výhody raw módu oceníte především potřebujete-li zapsat větší množství zpětných lomítek (např. při zápisu regulérních výrazů).

Nehledě na tyto standardní zápisy nabízí Python ještě celou řadu způsobů, jak Unicode řetězce vytvořit.

Pro konverzi znaků z osmibitového kódování (klasické řetězce) do kódování Unicode můžete použít interní funkci `unicode()`, která umožňuje přístup ke všem registrovaným kodekům, které dokáží vytvořit Unicode řetězce z řetězce v téměř libovolném kódování (např. *Latin-1*, *ASCII*, *UTF-8* nebo *UTF-16*). Implicitní kódování je ASCII. To pro hodnoty znaků používá pouze 7 bitů (proto je možné používat pouze znaky v rozsahu 0 až 127). Při použití rozšířených znaků (např. znaky s diakritikou apod.) dojde k chybě.

Pro převod Unicode řetězce zpět na standardní lze s výhodou použít interní funkci `str()`. Ta používá implicitní kódování.[7]

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

Chceme-li při konverzi Unicode řetězce na 8bitový specifikovat kódování, v němž má cílový řetězec být, musíme použít metodu `encode()` Unicodového řetězce. Té předáme jediný argument — jméno kódování (platí, že všechna jména kódování se píší malými písmeny):

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Opačnou konverzi umožňuje již zmíněná funkce `unicode()`, které lze opět předat jediný argument — jméno kódování, ve kterém je původní osmibitový řetězec. Převodu výše uvedeného výsledku zpět dosáhneme tímto voláním:

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 Seznamy

Jazyk Python podporuje i další datové typy. Jde především o tzv. *složené* datové typy, které slouží k uložení jiných hodnot. Nejuniverzálnější z nich je *seznam*.

Seznam vznikne, zapíšeme-li výčet *libovolných* hodnot oddělených čárkou mezi hranaté závorky. Není nutné, aby všechny prvky seznamu měly stejný typ:

[7] Nastavuje se v souboru site.py umístěném v hlavním adresáři běhového prostředí (např. /usr/local/lib). Jeho jméno lze zjistit pomocí funkce `sys.getdefaultencoding()`

```
>>> a = [’spam’, ’eggs’, 100, 1234]
>>> a
[’spam’, ’eggs’, 100, 1234]
```

Seznamy podporují, podobně jako řetězce, mnoho dalších operací. Namátkou jmenujme spojování, násobení celým číslem, indexování, operace s podsekvencemi (slice) a další:

```
>>> a[0]
’spam’
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
[’eggs’, 100]
>>> a[:2] + [’bacon’, 2*2]
[’spam’, ’eggs’, ’bacon’, 4]
>>> 3*a[:3] + [’Boe!’]
[’spam’, ’eggs’, 100, ’spam’, ’eggs’, 100, ’spam’, ’eggs’, 100, ’Boe!’]
```

Zásadní rozdíl mezi řetězci a seznamy spočívá v možnosti změny prvků. Zatímco u řetězců to nepřipadá v úvahu — jde o typ *neměnný*, seznamy se jí nebrání — tento typ nazýváme *proměnný*. Seznam lze změnit pomocí přiřazení nového prvku na určitý index:

```
>>> a
[’spam’, ’eggs’, 100, 1234]
>>> a[2] = a[2] + 23
>>> a
[’spam’, ’eggs’, 123, 1234]
```

Rovněž lze přiřadit hodnotu určité subsekvenci. Takto dokonce můžeme změnit i počet prvků seznamu:

```
>>> # Změna prvků:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Jejich odstranění:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Vložení nových:
... a[1:1] = [’bletch’, ’xyzzy’]
>>> a
[123, ’bletch’, ’xyzzy’, 1234]
>>> a[:0] = a     # Vložení kopie seznamu do sebe sama
>>> a
[123, ’bletch’, ’xyzzy’, 1234, 123, ’bletch’, ’xyzzy’, 1234]
```

Podobně jako na řetězce můžeme i na seznamy aplikovat interní funkci `len()`. Jako její návratovou hodnotu pak obdržíme počet prvků obsažených v seznamu:

```
>>> len(a)
8
```

Někdy se může hodit i možnost vytvořit seznam obsahující jiné seznamy, třeba:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Viz. sekce 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

V posledním uvedeném příkladě si všimněte jedné věci: objekty `p[1]` a `q` jsou jeden a týž seznam. Změnou jednoho se změní i druhý. To umožňuje mechanismus *odkazů na objekty*. Později se k nim ještě vrátíme, již nyní ale můžeme prozradit, že se jedná o velice důležitou problematiku a bez její znalosti Python těžko pochopíte.

3.2 První kroky

Jazyk Python lze samozřejmě použít k daleko komplikovanějším úlohám. Například můžeme vypsat počáteční prvky *Fibonacciho rozvoje*[8]:

```
>>> # Fibonacci rozvoj:
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Tento příklad demonstruje několik pro nás v tuto chvíli nových vlastností:

- První řádek kódu ukazuje použití *vícenásobného přiřazení*. Oběma proměnným `a` a `b` jsou *zároveň* přiřazeny dvě nové hodnoty. Na poslední řádce ukázky je tento obrat použit znovu pro získání nové dvojice čísel.
- Výrazy na pravé straně přiřazení jsou vyhodnoceny ještě před přiřazením hodnoty proměnné. Vyhodnocení výrazu probíhá zleva doprava.
- Cyklus `while` je ukázkou složeného příkazu, tj. příkazu který obsahuje jiné příkazy. Tyto příkazy se pak nazývají *tělo cyklu*. Cyklus typu while běží dokud podmínka (v našem případě `b < 10`) zůstává pravdivá. Test použitý v tomto případě (`b < 10`) je ukázkou jednoduchého porovnání. Paleta standardních porovnávacích operátoru je shodná s jazykem C: `<` (menší než), `>` (větší než), `==` (rovno), `<=` (menší nebo rovno), `>=` (větší nebo rovno) a `!=` (nerovno). Výsledkem porovnání je podobně jako v C číslo. Jako podmínku však můžeme použít

[8] Fibonacciho rozvoj je určen rovnicí $r[n+2] = r[n] + r[n+1]$. Začíná tudíž čísly 1, 1, 2, 3, 5, 8, 13 atd.:

libovolnou hodnotu. Jakékoli nenulové číslo znamená pravdivou hodnotu, nula nepravdivou. Podmínkou může být dokonce i jiný typ než číslo (například libovolná sekvence).

- Tělo cyklu je odsazeno. Odsazení od kraje je geniální způsob, jakým Python vyznačuje vnořené bloky kódu. Interpretr Pythonu nedisponuje (zatím!) inteligentním editováním příkazového řádku, takže tabulátory nebo mezery musíte psát ručně. Pokud ovšem budete připravovat složitější zdrojový kód, jistě použijete nějaký kvalitní textový editor, přičemž pro většinu z nich existuje speciální mód, který bloky kódu odsazuje zcela automaticky.[9] Zadáváte-li složený příkaz v interaktivním módu, musíte jej ukončit prázdnou řádkou, která indikuje konec příkazu (parser jinak nemůže zjistit, zda jste se již rozhodli blok ukončit). Každý řádek v daném bloku musí být odsazen o stejný počet mezer či tabulátorů, a nelze je kombinovat (např. na prvním řádku 2 mezery a 1 tabulátor a na dalším 1 tabulátor a následně 2 mezery).

- Příkaz `print`, který vytiskne hodnotu výrazu, který jste mu předali. Můžete použít i více výrazů oddělených čárkou, pak budou jednotlivé hodnoty odděleny mezerou. Všechny řetězce budou samozřejmě vytisknuty bez jejich vnějších uvozovek:

```
>>> i = 256*256
>>> print 'Hodnota proměnné i je', i
Hodnota proměnné i je 65536
```

Pokud výčet výrazů ukončíte čárkou, nedojde k vložení znaku nového řádku, další příkaz `print` tedy bude pokračovat na tomtéž řádku.

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>
```

Jestliže jste ale v interaktivním módu, interpretr před zobrazením další výzvy ukončí předchozí řádek, výzva pak je zobrazena v prvním sloupci terminálu.

[9] Pro oba nejlepší editory *vim* a *emacs* tyto módy samozřejmě existují.

KAPITOLA

ČTVRTÁ

Příkazy pro řízení toku programu

Podobně jako jiné jazyky, i Python obsahuje příkazy pro řízení toku programu. Kromě již zmíněného příkazu `while` představeného v minulé kapitole jde o další neméně významné konstrukce `if`, `for`, `break`, `continue` a především *uživatelsky definované funkce*. Těm všem je věnována tato kapitola.

4.1 Konstrukce `if`

Nejznámějším, nejjednodušším a nejpoužívanějším příkazem pro řízení toku programu je pravděpodobně příkaz `if`. S jeho pomocí lze zrealizovat jakýkoli jiný příkaz pro řízení toku. Implementuje nejjednodušší rozhodovací mechanismus, je-li nějaká podmínka pravdivá, vykoná určitý blok kódu. Časem se ujaly i rozšíření typu větví `else` a `elif`.

```
>>> x = int(raw_input("Zadejte celé číslo: "))
>>> if x < 0:
...      x = 0
...      print 'Záporné číslo změněno na nulu'
... elif x == 0:
...      print 'Nula'
... elif x == 1:
...      print 'Jedna'
... else:
...      print 'Více'
...
```

Příkaz `if` otestuje řídící podmínku a pokud je pravdivá, spustí svoje tělo. Pokud jde o jednoduchou podmínku (bez větví `else`), je v případě nepravdivosti podmínky řízení předáno na příkaz následující po těle příkazu `if`.

Pokud je příkaz `if` složen i z větví `elif`, je v případě nepravdivosti řídící podmínky otestována další podmínka v první větvi `elif`. Pokud je pravdivá, vykoná se tělo příslušející této větvi. Je-li nepravdivá, pokračuje se stejně s dalšími větvemi `elif`. V každém případě se ale po vykonání libovolného těla předá řízení až za poslední větev `elif` (případně `else`).

Příkaz `if` může mít i maximálně jednu větev `else`. Ta je spuštěna pokud nevyhovuje ani jedna řídící podmínka u větví `if` a `elif`.

Větev `elif` by se dala nahradit i větví `else` a dalším příkazem `if`, šlo by však o zbytečnou komplikaci, a především proto se ujala větev `elif` ("`elif`" je zkratkou pro "else if"). Jazyk Python neobsahuje příkaz typu `switch`, plně ho totiž nahrazuje příkaz `if` … `elif` … `elif` … .

4.2 Konstrukce `for`

Příkaz `for` v jiných jazycích jsou poněkud odlišné od toho, jak jej poznáme v jazyce Python. Od toho v jazyce Pascal je naprosto odlišný, Pascal umožňuje řídící proměnné přiřazovat hodnoty v určitém rozsahu čísel, nanejvýš bylo možné specifikovat krok mezi jednotlivými hodnotami. Jazyk C již nabízí tři části, které se provedou — první před samotným spuštěním cyklu, druhá určuje podmínku pro ukončení cyklu a třetí se opakuje po každém cyklu. To nabízí programátorovi široké možnosti, ne každý je však dokáže využít.

Python šel jinou cestou, obvyklou v interpretovaných jazycích. Příkaz `for` umožňuje iterovat prvky libovolné sekvence (připomeňme, že sekvencí jsou kromě seznamů i řetězce). Za řídící proměnnou se postupně dosazují všechny prvky sekvence v tom pořadí, v jakém jsou v sekvenci uloženy. K ukončení cyklu dojde po vyčerpání všech prvků (případně příkazem `break` nebo neodchycenou výjimkou):

```
>>> # Vytisknutí délky řetězců:
... a = [’kočka’, ’okno’, ’defenestrace’]
>>> for x in a:
...     print x, len(x)
...
kočka 5
okono 4
defenestrace 12
```

Během cyklu není bezpečné modifikovat řídící sekvenci, by mohlo dojít buď k vynechání nebo opakování prvku (to platí pouze pro proměnné sekvenční typy, u neměnných to nelze již z jejich principu). Potřebujeme-li přesto změnit pořadí prvků v seznamu přes nějž iterujeme, musíme iterovat přes prvky kopie původního seznamu. Kopii seznamu snadno a rychle vytvoříme pomocí subsekvence s vynechanými indexy:

```
>>> for x in a[:]: # vytvoření kopie celého seznamu
...    if len(x) > 6: a.insert(0, x)
...
>>> a
[’defenestrace’, ’kočka’, ’okno’, ’defenestrace’]
```

4.3 Funkce `range()`

Je-li třeba iterovat přes prvky aritmetické posloupnosti, budeme potřebovat interní funkci `range()`, která vrací tuto posloupnost jako klasický seznam. Funkci `range()` předáme konečnou hodnotu posloupnosti. Pro všechny prvky pak bude platit, že jsou menší než tato hodnota:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Programátor si rovněž může vyžádat, že chce aby posloupnost začínala jiným číslem než nulou, případně lze zvolit i jiný přírůstek než jedna (včetně záporných čísel):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Chceme-li prvky aritmetické posloupnosti použít jako indexy do nějaké sekvence, je velmi vhodné zkombinovat funkce `range()` a `len()`:

```
>>> a = ['Micinko', 'Lízinko', 'čičí', 'vylezte']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Micinko
1 Lízinko
2 čičí
3 vylezte
```

4.4 Podmínky

Python podporuje mnohem více operátorů psaní podmínek než třeba jazyk C. Kromě operátorů porovnání je možné použít i další. Především jde o test na přítomnost/nepřítomnost prvku v určité sekvenci — to zajišťují operátory `in` a `not in`.

Další dvojice operátorů (`is` a `not is`) porovnává dva objekty a vrátí logickou jedničku pokud jde (resp. nejde) o ty samé objekty (musí se rovnat nejen jejich hodnota, ale i umístění v paměti atd.).

Všechny operátory porovnání mají stejnou prioritu, která je nižší než priorita všech numerických operátorů, proto je možné psát výraz `x + 3 > y` bez uzávorkování.

Porovnání může být zřetězeno. Například výraz `a < b == c` testuje, jestli `a` je menší než `b` a zároveň `b` je rovno `c`.

Jednotlivé podmínky můžeme kombinovat použitím Booleových operátorů `and` a `or`. Libovolný logický výraz můžeme znegovat operátorem `not`. Všechny Booleovské operátory mají nejnižší prioritu ze všech operátorů (z nich pak `not` má nejvyšší, následuje `and` a `or`), takže `A and not B or C` je totéž, co `(A and (not B)) or C`. Závorky je možné samozřejmě použít na libovolném místě pro úpravu priority operátorů.

Booleovské operátory `and` a `or` mají v Pythonu tzv. *zkrácené* vyhodnocování, jejich argumenty jsou vyhodnocovány zleva doprava a pokud je možné definitivně určit výsledek výrazu, dojde k ukončení vyhodnocení. Z toho vyplývá, že pokud `A` a `C` jsou pravdivé, ale `B` nepravdivý, výraz `A and B and C` nevyhodnotí výraz `C`.

Výsledek porovnání je možné přiřadit libovolné proměnné. Například:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

V Pythonu, narozdíl od C, se nemůže vyskytovat přiřazení uvnitř výrazu. Programátorům v C to může vadit, ale Python se tímto vyhýbá mnoha chybám, ke kterým dojde zapsáním `=` ve výrazu namísto `==`.

4.5 Příkazy `break` a `continue` a větev `else` příkazu `for`

Zároveň s cykly je třeba zmínit i příkazy pro jejich řízení — `break` a `continue`. Oba dva jistě znáte i z jiných programovacích jazyků.

Příkaz `break` ukončí aktuální cyklus `for` nebo `while` a vrátí řízení za tento příkaz. Naproti tomu příkaz `continue` způsobí pokračování další iterací. Zbytek těla za tímto příkazem je ignorován a program se vrátí zpět k řídící části a znovu otestuje řídící podmínku (cyklus `while`) nebo pokračuje dosazením dalšího prvku za řídící proměnnou (cyklus `for`).

Oba typy cyklů mohou mít také větev `else`. K jejímu spuštění dojde vždy při "regulérním" ukončení cyklu, tj. stane-li se řídící podmínka nepravdivou (cyklus `while`) případně vyčerpají-li se všechny prvky řídící sekvence (cyklus `for`). V případě ukončení cyklu příkazem `break` nebo neodchycenou výjimkou se větev `else` *nevykonná*! Toto chování demonstruje ukázka — hledání prvočísel menších než deset:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, 'je rovno', x, '*', n/x
...            break
...     else:
...          # cyklus proběhl bez nalezení dělitele
...          print n, 'je prvočíslo'
...
2 je prvočíslo
3 je prvočíslo
4 je rovno 2 * 2
5 je prvočíslo
6 je rovno 2 * 3
7 je prvočíslo
8 je rovno 2 * 4
9 je rovno 3 * 3
```

4.6 Příkaz `pass`

Příkaz `pass` je trochu neobvyklý. Jedná se o prázdný příkaz, nedělá vůbec nic. Používá se obzvláště na místech, kde jazyk syntakticky vyžaduje nějaký příkaz, ale programátor žádnou akci nepožaduje:

```
>>> while 1:
...       pass # činné čekaní na přerušení
...
```

4.7 Definování funkce

V Pythonu, stejně jako v jiných jazycích, můžeme používat množství funkcí. Každý jazyk také umožňuje programátorovi definovat si své vlastní funkce. Do těchto *uživatelsky definovaných funkcí* lze soustředit izolované kusy kódu s určitým rozhraním. Například si můžeme napsat funkci, jenž vytiskne Fibonacciho rozvoj do zadané meze:

```
>>> def fib(n):    # vypiš Fibonacci rozvoj do n
...     """Vytiskne Fibonacciho rozvoj do n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Nyní zavoláme funkci tak, jak jsme si ji definovali:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Definice funkce je uvozena klíčovým slovem `def`. Toto klíčové slovo následuje jméno funkce a výčet formálních argumentů. Následující příkazy, tvořící tělo funkce, jsou zapsány na dalších řádkách. Tělo funkce je blok kódu a proto musí být odsazeno podobně jako tělo cyklu apod. Nepovinnou součástí těla funkce může být i tzv. *dokumentační řetězec*. Tento řetězec je uvedený bezprostředně po hlavičce funkce.

Existuje mnoho nástrojů, které používají dokumentační řetězce k automatickému generování dokumentace. Některé z nich umožňují uživateli interaktivně procházet kód programu, přičemž pro lepší orientaci zobrazují právě tyto dokumentační řetězce. Je proto dobrým zvykem psát dokumentační řetězce ke každé funkci, kterou napíšete. Kromě funkce lze dokumentační řetězce uvádět i u dalších objektů — u tříd, metod nebo modulů.

Zavoláním funkce se vytvoří nový prostor jmen používaný pro lokální proměnné této funkce. To znamená, že všechna přiřazení uvnitř těla funkce jdou do tohoto lokálního prostoru jmen. To zabraňuje přímému přiřazení hodnoty globální proměnné (tj. proměnné definované mimo funkci). Použijeme-li některou proměnnou v libovolném výrazu, je její jméno hledáno nejprve v lokálním oboru jmen a teprve pokud tam není nalezeno, je prohledán globální obor jmen (a pokud není nalezeno ani tam je nakonec prohledán interní obor jmen, není-li toto jméno nalezeno vůbec, dojde k výjimce). Je důležité podotknout, že přestože globální proměnné není možné z těla funkce modifikovat, je stále možné je číst. (Pokud deklarujeme v těle funkce nějakou proměnnou jako globální příkazem `global`, je možné jí takto používat, čili je možná i její modifikace kódem funkce).

Skutečné argumenty předané funkci při jejím zavolání jsou uloženy v lokálním prostoru jmen této funkce. Argumenty jsou předávány *hodnotou*.[1] Jestliže volaná funkce zavolá jinou funkce, je opět pro toto volání vyhrazen nový prostor jmen. To zabrání vzájemnému ovlivňování funkcí, které by zákonitě nastalo, pokud by sdílely stejný prostor jmen.

Definice funkce vytvoří novou proměnnou, jejíž jméno je stejné jako jméno funkce a je uloženo v lokálním prostoru jmen. Její hodnota má speciální typ — uživatelsky definovaná funkce. Tato hodnota může být přiřazena jiné proměnné, kterou poté lze použít pro volání této funkce. Tímto způsobem lze libovolný objekt přejmenovat.

Zde se opět uplatňuje mechanismus *odkazů na objekty*. Objekty existují nezávisle na svých jménech. Jméno objektu je naprosto irelevantní, jeden objekt může vystupovat pod různými jmény. Dokonce objekt nemusí být pojmenovaný vůbec, to nastává v případě, že objekt se stane součástí třeba seznamu. Jména objektů slouží pouze pro jednoduchou orientaci programátora, jednoduše pojmenovávají objekty a umožňují snadný přístup k nim.

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Programátoři v jazyce Pascal zřejmě namítnou, že funkce `fib` není funkcí, ale procedurou. Python, podobně jako C, považuje procedury za speciální druh funkce, která nevrací žádnou hodnotu. Ve skutečnosti procedury jazyka Python vrací hodnotu `None`. Tato hodnota je obdobou hodnoty `null` v jazyce C. Její vypsání je ale potlačeno interpretrem, pokud tuto návratovou hodnotu chcete vidět, musíte si o to říci sami:

[1] Slovo *hodnota* vždy znamená *odkaz na objekt*, ne přímo hodnotu objektu. Předáme-li totiž funkci proměnný objekt a tato funkce ho nějakým způsobem modifikuje, volající funkce "uvidí" všechny tyto změny.

```
>>> print fib(0)
None
```

Velice jednoduše lze napsat funkci, která, místo aby vytiskla výsledek, vrátí Fibbonacciho rozvoj jako seznam čísel:

```
>>> def fib2(n): # vrátí Fibonacciho rozvoj do čísla n
...     """Vrátí seznam obsahující Fibonacciho rozvoj do čísla n."""
...     vysledek = []
...     a, b = 0, 1
...     while b < n:
...         vysledek.append(b)    # viz níže
...         a, b = b, a+b
...     return vysledek
...
>>> f100 = fib2(100)    # zavoláme funkci fib2()
>>> f100                # a vytiskneme její výsledek
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Tato ukázka nám názorně ukazuje další nové vlastnosti jazyka Python:

- Vrácení hodnoty zajistí příkaz `return`, který ukončí provádění funkce a jako její návratovou hodnotu použije výraz uvedený za slovem return.

  Samotný příkaz `return` vrátí `None` podobně jako vykonání všech příkazů těla funkce.

- Příkaz `vysledek.append(b)` znamená zavolání *metody* seznamu `vysledek`. Metoda je zvláštní funkce, která určitým způsobem "patří" k určitému objektu. Její funkcí je většinou nějak modifikovat samotný objekt. Proto každý objekt má svoje vlastní metody, každá modifikuje pouze svůj vlastní objekt a ostatní nechá na pokoji. Různé objekty mohou obsahovat různé metody. Dokonce lze, aby metody různých objektů měly stejná jména a přitom prováděly naprosto jinou činnost. Za použití *tříd*[2] je možné definovat vlastní typy objektů a jejich metody. Metodu `append()` definují objekty typu seznam. Ta při svém vykonání přidá na konec seznamu prvek, který jí byl předán jako jediný argument. Odpovídá výrazu '`vysledek = vysledek + [b]`' ale je mnohem efektivnější.

4.8 Další možnosti při definici funkce

Funkci lze definovat s mnoha různými argumenty. Python nabízí i několik speciálních zápisů, které programátorovi značně usnadňují život. Navíc všechny tyto speciální zápisy mohou být libovolně mezi sebou kombinovány.

4.8.1 Implicitní hodnoty argumentů

Velice užitečné je specifikování implicitních hodnot argumentů. Při volání funkce je pak možné tyto argumenty vynechat. V tomto případě jim bude zcela automaticky přiřazena implicitní hodnota. Typickým příkladem může být funkce, jenž zobrazí výzvu a čeká na odpověď uživatele:

[2] Viz. kapitola 9.

```
def ano_ne(vyzva, opakovani=4, upozorneni='Ano nebo ne, prosím'):
    while 1:
        ok = raw_input(vyzva)
        if ok in ('a', 'an', 'ano'): return 1
        if ok in ('n', 'ne'): return 0
        opakovani = opakovani - 1
        if opakovani < 0: raise IOError, 'uživatel ignorant'
        print upozorneni
```

Tato funkce může být volána jako `ano_ne('Chcete opravdu ukončit program?')` případně jako `ano_ne('Přejete si prepsat soubor?', 2)`.

Implicitní hodnoty argumentů jsou vyhodnoceny při definici funkce, přičemž jako jmenný prostor (místo, kde se vyhledávají jména proměnných) je použit prostor jmen definujícího bloku kódu takže kód

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

vytiskne číslo `5`.

Implicitní hodnoty argumentů jsou vyhodnoceny pouze jednou, i při opakovaném volání funkce se používá stále tentýž objekt. To má význam v případě proměnných objektů. Názornou ukázkou může být funkce shromažďující argumenty, které jí byly předány při jednotlivých voláních, načež vrátí seznam těchto argumentů:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Předchozí blok kódu potom postupně vytiskne následující výstup:

```
[1]
[1, 2]
[1, 2, 3]
```

V některých případech však může být toto sdílení hodnot mezi voláními překážkou. Řešením tohoto problému může být použití nějaké známé implicitní hodnoty argumentu (třeba `None`) a následné testování hodnoty argumentu na tuto hodnotu. Pokud je argument roven známé implicitní hodnotě, přiřadí se mu nová hodnota (proměnný objekt, u něhož si nepřejeme sdílení), není-li argument roven této hodnotě, použije se předaná hodnota:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.8.2 Keyword argumenty

Pokud používáme implicitní hodnoty argumentů, často potřebujeme specifikovat hodnotu pouze jednoho argumentu. Pokud je tento argument první ze všech, je vše bez problémů. Potíž nastává v případě, že chceme specifikovat hodnotu jiného argumentu.

Samozřejmě by se nechalo do volání vepsat implicitní hodnoty předcházejících argumentů. Problém by však nastal v případě změny hodnoty jednoho z implicitních argumentů. Následné dohledání všech volání by bylo velice těžké.

Python proto může přiřadit hodnotu určitému formálnímu argumentu. Jednoduše do volání funkce napíšeme jméno tohoto argumentu následované rovnítkem a jeho hodnotou (čili ‘`funkce(`*jméno_formálního_parametru* `=` *hodnota*`)`’. V tomto případě se formálnímu argumentu přezdívá keyword argument. Keyword argumenty můžeme použít i v případě, kdy nepoužíváme implicitní hodnoty argumentů.

Pokud kombinujeme keyword a klasické (poziční) argumenty, je třeba důsledně dbát na to, aby poziční argumenty byly uvedeny před keyword argumenty. Zároveň je třeba splnit požadavek předání hodnot všem argumentům, které nemají implicitní hodnotu (tj. jde o povinné argumenty). Použijeme-li neznámý keyword argument, dojde při volání funkce k chybě. Je třeba se také vyhnout předání hodnoty jednomu argumentu dvakrát (jednou jako pozičnímu argumentu, podruhé jako keyword argumentu).

Následuje příklad použití keyword argumentů. Funkci definovanou jako

```
def papousek(napeti, xxx=’Python’, akce=’zpívat’, keczy=’bla, bla, bla’):
    print "-- Tento papoušek nebude", akce,
    print "když do něj pustíte", napeti, "Voltů."
    print "-- A nějaké kecy:", keczy
    print "-- A ještě něco:", xxx, "!"
```

můžeme volat následujícími způsoby:

```
papousek(1000)
papousek(akce = ’řvát’, napeti = 1000000)
papousek(’tisíc’, xxx = ’Chudák papouch’)
papousek(’milion’, ’Papoušek to nerozchodí’, ’skákat’)
```

Všechna následující volání jsou ale chybná:

```
papousek()                      # chybějící povinný argument
papousek(napeti=5.0, ’ahoj’)    # povinný argument následující po keyword
papousek(110, napeti=220)       # hodnota argumentu předána dvakrát
papousek(vrah=’Pepa Zdepa’)     # neznámý keyword argument
```

4.8.3 Funkce s proměnným počtem argumentů

Pokud při volání funkce specifikujeme více pozičních argumentů, než samotná funkce požaduje, dojde při jejím volání k chybě. Proto je možné v hlavičce funkce specifikovat speciální formální argument s předponou `*` (např. `*arg`). Tomuto argumentu budou přiřazeny všechny přebývající argumenty, předané při volání funkce (typem tohoto argumentu je tuple). Pokud žádné takové argumenty nejsou předány, nabude tento argument hodnotu prázdné tuple.[3]. Funkce, jenž tuto možnost určitě využije, je obdoba céčkovské funkce `fprintf`:

```
def fprintf(soubor, format, *argumenty):
    file.write(format % argumenty)
```

Kromě argumentu typu `*arg` je možné použít i argument ve tvaru `**kwarg`, kterému bude přiřazeno asociativní pole keyword argumentů, jejichž jména neodpovídají definici funkce. Pak můžeme funkci definovanou jako

```
def big_burger(druh, *argumenty, **keyword):
    print "-- Máte", druh, '?'
    print "-- Promiňte,", druh, "došly."
    for arg in argumenty: print arg
    print '-'*40
    for kw in keyword.keys(): print kw, ':', keyword[kw]
```

volat následujícími způsoby:

```
big_burger('cheesburgery', "Velice se omlouvám, pane.",
           "Opravdu se velice se omlouvám, pane.",
           zakaznik='Adam',
           prodavac='Bedřich',
           obchod='Honzův domácí BigBurger')
```

Jejím výstupem pak bude text

```
-- Máte cheesburgery ?
-- Promiňte, cheesburgery došly.
Velice se omlouvám, pane.
Opravdu se velice se omlouvám, pane.
----------------------------------------
obchod : Honzův domácí BigBurger
prodavac : Bedřich
zakaznik : Adam
```

Oba speciální typy argumentů lze kombinovat, pak musí argument `*arg` předcházet argumentu `**kwarg`.

4.8.4 Lambda funkce

Často je potřeba napsat funkci, která vykonává velice jednoduchý příkaz. Ve funkcionálních programovacích jazycích (a nejen v nich) se proto ujaly tzv. *lambda funkce*, tj. krátké funkce, většinou bezejmenné, určené pro vykonávání jednoduchých příkazů.

V Pythonu je možné lambda funkce také používat. Nejjednodušší ukázkou může být funkce vracející součet dvou argumentů: '`lambda a, b: a+b`'. Lambda funkce v Pythonu je výraz, proto, abychom zachovali její hodnotu, musíme jí přiřadit nějaké proměnné (případně předat nějaké funkci apod.):

[3] Více o tuple v 5.3 kapitole.

```
>>> soucet = lambda a, b: a+b
>>> print soucet(1, 2)
3
```

Lambda funkce v Pythonu jsou tvořeny klíčovým slovem `lambda`, výčtem argumentů, dvojtečkou a samotným tělem funkce. Je třeba podotknout, že tělo funkce je tvořeno jediným výrazem. Ten je při zavolání funkce vyhodnocen a jeho hodnota je zároveň návratovou hodnotou lambda funkce. Toto omezení je třeba respektovat, v těle lambda funkce *nelze* použít příkazy typu `print` atd. Podobně jako je tomu u vložených funkcí, i lambda funkce se mohou odkazovat na proměnné nadřazené funkce:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

4.8.5 Dokumentační řetězce

Jak jsme si již řekli, je dobrým zvykem psát dokumentační řetězce ke každé funkci, modulu nebo třídě, které vytvoříme. Nyní si povíme o tom jak psát a formátovat dokumentační řetězce.

Na prvním řádku dokumentačního řetězce by měla být krátká věta shrnující účel celého objektu. Neměla by explicitně pojmenovávat objekt, jeho jméno je přístupné jinou cestou. Jako každá věta by měla začínat velkým písmenem a končit tečkou.

Je-li třeba rozsáhlejší dokumentační řetězec, je možné použít víceřádkový řetězec. V tomto případě by měla být druhá řádka prázdná, oddělující souhrn na první řádce od zbytku popisu. Ten by může být tvořen i více odstavci. Ty se mohou týkat volajících konvencí používaných u objektu, vedlejších efektů tohoto objektu apod.

Parser interpretru neodstraňuje (!) odsazení z víceřádkových řetězců, proto by měly nástroje, které generují dokumentaci z dokumentačních řetězců, toto odsazení odstranit. Přitom lze použít jednoduchý postup, nejprve se nahradí tabulátory odpovídajícím počtem mezer (8). Poté se určí odsazení. Protože z prvního řádku řetězce toto odsazení nelze zjisti, použije se první neprázdný řádek dokumentačního řetězce. O toto zjištěné odsazení se poté zkrátí každý řádek řetězce. Pokud se v dokumentačním řetězci nachází řádek, který je odsazen méně, než činí průběžné odsazení, jsou z jeho začátku odsazeny všechny bílé znaky.

Zde je příklad víceřádkového dokumentačního řetězce, jehož by se případná úprava týkala. Jak vidíte, druhý řádek textu je opravdu odsazen:

```
>>> def moje_funkce():
...     """Nic nedělá, ale zdokumentujeme ji.
...
...     Nedělá opravdu nic.
...     """
...     pass
...
>>> print moje_funkce.__doc__
Nic nedělá, ale zdokumentujeme ji.

    Nedělá opravdu nic.
```

KAPITOLA

PÁTÁ

Datové struktury

Tato kapitola bude věnována dalším datovým typům jazyka Python. Jmenovitě si budeme povídat o *seznamech*, *tuple* a *slovnících*, přičemž si dále rozšíříme znalosti syntaxe různých příkazů.

5.1 Seznamy

O seznamech jsme se již zmínili ve čtvrté kapitole, nyní si naše znalosti poněkud rozšíříme. Následuje popis všech metod všech seznamů, metoda je de facto funkce, která se určitým způsobem váže na určitý objekt, při svém zavolání modifikuje pouze tento objekt:

`append(x)` Přidá prvek na konec seznamu. Tato metoda je ekvivalentní zápisu `a[len(a):] = [x]`.

`extend(L)` Na konec seznamu přidá všechny prvky seznamu L, je ekvivalentní zápisu `a[len(a):] = L`.

`insert(i, x)` Vloží prvek `x` na pozici `i`. Argument `i` znamená index prvku, před který se má nová položka vložit, tudíž `a.insert(0, x)` vloží prvek na začátek seznamu, zatímco `a.insert(len(a), x)` na konec (jde o totéž jako `a.append(x)`.

`remove(x)` Ze seznamu odstraní daný prvek `x`, pokud prvků rovných `x` se v seznamu nachází více, odstraní se první jeho výskyt. Nenajde-li metoda prvek `x`, dojde k výjimce.

`pop([i])` Odstraní prvek na pozici `i` a vrátí jeho hodnotu. Argument `i` je nepovinný, jeho vynecháním dojde k odstranění posledního prvku seznamu.

`index(x)` Vrátí index prvního prvku seznamu, jehož hodnota je rovna `x`. Není-li prvek nalezen, dojde k výjimce.

`count(x)` Vrátí počet všech výskytů prvků, jejichž hodnota je rovna `x`. Není-li nalezen žádný prvek, vrátí nulu.

`sort()` Seřadí prvky seznamu podle velikosti, přičemž modifikuje původní seznam.[1]

`reverse()` Zrevertuje seznam — první prvek se stane posledním, druhý předposledním atd. Změna se děje opět na původním seznamu.

Následuje příklad ukazující použití jednotlivých metod seznamu.

[1] V anglickém originále je tato vlastnost nazývána "in place sorting"

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Zásobníky

Pomocí metod `append()` a `pop()` lze ze seznamů vytvořit zásobníky. Zásobník je fronta typu LIFO[2] — poslední přidaný prvek bude odebrán jako první. Pro přidání prvku na vrchol zásobníku použijeme metodu `append()`, pro odebrání prvku metodu `pop()`. Připomeňme si, že metoda `pop()` bez dalších argumentů vrátí poslední prvek seznamu, čili prvek na vrcholu zásobníku:

```
>>> zasobnik = [3, 4, 5]
>>> zasobnik.append(6)
>>> zasobnik.append(7)
>>> zasobnik
[3, 4, 5, 6, 7]
>>> zasobnik.pop()
7
>>> zasobnik
[3, 4, 5, 6]
>>> zasobnik.pop()
6
>>> zasobnik.pop()
5
>>> zasobnik
[3, 4]
```

5.1.2 Fronty

Obdobou zásobníků jsou i fronty FIFO[3] — první vložený prvek je odebrán jako první. Pro přidání prvku na konec fronty použijeme opět metodu `append()`, pro odebrání prvního prvku pak metodu `pop()`, které předáme index `0` (ta tudíž odstraní a vrátí první prvek celého seznamu):

[2] Anglicky *last in — first out*
[3] *first in — first out*

```
>>> fronta = ["Python", "Perl", "PHP"]
>>> fronta.append("Ruby")
>>> fronta.append("Lisp")
>>> fronta.pop(0)
'Python'
>>> fronta.pop(0)
'Perl'
>>> fronta
['PHP', 'Ruby', 'Lisp']
```

5.1.3 Funkcionální programování v jazyce Python

Python se snaží maximálně ulehčit práci programátorovi, proto již od samého začátku obsahuje některé rysy funkcionálních programovacích jazyků. Jde především o lambda funkce a stručné seznamy.

Lambda funkce lze *velmi* výhodně používat v součinnosti s funkcemi pro manipulaci se seznamy. Tyto funkce přebírají několik argumentů, z nichž jeden je vždy funkce (většinou lambda funkce) a další seznam.

Začněme funkcí `filter()`. Její definice vypadá takto: '`filter(`*funkce*`, `*sekvence*`)`'. Po zavolání s těmito argumenty vrátí sekvenci[4] stejného typu jako je *sekvence*. Tato vrácená sekvence bude obsahovat pouze ty prvky původní *sekvence*, pro které má volání '`funkce(`*prvek*`)`' pravdivou hodnotu. Takto lze třeba napsat konstrukci, která vybere čísla, která *nejsou* dělitelná dvěma ani třemi:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

Naproti tomu funkce '`map(`*funkce*`, `*sekvence*`)`' vrátí seznam prvků. Tento seznam je vytvořen z návratových hodnot volání '`funkce(`*prvek*`)`'. Funkce `map()` vlastně přemapuje prvky původní sekvence na nové hodnoty (od tud pochází i její název). Pro výpočet třetích mocnin lze tedy napsat tuto konstrukci:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Funkci `map()` můžeme předat i více sekvencí, *funkce* pak musí mít tolik argumentů, kolik je sekvencí. Během jediného volání pak dostane prvky všech sekvencí najednou. Pokud je jedna sekvence kratší než druhá, předá se místo jejího prvku hodnota `None`.

Další variantou volání funkce `map()` je vynechání funkce. To se provede použitím hodnoty `None`. V tomto případě je použita interní funkce vracející všechny své argumenty jako *n-tici*.

Pokud zkombinujeme tyto dva speciální případy, získáme volání '`map(None, `*seznam1*`, `*seznam2*`)`', které převede dvojici seznamů na seznam dvojic.

[4] Více o sekvencích dále v této kapitole

```
>>> seq = range(8)
>>> def mocnina(x): return x*x
...
>>> map(None, seq, map(mocnina, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

Konečně poslední funkce `reduce()` s hlavičkou '`reduce(`*funkce*`, `*sekvence*`)`', která vrací *jedinou* hodnotu určitým způsobem reprezentující celou sekvenci. Tuto hodnotu ovlivňuje právě předaná `funkce`. Té jsou nejprve předány první dva prvky sekvence, ona pro ně vrátí výsledek a následně je opět zavolána tato funkce, předá se jí návratová hodnota předchozího volání spolu s následujícím prvkem atd. Poslední návratová hodnota `funkce` je použita jako návratová hodnota celé funkce `reduce()`. Jako příklad může sloužit konstrukce pro součet všech čísel 1 až 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Funkce `reduce()` ošetřuje ještě dva speciální stavy — obsahuje-li sekvence jediný prvek, je vrácena hodnota tohoto prvku. Je-li sekvence dokonce prázdní dojde k výjimce.

Nakonec si povíme ještě o nepovinném argumentu funkce `reduce()`, hodnota tohoto argumentu je totiž použita jako počáteční hodnota a `funkce` je poprvé volána s argumenty počáteční hodnota a první prvek a následuje obvyklý postup. Pokud je sekvence prázdná, je vrácena ona počáteční hodnota:

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

5.1.4 Stručný seznam

Aby se Python vyhnul nadměrnému používání funkcí `map()` a `filter()`, zavedla se konstrukce nazývaná *stručný seznam*.

Pomocí stručného seznamu je možné definovat seznam mnohem čistěji, než je tomu v případě výše uvedených funkcí. Každý stručný seznam se skládá z *výrazu* následovaného klíčovým slovem `for` a *sekvencí*. Při vyhodnocení stručného seznamu se prochází *sekvencí* a její prvky se dosazují do *výrazu*. Prvky nového seznamu pak tvoří hodnoty tohoto *výrazu*:

```
>>> ovoce = [’  banán’, ’  rajče ’, ’protlak  ’]
>>> [zbran.strip() for zbran in ovoce]
[’banán’, ’rajče’, ’protlak’]
>>> vec = [2, 4, 6]
>>> [x*3 for x in vec]
[6, 12, 18]
```

Za sekvencí je možné uvést ještě klauzuli `if` následovanou *podmínkou*. Pak jsou ze sekvence použity pouze ty prvky, které vyhovují *podmínce*.

```
>>> [x*3 for x in vec if x > 3]
[12, 18]
>>> [x*3 for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec] # CHYBA - jsou vyžadovány závorky
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

5.2 Příkaz `del`

Na začátku této kapitoly jsme se dověděli od metodě `remove()`, která odstraní určitý prvek ze seznamu. Někdy potřebujeme odstranit prvek na určité pozici v seznamu, pak použijeme právě příkaz `del`. Takto můžeme ze seznamu vymazat dokonce i celou podsekvenci:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

Příkazem `del` můžeme také odstranit celé proměnné. Po odstranění této proměnné již nebude možné se na ní odkazovat.

```
>>> del a
>>> print a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
```

5.3 Tuple a sekvence

V předchozí části jsme poznali, že seznamy a řetězce toho mají mnoho společného, lze je indexovat, používat jejich podsekvence apod. Kromě těchto dvou datových typů Python podporuje ještě další datový typ — *tuple*. Tuple si lze představit jako *uspořádanou n-tici*, jejíž prvky nelze měnit.

Tuple se zapisuje jako n-prvků oddělených čárkou:

```
>>> t = 12345, 54321, 'ahoj!'
>>> t[0]
12345
>>> t
(12345, 54321, 'ahoj!')
>>> # Tuple mohou být skládány:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'ahoj!'), (1, 2, 3, 4, 5))
```

Závorky kolem výčtu prvků nejsou povinné, v některých případech se jim však nevyhneme (např. předávání argumentů funkcím). Tuple lze vkládat do sebe, lze používat klasické operace jako indexování prvků a podsekvencí.

Tuple mají mnoho použití (např. páry souřadnic (x, y) atd.). Důležité pro nás je, že jde o neměnný datový typ, tudíž je lze za určitých podmínek použít jako klíče slovníků[5]. Zároveň jejich neměnnost vyžaduje menší režii a tedy i práce s nimi je rychlejší než práce se seznamy.

Existují dva speciální případy tuple — prázdná tuple (zapisuje se párem prázdných závorek) a tuple o jednom prvku (prvek je následován čárkou, není nutné ho uzavírat do závorky).

Například:

```
>>> prazdna = ()
>>> singleton = 'ahoj',    # <-- nezapomeňte ukončující čárku
>>> len(prazdna)
0
>>> len(singleton)
1
>>> singleton
('ahoj',)
```

Operací nazývanou *skládání tuple* se rozumí přiřazení více prvků jedné proměnné. Tato proměnná pak bude tvořena tuple obsahující tyto prvky.

Opačnou operací je *rozklad sekvencí*. V tomto případě stojí na levé straně proměnné oddělené čárkou a straně pravé pak tuple obsahující prvky, které budou přiřazeny proměnné:

```
>>> x, y, z = t
```

Rozklad sekvencí vyžaduje, aby délka tuple na pravé straně byla stejná jako délka výčtu proměnných na straně levé. Nakonec si všimněte malé asymetrie, zatímco skládání vždy vytvoří tuple, rozklad funguje na libovolnou sekvenci (tj. jak na tuple, tak i na řetězce a seznamy)!

[5] O slovnících se dovíme více dále v této kapitole

5.4 Slovníky

Dalším neméně důležitým datovým typem Pythonu je *slovník*, někdy nazývaný též *asociativní pole*. Jde o neuspořádanou množinu dvojic *klíč*: *hodnota*. Hodnoty nejsou reprezentovány indexy, ale *klíči*. Z toho vyplývá, že klíče musí být v rámci jednoho slovníku jedinečné.

Na klíč jsou kladeny určité požadavky. Jde především o *neměnnost*, klíčem nemůže být žádný proměnný datový typ. Lze tedy použít čísla, řetězce a dokonce tuple, ty však musí opět obsahovat pouze tyto typy. Seznamy nelze jako klíče nikdy použít[6]. Hodnotami ve slovníku mohou být libovolné typy, dokonce opět slovníky.

Slovník v Pythonu zkonstruujeme zápisem dvojic *klíč*: *hodnota* mezi složené závorky. Pro vytvoření prázdného slovníku vynecháme zápis hodnot:

```
>>> slovnik1 = {1: 'jedna', 2: 'dva', 3: 'tri'}
>>> slovnik2 = {}
```

Slovník je *proměnný* datový typ, je možné do něj ukládat hodnoty pod zadaným klíčem a číst z něj hodnoty určitého klíče. Příkazem `del` je dokonce možné smazat pár *klíč*: *hodnota*. Pokud do slovníku uložíme novou hodnotu pod klíčem, který již slovník obsahuje, dojde k přepsání hodnoty tohoto klíče. Pokud ze slovníku chceme získat hodnotu neexistujícího klíče, dojde k výjimce.

Slovníky mají i několik metod, které jsou vesměs určeny pro manipulaci s klíči apod. My si uvedeme pouze dvě — metoda `keys()`, vrátí seznam všech existujících klíčů použitých ve slovníků (jejich pořadí je vesměs náhodné). Druhá metoda `has_key()` vrátí logickou jedničku, pokud je klíč, který jí byl předán jako argument, obsažen ve slovníku:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

5.5 Porovnávání sekvencí a dalších typů

Sekvenční objekty mohou být porovnávány s dalšími sekvencemi. Toto porovnání se děje *lexikograficky*, nejprve se porovnají první dva prvky, dále další dva atd. Pokud se na nějakém indexu liší, pak tento rozdíl dá výsledek celému porovnání[7].

Jestliže některé prvky jsou opět sekvenční typy, dojde k dalšímu lexikografickému porovnání. Jestliže je jedna sekvence delší než druhá a ve stejně dlouhé části se shodují, je ta kratší vyhodnocena jako menší. Lexikografické porovnávání řetězců používá ASCII (případně Unicode) kódování.

[6] Trváme-li na tom, musíme nejprve seznam převést na tuple funkcí `tuple()` a teprve potom ho použít jako klíč.
[7] Každý pochopí, že jde o známé porovnávání např. řetězců podle velikosti

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Porovnávání objektů různých datových typů Python také umožňuje. Pravidla pro toto porovnání se mohou v budoucnu měnit, nicméně je vždy zaručena jednoznačnost porovnání. Různé typy čísel jsou porovnávány s ohledem na jejich číselnou hodnotu.

KAPITOLA

ŠESTÁ

Moduly

Jak jste si jistě všimli, po ukončení prostředí jazyka Python přijdete o všechny proměnné a funkce, které jste si definovali. Proto, píšete-li delší programy, je vhodnější zdrojový kód uložit do souboru a interpretr nasměrovat na tento soubor. Ten se pak v terminologii jazyka Python nazývá *skript*.[1]. Tím, jak se program stává delší a delší, se často vyskytne potřeba ho rozdělit na několik (relativně) nezávislých částí. Tyto části se nazývají *moduly* a lze je použít i pro vytvoření sdílených knihoven kódu, kdy jednu funkci definovanou v globálně přístupném modulu může používat kterýkoli jiný modul, jenž k němu má přístup. Funkce a proměnné mohou být z modulu *importovány* (*zavedeny*) do jiných modulů, která je pak můžou používat obvyklým způsobem.

Modul je soubor obsahující kód jazyka Python (např. definice funkcí a další příkazy). Je zvykem ukládat moduly do souborů s příponou '.py', pak jméno souboru bez přípony znamená jméno modulu.[2]. Jako příklad má poslouží následující kód (uložený například v souboru 'fibo.py'):

```
# Modul obsahující funkce pro výpočet Fibonacciho rozvoje

def fib(n):    # vytiskne Fibonacciho rozvoj do čísla n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # vrátí Fibonacciho rozvoj do čísla n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Nyní přejděte do adresáře, v němž se nachází váš soubor 'fibo.py', spusťte interpretr Pythonu a zaveďte modul následujícím příkazem:

```
>>> import fibo
```

Tento příkaz vytvoří v lokálním oboru jmen nové jméno (proměnnou) s názvem `fibo`. Toto jméno odkazuje na *objekt modulu*, tento objekt již obsahuje funkce a proměnné definované modulem `fibo`. Na tyto objekty se odkazujeme použitím tečkové notace:

[1] Termín skript platí všeobecně, používá se u všech interpretovaných jazyků.

[2] Přípona '.py' je nutná, běhové prostředá jazyka podle něho rozpozná, že soubor je modulem a umožní ho importovat, toto chování lze změnit přepsáním interní funkce `__import__`

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Pokud nějakou proměnnou nebo funkci budeme potřebovat častěji, je možné si vytvořit lokální jméno odkazující na objekt uvnitř modulu. Přístup k lokálním proměnným je rychlejší a hlavně ušetříte svojí klávesnici, protože nebudete muset neustále opisovat jméno modulu:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Používáme moduly

Moduly mohou obsahovat libovolný kód, jejich použití se tedy neomezuje na pouhé definování řady funkcí. Před prvním zavedením modulu interpretr nalezne soubor obsahující modul, načte ho a spustí všechny příkazy, které obsahuje. Tento proces se nazývá *inicializace* modulu. Při inicializaci vznikají objekty reprezentující funkce uvnitř modulu a je možné při ní nastavit i globální proměnné modulu do určitého stavu. Pokud chce program importovat modul, který je již zinicializován, nedojde k opětovnému spuštění kódu, nýbrž se použije objekt modulu vytvořený při inicializaci.

Zavedením modulu vznikne nový prostor jmen určený pro globální proměnné tohoto modulu. Díky tomuto mechanismu může autor modulu používat libovolné názvy proměnných a přesto nedojde ke kolizi s proměnnými jiného modulu. Proměnné v tomto jmenném prostoru jsou přístupná za použití objektu modulu (klasicky jako `jméno_modulu.proměnná`). Nedoporučuje se ovšem libovolným způsobem modifikovat soukromé proměnné modulu, mohli byste totiž narušit konzistenci dat modulu a ten by mohl začít chybně pracovat.

Je samozřejmé, že moduly mohou importovat jiné moduly. V Pythonu je zvykem umístit veškeré příkazy `import` na začátek modulu či skriptu. Definice jazyka to ovšem nevyžaduje, uznáte-li za vhodné, můžete příkaz `import` použít například ve větvi příkazu `if` nebo uvnitř funkce.

Jak jsme si řekli výše, příkaz `import` zavede do lokálního prostoru jmen objekt modulu. Existuje ale i varianta příkazu `import`, která rovnou zavede některé (popř. všechny) objekty z určitého modulu. Nemusíme tedy používat přiřazení pro vytvoření lokálního jména:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Toto použití příkazu `import` *nevytvoří* jméno, odkazující na objekt modulu, v příkladě výše je tedy proměnná `fibo` (reprezentující modul `fibo`) nedefinována.

Pro zavedení všech jmen z modulu je zde další varianta příkazu `import`:[3]

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

[3] Je známo, že `import` je příkaz s největším množstvím různých variací. Zároveň jde o velmi používaný příkaz, proto byste měli dokonale vědet co a jak provádí. Vše o syntaxi tohoto příkazu se dozvíte z dokumentu *Python Language Reference*.

Tento příkaz zavede z modulu všechna jména s výjimkou těch, která začínají znakem podtržítko. Je třeba podotknout, že tento způsob použití modulů je velmi zrádný. Jednak není známo, na kterém místě je která proměnná definována a jednak již může dojít ke kolizi jmen a proto u některých modulů je výslovně zakázáno použití `from ... import *`. Snažte se proto používání tohoto příkazu omezit pouze na interaktivní sezení a ve skriptech ho raději vůbec nepoužívejte.

6.1.1 Vyhledávání modulů

Co se stane, pokud importujeme modul `fibo` poprvé? Kde všude interpretr hledá soubory obsahující definici tohoto modulu? Především nejprve se interpretr podívá do pracovního adresáře, jestliže zde najde soubor 'fibo.py', veškerá jeho práce končí a pokusí se zavést tento modul.

Pokud však tento soubor v tomto adresáři nenalezne, začne ho vyhledávat podle přesných pravidel specifikovaných definicí jazyka. Tedy: nejprve se podívá na proměnnou prostředí PYTHONPATH, která by měla mít stejnou syntaxi jako proměnná prostředí PATH. Není-li proměnná PYTHONPATH nastavena nebo v adresářích, které specifikuje, není požadovaný soubor nalezen, pokračuje vyhledávání v implicitní cestě. Tu specifikuje proměnná `sys.path`, jde o seznam řetězců reprezentujících adresáře. Obsah této proměnná je závislý na instalaci interpretru a jeho nastavení v modulu `site`. Může vypadat třeba takto: `['/usr/lib/python2.2', '/usr/lib/python2.2/plat-linux2']`.

Je třeba důsledně dbát na to, aby nějaký soubor v pracovním adresáři neměl stejné jména jako nějaký standardní modul. Uzavřeli bychom si tak cestu k tomuto standardnímu modulu, protože při pokusu o zavedení tohoto modulu bychom ve skutečnosti obdrželi modul ze souboru v pracovním adresáři. Pro více informací o standardních modulech nahlédněte do sekce 6.2.

Pokud interpretr požadovaný modul nenalezne, dojde k výjimce, která se rozšíří z příkazu `import`[4]. Obdobně je tomu i v případě, kdy se požadovaný modul podařilo nalézt, ale při jeho inicializaci došlo k výjimce. Zde je třeba dávat pozor na to, že přestože modul nebyl korektně zinicializovaný, při dalším pokusu o jeho zavedení se již druhá inicializace konat nebude a modul bude možné používat nezinicializovaný.

6.1.2 "Kompilované" moduly

Pro zrychlení *spouštění* (nikoli běhu!) krátkých programů používajících velké množství standardních modulů používá Python soubory s příponou '.pyc', tzv. kompilované moduly. Pokud se kód pokusí zavést modul `fibo` a interpretr najde vedle souboru 'fibo.py' i soubor 'fibo.pyc', považuje tento soubor za zkompilovaný modul. Proto porovná čas modifikace souboru 'fibo.py' s časem zaznamenaným v souboru 'fibo.pyc'. Jestliže jsou tyto časy shodné, rozhodne, že soubor 'fibo.pyc' byl vytvořen ze souboru 'fibo.py' a použije ho místo něj. Tím dojde k významnému urychlení zavedení modulu, není třeba jeho kód znovu překládat do bytekódu, protože se použije již přeložená verze.

O vytvoření souboru 'fibo.pyc' se programátor nemusí vůbec starat, interpretr ho vytváří zcela sám kdykoli, když se mu podaří kód souboru 'fibo.py' úspěšně přeložit do bytového kódu. Jestliže se soubor nepodaří vytvořit (důvodem může být plný disk nebo nedostatečná práva uživatele), nedojde k chybě, protože při příštím importu modulu bude byte kód vytvořen znovu. Stejně tak tomu bude i v případě, kdy se sice soubor 'fibo.pyc' podaří vytvořit, ale nebude možné ho zapsat celý, případně jeho obsah bude chybný, interpretr na základě kontrolních součtů tento soubor vyhodnotí jako nevyhovující a po importu souboru 'fibo.py' ho vytvoří znovu.

Následuje několik *důležitých* poznámek týkajících se přeložených modulů:

- Pro administrátory UNIXových systémů je důležité vědět, že obsah souboru 'fibo.pyc' je nezávislý na platformě, je tudíž možná ho sdílet mezi počítači v síti (i heterogenní, obsah nezávisí ani na použitém OS).
- Pokud interpretr jazyka Python spustíte s volbou **-O**, bude generovat optimalizovaný bytový kód, který bude ukládat do souborů '.pyo'. Je třeba podotknout, že optimalizace není (zatím) příliš dokonalá, prostředí odstraní

[4] Více o výjimkách v kapitole Výjimky

pouze příkazy `assert` a instrukce `SET_LINENO`. Při používání optimalizace jsou ignorovány všechny soubory `.pyc`.

- Při použití volby **-OO** dojde ke generování byte kódu, který může ve výjimečných případech ústit v chybnou činnost programu. Nynější verze interpretru odstraňují dokumentační řetězce, díky čemuž vzniknou kompaktnější '.pyo' soubory, naproti tomu ty programy, které závisí na korektnosti dokumentačních řetězců, začnou pracovat chybně.
- Zopakujme, že programy, jejichž moduly jsou čtené ze zkompilovaných souborů, nepoběží rychleji, dojde pouze k rapidnímu zkrácení doby potřebné pro zavedení modulu
- Pokud předáváte jméno skriptu, který si přejete spustit, přímo interpretru, nebude se pro něj generovat kompilovaný soubor '.pyc' (popř. '.pyo'). Rychlost startu skriptů proto může být zvýšena přesunutím většiny kódu do speciální modulu. Interpretru potom předáme pouze malý skript, který spustí hlavní výkonný kód z tohoto modulu.
- Interpretr dokáže používat soubory '.pyc' ('.pyo'), aniž by existoval jejich originál v podobě souboru '.py'. Interpretru je dokonce možné předat jméno '.pyc' souboru a ten ho bez problémů spustí. Obsah souboru '.pyc' je vytvořen tak, že není možné získat původní podobu souboru '.py'. Takto je možné bezpečně vytvářet proprietární kód.
- Python nabízí utilitu ve formě modulu `compileall`, který dokáže vytvořit '.pyc' nebo '.pyo' soubory pro všechny moduly v určitém adresáři.

6.2 Standardní moduly

Základní distribuce Pythonu obsahuje velké množství modulů (jsou popsány v samostatném dokumentu *Python Library Reference*). Díky nim je možné realizovat množství úloh spojených s internetem, unixovými databáze dbm, prací se soubory, operačním systémem a mnoho a mnoho dalších.

Některé moduly jsou přímo součástí interpretru, umožňují totiž přístup k operacím, které nejsou přímo součástí jádra prostředí, ale jsou mu velice blízká (např. v dané aplikaci závisí na rychlosti vykonávání nebo je třeba volat funkce operačního systému či dalších knihoven). Zahrnutí těchto modulů je závislé na konfiguraci a překladu interpretru. (Například modul `Tkinter` určený jako rozhraní ke grafické knihovně Tk je přítomný pouze na systémech s knihovnou Tk).

Mezi všemi moduly najdeme jeden, který nabízí samotné jádro běhového systému, modul `sys` a obsahuje ho každý interpretr. Jeho proměnné `sys.ps1` a `sys.ps2` umožňují v interaktivním módu nastavení primární a sekundární výzvy interpretru:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Ahoj!'
Ahoj!
C>
```

Jak již jistě víte, proměnná `sys.path` je seznam řetězců, který určuje cesty v nichž interpretr vyhledává moduly. Je inicializována z proměnné prostředí PYTHONPATH příp. je nastavena v modulu `site`. Jde o klasický seznam, který můžete měnit běžnými operacemi definovanými nad seznamy:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 Funkce `dir()`

Interní funkci `dir()` použijete v případě, že chcete zjistit všechna jména, která jsou definována uvnitř nějakého objektu. Tato funkce vrací seznam řetězců setříděný podle abecedy, jeho používání reprezentuje následující příklad:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'argv', 'builtin_module_names',
 'byteorder', 'copyright', 'displayhook', 'exc_info', 'exc_type',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'getdefaultencoding',
 'getdlopenflags', 'getrecursionlimit', 'getrefcount', 'hexversion',
 'maxint', 'maxunicode', 'modules', 'path', 'platform', 'prefix', 'ps1',
 'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
 'version_info', 'warnoptions']
```

Bez argumentů vrátí seznam jmen, které jsou definovány v aktuálním prostoru jmen:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Seznam vrácený funkcí `dir()` zahrnuje všechna jména definovaná v objektu, nejde tudíž pouze o proměnné, ale i o funkce, moduly, třídy apod. Do tohoto seznamu nejsou zahrnuta jména interních funkcí a proměnných (tj. těch definovaných modulem `__builtin__`). Pokud potřebujete seznam interních funkcí použijte

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'FloatingPointError', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TypeError', 'UnboundLocalError',
 'UnicodeError', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', 'abs', 'apply', 'buffer', 'callable', 'chr', 'classmethod',
 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr',
 'dict', 'dir', 'divmod', 'eval', 'execfile', 'exit', 'file', 'filter',
 'float', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len',
 'license', 'list', 'locals', 'long', 'map', 'max', 'min', 'object',
 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range', 'raw_input',
 'reduce', 'reload', 'repr', 'round', 'setattr', 'slice', 'staticmethod',
 'str', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange',
 'zip']
```

6.4 Balíčky

Balíčky jsou logickým rozšířením mechanismu Pythonových modulů. V Pythonu je totiž široce používána "tečková notace" a právě balíčky umožňují hiearchickou organizaci modulů.

Například modul se jménem `A.B` znamená modul pojmenovaný ‘`B`’ v balíčku ‘`A`’. Balíčky umožňují dekompozici rozsáhlých knihoven do menších souborů - modulů. Autoři jednotlivých modulů se nemusí zajímat o jména globálních proměnných jiných modulů v tomtéž balíčku, jazyk zajišťuje vzájemnou "izolaci" balíčků mezi sebou. Proto se například autoři jednotlivých modulů rozsáhlých balíčků (např. *NumPy* nebo *Python Imaging Library*) nemusí obávat střetu jmen svých globálních proměnných s proměnnými jiného autora.

Představte si modelovou situaci: chcete navrhnout balíček (kolekci modulů) pro manipulaci se zvukovými soubory a zvukovými daty obecně. Poněvadž existuje mnoho různých zvukových formátů, potřebujete vytvořit a spravovat kolekci modulů pro konverzi mezi těmito formáty. Také si můžete představit mnoho operací, které lze se zvukovými daty provádět (např. mixování stop, přidávání ozvěny, aplikování ekvalizéru ...). Postupem času si vytvoříte mnoho modulů pro tyto činnosti. Pro jejich organizaci je mechanismus balíčků naprosto ideální. Zde je možná struktura vašeho balíčku (zobrazená jako hiearchický souborový systém):

```
Sound/                          Hlavní balíček
      __init__.py               Inicializace balíčku
      Formats/                  Balíček pro konverzi souborových formátů
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  Balíček pro zvukové efekty
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  Balíček filtrů
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Jistě se ptáte, co znamenají soubory '__init__.py'. Python podle nich rozpoznává adresáře obsahující balíčky. To zabraňuje nedorozuměním, kdy Python považoval libovolný adresář nacházející se v jeho cestě za balíček. V nejjednodušším případě je soubor '__init__.py' pouhopouhý prázdný soubor, můžete sem ale umístit inicializační kód balíčku nebo nastavit proměnnou `__all__` (její význam je popisován později).

Uživatelé balíčku z něj mohou přímo importovat jednotlivé moduly:

```
import Sound.Effects.echo
```

Tento příkaz načte modul `Sound.Effects.echo`. Jeho proměnné a funkce však musí být odkazovány plným jménem:

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Jinou možností je přímé importování modulu z balíčku:

```
from Sound.Effects import echo
```

Tento příkaz také načte modul `echo` a učiní jej přístupným za použití jeho jména. Funkce v něm definované můžete používat třeba následovně:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Další cestou je importování určité funkce nebo proměnné přímo:

```
from Sound.Effects.echo import echofilter
```

Tento příkaz opět načte modul `echo`, ale učiní jeho funkci `echofilter()` přístupnou přímo:

```
echofilter(input, output, delay=0.7, atten=4)
```

Na závěr podotkněme, že při importování nějakých jmen (proměnných, modulů apod.) za použití příkazu `import` můžeme na místě objektu, z něhož chceme import provést uvést pouze balíček nebo modul. Příkaz '`from objekt_různý_od_modulu_či_balíčku import jeho_atribut`' je považován za běhovou chybu a z místa jeho použití se rozšíří výjimka. Jako náhradu takovéto konstrukce lze použít '`jeho_atribut = objekt_různý_od_modulu_či_balíčku.jeho_atribut`'.

6.4.1 Importování * z balíčku

Co se nyní stane, napíše-li uživatel `from Sound.Effects import *`? Nejideálnější, ale nereálná cesta by mohla vypadat takto: interpretr projde souborový systém, najde moduly patřící tomuto balíčku a zavede je. Bohužel. Na platformách Mac a Windows totiž souborový systém neuchovává informace o velikosti písmen jmen souborů! Neexistuje zde žádná záruka, že soubor 'echo.py' nebude importován jako modul `echo`, `Echo` nebo dokonce `ECHO`. V operačním systému DOS se k těmto problémům ještě přidává omezení délky jmen souborů na 8 znaků + 3 znaky přípony.

Python proto nabízí své platformně nezávislé řešení: autor balíčku prostě interpretru sdělí, jaké moduly balíček obsahuje. Příkaz import byl modifikován a používá následující konvenci: jestliže kód v souboru '__init__.py' (vzpomeňte si, že jde o inicializační soubor celého balíčku) definuje proměnnou pojmenovanou `__all__` typu seznam, pak je tato proměnná považována za výčet všech modulů, které mají být importovány při vykonání příkazu '`from balíček import *`'. Udržování tohoto seznamu v aktuálním stavu je povinností autora balíčku. Ten se může rozhodnout ho dokonce nepodporovat, pokud považuje importování * za bezvýznamné. Například soubor 'Sounds/Effects/__init__.py' by mohl obsahovat následující kód:

```
__all__ = ["echo", "surround", "reverse"]
```

Tím interpretru říkáme, že balíček `Sound.Effects` obsahuje celkem tři moduly: `echo`, `surround` a `reverse`.

Jestliže proměnná `__all__` není definována, příkaz `from Sound.Effects import *` pouze spustí inicializační kód balíčku (soubor '__init__.py') a posléze zavede všechna jména definovaná tímto kódem. Tento kód může explicitně zavést nějaký z modulů balíčku do hlavního prostoru jmen balíčku. Import všech jmen pak zavede i tento modul.

Existuje ještě jedna nuance při importování modulů z balíčků. Pokud před provedením `from balíček import *` zavedete explicitně nějaký modul tohoto balíčku, bude při importování všech jmen z modulu mezi nimi i tento modul. Vše snad názorně vysvětlí následující kód (předpokládá, že soubor '__init__.py' je prázdný soubor):

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

Po posledním příkazu import budou v lokálním prostoru jmen načteny moduly `echo` a `surround`, protože již byly načteny předchozími příkazy import. (Toto chování je nezávislé na proměnné `__all__` a je vcelku logické, pokud jsme nějaký modul zavedly a interpretr o něm "nevěděl", pro příště si ho zapamatuje a používá ho.)

Stejně jako u modulů, i u balíčků platí: s konstrukcí ‘`from ... import *`’ zacházejte obezřetně, následné starosti s nefunkčním a nepřehledným kódem vás propříště vyléčí. Pokuste se jí používat pouze v *interaktivních* sezeních.

6.4.2 Odkazování se na moduly uvnitř balíčku

Mnohdy se potřebuje jeden modul odkazovat na jiný modul téhož balíčku. Vyjděme z našeho modelového příkladu. Modul `surround` bude potřebovat nějakou funkci z modulu `echo`. V tomto případě platí následující: příkaz `import` se nejprve podívá do aktuálního balíčku a až poté prohledá standardní cestu (určenou proměnnou `sys.path`). Proto může modul `surround` jednoduše použít ‘`import echo`’. Pokud by modul `echo` nebyl součástí aktuálního balíčku, přišly by na řadu standardní pravidla pro vyhledávání modulu.

Je-li balíček strukturovaný na podřízené balíčky (v modelovém příkladě například balíček `Sound`), neexistuje žádný způsob, jak se odkazovat na modul rodičovského balíčku, musíme použít jeho plné jméno. Pokud třeba modul `Sound.Filters.vocoder` potřebuje modul `echo` balíčku `Sound.Effects`, musí použít příkaz `from Sound.Effects import echo`.

KAPITOLA

SEDMÁ

Vstup a výstup

Každý program je vlastně předpis, transformační funkce, která určitým způsobem převádí vstupní data na výstupní. Celá toto kapitola bude věnována způsobům, kterými může program vstupní data získat a naopak jak může uživateli vrátit svoje výstupní data.

Program může výstupní data vrátit dvěma způsoby, buď to je naformátuje do určitého tvaru tak, aby je uživatel mohl bez problémů přečíst a zapíše je na standardní výstup, nebo výstupní data určitým způsobem zakóduje a uloží do souboru.[1]

7.1 Formátování výstupu

Čtenář, který se dostal až sem, jistě poznal dva způsoby, kterými lze na obrazovku vytisknout nějakou hodnotu. První, která přichází v úvahu pouze v případě interaktivního interpretru, je vyhodnocení výrazu, kdy interpretr vytiskne hodnotu tohoto výrazu na obrazovku. Druhou možnost, použitelnou všude, nabízí příkaz `print`. Ten vypíše argumenty jemu předané na standardní výstup.[2]

S největší pravděpodobností budete potřebovat větší kontrolu nad formátováním výstupu vašho programu. Samotný příkaz `print` vám nabízí pouhé vytisknutí hodnot, je-li jich více, oddělí je mezerou. Jak jistě cítíte, nebude to to pravé ořechové. Nyní máte dvě cesty, kterými se můžete dát, první - složitější - spočívá v napsání veškeré formátovací logiky, druhá - jednodušší - používá standardní modul `string`.

Python umožňuje převést libovolnou hodnotu na řetězec. K tomu slouží dvojice funkcí `str()` a `repr()`. Ekvivalentem poslední jmenované funkce je zapsání výrazu mezi obrácené apostrofy ‘‘.

Funkce `str()` by měla vracet takovou reprezentaci argumentu, která je srozumitelná především pro člověka. Proto pro řetězce vrátí jejich čistou hodnotu, pro reálná čísla vrátí řetězec, obsahující desetinný rozvoj na 12 platných číslic.

Naproti tomu funkce `repr()` (a její ekvivalent v podobě obrácených apostrofů) vrací reprezentaci argumentu, která je vhodná pro interpretr. Čili vrací hodnotu tak, jak bychom jí zapsali do zdrojového souboru. Např. k řetězcům přidává úvodní a ukončovací uvozovky, reálná čísla vrací s přesností na 17 platných míst. Pro ty objekty, pro něž neexistuje čitelná reprezentace, vrací funkce `repr()` stejnou hodnotu jako `str()`, to platí především pro strukturované datové typy (seznamy, slovníky):

[1] Poznamenejme, že na UNIXových systémech se stírá rozdíl mezi oběma přístupy, lze zde totiž přesměrovat standardní výstup do souboru. To je ovšem dáno filozofií tohoto výborného systému.

[2] Další možností je používat objekty reprezentující standardní výstup (resp. standardní chybový výstup) `sys.stdout` (`sys.stderr`). Pomocí jejich metody `write()` do nich můžete zapsat libovolný řetězec. Více informací o těchto standardních souborových objektech získáte z dokumentu Python Library Reference.

```
>>> s = 'Ahoj světe.'
>>> str(s)
'Ahoj světe.'
>>> `s`
"'Ahoj světe.'"
>>> str(0.1)
'0.1'
>>> `0.1`
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Hodnota x je ' + `x` + ' a y ' + `y` + '...'
>>> print s
Hodnota x je 32.5 a y 40000...
>>> # Obrácené uvozovky je možné použít i s jinými typy:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[32.5, 40000]'
>>> # Konvertování řetězců přidá uvozovky a zpětná lomítka:
... ahoj = 'ahoj světe\n'
>>> ahojs = `ahoj`
>>> print ahojs
'ahoj světe\n'
>>> # Zpětné uvozovky můžeme použít i na tuple:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

Vraťme se však k problematice formátování výstupu programu. Nejprve si uvedeme krátký příklad - program vypisující tabulku druhých a třetích mocnin:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(`x`, 2), string.rjust(`x*x`, 3),
...     # Nezapomeňte ukončující čárku na předchozím rádku
...     print string.rjust(`x*x*x`, 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Tento příklad je ukázkou použití formátovací funkce `string.rjust()`, která zarovná řetězec mezerami vpravo, přičemž jako šířku sloupce použije druhý argument. Stejně tak existují i funkce `string.ljust()` a `string.center()`.

Jde o standardní funkce, které pouze vrátí upravený řetězec. Pro vypsání jejich hodnoty musíme použít příkaz `print`. V předchozím příkladě si všimněte, jak příkaz `print` vložil mezi svoje argumenty mezery. (Pro zkrácení řádku jsme ho rozložili na dva, první příkaz `print` je ukončen čárku a nevypisuje tudíž znak konce řádku).

U všech funkcí, které ovlivňují zarovnání řetězce se setkáme s problémem, kdy obsah pole má větší šířku než požaduje

programátor. Zde si zapamatujte, že výše uvedené funkce v tomto případě zachovávají původní řetězec, což sice pokazí vaše formátování, nicméně data zůstanou korektní. Pokud opravdu víte, že potřebujete výsledek pevně oříznout, použijte cosi jako ‘`string.ljust(x, n)[0:n]`’.

Pro úplnost si uvedeme ještě čtvrtou funkci `string.zfill()`. Ta doplní řetězec, který reprezentuje nějaké číslo, zleva nulami na požadovanou šířku sloupce, přičemž správně interpretuje znaménko plus a mínus před číslem:

```
>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

Pro účely formátování výstupu je možné použít i operátor %. Jak je vám zcela jistě známo, jde o operátor modulo (zbytek po dělení), nicméně, pokud na místě prvního operandu uvedete řetězec, změní se zcela jeho funkce. Začne totiž pracovat obdobně jako C funkce `sprintf()`, první řetězec je použit jako tzv. formátovací řetězec, přičemž výskyty speciálních znaků jako `%s`, `%r` a dalších jsou nahrazeny určitou hodnotou.

Nyní se výše uvedený výpis kódu pro vytištění tabulky druhých a třetích mocnin přepíšeme za použití operátoru %:

```
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Operátor % prochází řetězec - první argument a hledá v něm předlohy ve tvaru `%<formát>`. Posloupnost znaků `<formát>` specifikuje, jak bude interpretována hodnota předaná jako druhý operand. Může se určit jak konverzní funkce (‘`f`’ - převod z čísla v plovoucí řádové čárce na řetězec, ‘`d`’ - převod číslo na řetězec), tak i formát výsledku, lze specifikovat celkovou šířku pole a počet číslic za desetinnou čárkou. Výsledek konverzní funkce bude "dosazen" do původního řetězce:[3]

```
>>> import math
>>> print 'Hodnota Ludolfova čísla je přibližně %5.3f.' % math.pi
Hodnota Ludolfova čísla je přibližně 3.142.
```

Budete-li chtít do jednoho řetězce dosadit více hodnot, musí být pravý operand tuple. To nám názorně ukazuje další příklad:

[3] Poznamenejme, že úplnou tabulku kontrolních znaků a popis operátoru % najdete ve specifikaci jazyka Python.

```
>>> tabulka = {'Adolf': 4127, 'Běta': 4098, 'Cyril': 7678}
>>> for jmeno, telefon in tabulka.items():
...     print '%-10s ==> %10d' % (jmeno, telefon)
...
Běta       ==>       4098
Adolf      ==>       4127
Cyril      ==>       7678
```

Pro programátory v C bude dobrou zprávou, že formátovací sekvence operátoru % pracují většinou stejně jako v C. Pokud C funkci `sprintf()` předáte chybní argument, program většinou skončí s neoprávněným přístupem do paměti, zatímco v Pythonu se z tohoto místa "pouze" rozšíří výjimka.

I význam příkazu `%s` je mnohem volnější, originál v jazyce C připouštěl na místě argumentu pouze řetězec, zatímco v Pythonu můžete použít libovolný typ. Jeho řetězcovou reprezentaci operátor získá pomocí nám již důvěrně známé funkce `str()`.[4] Podobný pár tvoří příkaz `%r` a funkce `repr()`.

Další variantou, kterou Python usnadňuje programátorovi život, je možnost se ve formátovacím řetězci (tj. tom, který stojí vlevo od operátoru %) odkazovat na položku nějakého slovníku, ten je operátoru předán jako druhý argument. Formátovací příkaz pak bude mít tvar `%(jméno_klíče)<formát>`:

```
>>> tabulka = {'Adolf': 4127, 'Běta': 4098, 'Cyril': 8637678}
>>> print 'Běta: %(Běta)d; Adolf: %(Adolf)d; Cyril: %(Cyril)d' % tabulka
Břét'a: 4098; Adolf: 4127; Cyril: 8637678
```

Takto můžeme velice "vkusně" vytisknout hodnotu nějaké proměnné. Využijeme přitom funkce `vars()`, která vrací slovník, obsahující všechny proměnné (klíče jsou jména proměnných, hodnoty pak skutečné hodnoty proměnných):

```
>>> ahoj = 'Vítáme Vás'
>>> print 'Máme pro vás následující zprávu: %(ahoj)s!' % vars()
Máme pro vás následující zprávu: Vítáme Vás!
```

7.2 Práce se soubory

Velice často programátor potřebuje přečíst/zapsat nějaká data z/do souboru. Pomineme-li možnost UNIXového shellu, jenž dokáže přesměrovat standardní vstup a výstup programu, musí se o to programátor postarat sám.

Ve většině programovacích jazycích je soubor reprezentován jako *souborový objekt*, který obsahuje data o samotném souboru. Před jeho používáním programem ho musíme *otevřít*, čímž vytvoříme v paměti objekt reprezentující tento soubor. Při otevření specifikujeme i další potřebné informace - *jméno souboru*, *mód přístupu* (*pouze ke čtení*, *pouze pro zápis*, *přidávání*, *čtení i zápis*), *velikost vyrovnávací paměti* použité k přístupu k tomuto souboru atd.

Nový souborový objekt vytvoříme pomocí funkce `open()`[5]. Tato funkce má jeden povinný argument - jméno souboru a další dva nepovinné, první z nich specifikuje mód přístupu (implicitně *pouze pro čtení*) a druhý velikost vyrovnávací paměti (tento údaj je důležitý pro knihovnu jazyka C, jež je použita pro implementaci práce se soubory):

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

[4] Oproti C však nejsou podporovány některé příkazy, jmenovitě `%n` a `%p`.
[5] V novějších verzích jazyka je preferována funkce `file()` se stejným rozhraním.

Zůstaňme na chvíli u řetězce, reprezentujícího mód přístupu k souboru. První možností je řetězec `'r'` určující přístup *pouze pro čtení*, v našem příkladě jsme použili mód `'w'`, který specifikuje přístup *pouze pro zápis*. Další možností je `'a'`, který otevře soubor pro přidávání, což je totéž jako zápis, jen data se budou přidávat na konec souboru. Dejte si pozor při používání módu `'w'`, pokud soubor existuje, budou data bez jakéhokoli varování smazána. Poslední mód `'r+'` umožňuje *čtení i zápis*.

Na systémech Windows a Macintosh se setkáte ještě s modem `'b'`, který způsobí otevření souboru v binárním módu. Tyto systémy totiž rozlišují mezi textovými a binárními soubory. Z toho vyplývá, že exitují i módy jako `'rb'`, `'wb'` a `'r+b'`. Opomenutí znaku `'b'` může způsobit nemalé problémy, proto se ho snažte používat i na platformách, které sice tento mód nepoužívají. Brzy byste totiž mohli chtít váš program portovat na jinou platformu a mohli byste se dočkat nemilých překvapení.

7.2.1 Čtení a zápis souborů

Předpokládejme, že existuje souborový objekt, pojmenovaný `f`. Tento objekt nám poskytuje několik metod (funkcí, které se vážou na tento objekt), pomocí nichž můžeme načítat obsah tohoto souboru.

První operací, kterou souborové objekty podporují je čtení jejich obsahu, k tomu slouží metoda `f.read()`. Jako argument jí můžeme předat i počet bytů, které se mají ze souboru přečíst. Výsledné byty vrátí jako řetězec. Pokud bylo při tomto volání `f.read()` dosaženo konce souboru, vrátí se pouze ty znaky, které v souboru zbývaly. Při opakovaném čtení za koncem souboru vrací tato metoda prázdný řetězec. Argument, specifikující počet bytů je volitelný, v případě, že se rozhodnete ho nespecifikovat, přečte se celý obsah souboru.

```
>>> f.read()
'Toto je obsah celého souboru.\n'
>>> f.read()
''
```

Metoda `f.readline()` přečte ze souboru jeden řádek a vrátí ho jako řetězec. Znak konce řádku `'\n'` je obsažen na konci řetězce. V případě čtení za koncem souboru vrací prázdný řetězec. To umožňuje programu rozlišit mezi prázdným řádkem a koncem souboru, prázdný řádek pak obsahuje jediný znak konce řádku.

```
>>> f.readline()
'Toto je první řádka souboru.\n'
>>> f.readline()
'Druhá řádka souboru\n'
>>> f.readline()
''
```

Užitečná metoda `f.readlines()` přečte několik řádků ze souboru a vrátí je jako seznam. Pokud jí program nepředá žádný argument, přečte všechny řádky, jinak pouze tolik bytů, kolik určuje argument a navíc ještě ty, které jsou třeba k dokončení aktuální řádky. Pro znak konce řádku platí stejná pravidla jako u metody `f.readline()`.

```
>>> f.readlines()
['Toto je první řádka souboru.\n', 'Druhá řádka souboru\n']
```

Z pomoci metody `f.write()` můžeme do souboru zapisovat. Předáme jí vždy řetězec, který se zapíše do souboru přesně tak, jak je. Tato metoda vždy vrací hodnotu `None`.

```
>>> f.write('Zkouška mikrofonu - 1 ... 2 ... 3\n')
```

V souboru je možné se i pohybovat a zjišťovat aktuální pozici (tj. pozici, ze které bude probíhat další čtení a kam se

budou zapisovat další data). Pomocí metody `f.tell()` můžeme zjisti aktuální pozici (*offset*) ukazatele v souboru, měřenou v bytech od začátku souboru. Pro posunutí tohoto ukazatele použijete metodu `f.seek()`, jejíž první argument bude specifikovat *offset* a druhý *pozici* vzhledem ke které se offset vztahuje. Jestliže pozici nespecifikujete (nebo použijete hodnotu 0), bude offset reprezentovat novou vzdálenost ukazatele od začátku souboru, hodnota 1 bude znamenat posun relativně k aktuální pozici (záporná hodnota posouvá zpět), konečně hodnota 2 posun o offset bytů od konce souboru.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Jdi na šestý byte v souboru
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Jdi na 3 byte před koncem souboru
>>> f.read(1)
'd'
```

Pokud jste skončili veškerou práci se souborem, je slušné ho *uzavřít*, čímž uvolníte systémové prostředky vyhrazené k manipulaci s tímto souborem. K uzavření souboru slouží metoda `f.close()`, po uzavření souboru již není možné k souboru jakýmkoli způsobem přistupovat:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Další metody souborových objektů (`f.isatty()` nebo `f.truncate()`) se používají pouze zřídka a proto se jim zde nebudeme věnovat, případný zájemce najde veškeré informace v Python Library Reference.

7.2.2 Modul `pickle`

Jak jsme si ukázali, Python dokáže bez problémů zapsat do souboru řetězce. Jestliže však potřebujete číst nebo zapisovat čísla, neobejdete se bez dalšího kódu, který bude provádět případnou konverzi z čísla na řetězec a naopak. V případě, kdy budete chtít zapisovat komplexnější datové typy jako třeba seznamy, tuple, slovníky nebo dokonce instance uživatelských tříd, budete muset buď napsat hodně kódu nebo použít již připravený modul `pickle`.

Ten umí ukládat do souboru veškeré interní datové typy Pythonu, většinu instancí uživatelských tříd a dokonce i některé formy kódu. Pro implementaci tohoto zápisu používá převod na řetězcovou reprezentaci (tento proces se nazývá *pickling* nebo též *serializace*). Zároveň dokáže z řetězcové reprezentace zrekonstruovat původní objekt (*unpickling*, *deserializace*). Řetězcová reprezentace objektu obsahuje veškeré informace o jeho vnitřním stavu, můžete jí přenést po počítačové síti nebo uložit do souboru a přesto při deserializaci získáte zpět objekt s původním obsahem.

Předpokládejme, že chcete do souboru `f` uložit nějaký objekt `x`. Jednoduše tak učiníte funkcí `pickle.dump()`:

```
import pickle
pickle.dump(x, f)
```

Pro rekonstrukci objektu ze souboru `f` stačí použít funkci `pickle.load()`:

```
x = pickle.load(f)
```

Modul `pickle` podporuje mnohem více funkcí, než jsme si zde v krátkosti načrtli, kompletní přehled opět poskytne

Python Library Reference.

Připomeňme, že modul `pickle` je standardní modul pro implementaci *perzistentních* objektů, tj. objektů, které mohou být uloženy a znovu použity jiným programem, "žijí" nezávisle na programu. A jelikož module `pickle` najdete v každé instalaci Pythonu, existuje silný tlak na tvůrce nových modulů, kteří musí chtě nechtě podporovat rozhraní tohoto modulu. Díky tomu je možné pomocí `pickle` ukládat i další datové typy (např. matice balíku NumPy apod.).

KAPITOLA

OSMÁ

Chyby a výjimky

V předchozím výkladu jsme mlčky přehlíželi jednu důležitou skutečnost - žádný program není bez chyby. Pokud uvažujeme chyby programátora, může jít o *syntaktické chyby* (většinou překlepy, špatně napsaný zdrojový kód) nebo *chyby v logice programu* (špatný návrh programu, chybné používání funkcí atd.). Syntaktické chyby lze poměrně snadno odstranit, chyby v logice však nikoli. Pokud již program pracuje jak má, stále na něj působí jakési vnější vlivy (operační systém, ostatní programy, omezení vyplývající z hardware apod.). Pokud se třeba program snaží zapisovat na disk a přitom je tento disk zaplněn, běhové prostředí Pythonu mu to sdělí pomocí *výjimek*. Právě jim proto bude věnována převážná část této kapitoly.

8.1 Syntaktické chyby

Pokud jste si zkoušeli jednotlivé příklady v této knize, určitě jste na nějaké syntaktické chyby narazili. Syntaktické chyby vznikají při zápisu programu, příkladem budiž následující řádka kódu:

```
>>> while 1 print 'Ahoj, světe'
  File "<stdin>", line 1, in ?
    while 1 print 'Ahoj, světe'
                ^
SyntaxError: invalid syntax
```

Jistě jste si všimli chybějící dvojtečky za příkazem `while`. Pokud se takto zapsaný kód pokusíte předat interpretru, pak ho vyhodnotí[1] jako chybně zapsaný a vyvolá chybu (výjimku).

Důsledkem výjimky je vytisknutí chybové hlášky, kterou vidíte na předchozí ukázce. Všimněte si především malé "šipky", která ukazuje na místo, kde byla chyba detekována (tj. na příkaz `print`). Výskyt chyby lze předpokládat v těsném okolí této šipky (v našem případě *před* příkazem `print`). Součástí chybového výpisu je také jméno souboru a číslo řádky, takže přesně víte, kde chybu hledat.

8.2 Výjimky

Pokud je příkaz nebo výraz syntakticky správně, pokusí se ho interpretr spustit. I v této fázi však může dojít k chybě, proto interpretr podporuje mechanismus *výjimek*. Pomocí nich lze tyto chybové stavy indikovat a zajistit tak nápravu. Každá výjimka nejprve vznikne, je *vyvolána* nějakým příkazem. Po vyvolání výjimky je ukončen aktuální příkaz a výjimka se z tohoto příkazu *šíří* do funke, která tento příkaz spustila, z této funkce do funkce, která jí volala atd. stále výše v zásobníku volaných funkcí. Každá funkce může výjimku *odchytit*, čímž zastaví její šíření a může jí zpracovat.

[1] Syntaxe programu se zpracovává v části nazývané parser, proto se také syntaktickým chybán někdy říká chyby při parsování.

Pokud žádná funkce výjimku neodchytí (tzv. *neodchycená výjimka*), odchytí ji interpretr, ukončí program a vytiskne chybové hlášení:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: spam
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

Jak vidíte v ukázce, chybových hlášení existuje mnoho. Každé indikuje, k jakému druhu chyby došlo, na jakém místě v programu a krátký popis chyby. V příkladu výše najdeme následující výjimky: `ZeroDivisionError` (dělení nulou), `NameError` (neexistující jméno proměnné) a `TypeError` (chybný datový typ). Všem těmto výjimkám odpovídají interní proměnné, narozdíl od jiných jazyků, kde v některých případech jde o klíčová slova! Popis chyby se liší v závislosti na typu výjimky. Většinou blíže specifikuje okolnosti chyby.

Součástí chybového výpisu je i výpis zásobníku volaných funkcí. Tak lze chybu snadno lokalizovat, určit její příčiny a zajistit nápravu. Jako součást výpisu funkcí jsou vypsánu i čísla řádek, jména souborů a samotný řádek, na kterém k chybě došlo. Detailní popis všech interních výjimek hledejte v *Python Library Reference*, části, zabývající se modulem `exceptions`.

8.3 Obsluhování výjimek

Každý blok kódu v programu může odchytit libovolnou výjimku, která vznikne v jeho těle. Ukažme si příklad:

```
>>> while 1:
...     try:
...         x = int(raw_input("Zadejte celé číslo: "))
...         break
...     except ValueError:
...         print "Nebylo zadáno správně, zkuste to znovu..."
...
```

Jistě jste pochopili, že jde o nekonečný cyklus, který se dotazuje uživatele na celé číslo (funkce `raw_input()`), které následně převeden na celé číslo (funkce `int()`). "Vtip" je v tom, pokud funkce `int()` dostane špatný argument (nepůjde o celé číslo, např. '`1.0`' nebo '`nula`' atd.), vyvolá výjimku `ValueError`.

U tohoto programu se setkáme ještě s jednou výjimkou. I když to není na první pohled zřejmé, může v libovolném místě programu vzniknout výjimka `KeyboardInterrupt`. Ta indikuje přerušení od uživatele (tj. stisk kombinace kláves `Control-C`). Tato výjimka odchycena nebude, tudíž cyklus se ukončí.

Možná jste si všimli nového příkazu `try`, který pracuje následovně:

- Nejprve je spuštěno *tělo* příkazu try (tj. kód mezi klíčovými slovy `try` a `except`).
- Pokud tento kód nevyvolá žádnou výjimku, pokračuje běh programu až za příkazem `try`.
- Dojde-li ovšem k *vyvolání* výjimky, je program *přerušen* v místě vzniku výjimky. Následuje porovnání typu výjimky s výčtem výjimek za klíčovým slovem `except` (poznamenejme, že těchto větví může být více, každá

s jiným výčtem výjimek). Je-li odpovídající větev nalezena, výjimka je *odchycena* a je spuštěn kód odpovídající této větvi. Po jeho vykonání pokračuje běh normálně za příkazem `try`. V případě, že není nalezena větev (někdy též *handler*) obsluhující tuto výjimku, výjimka *není odchycena* a šíří se dále v zásobníku volaných funkcí, přičemž může být odchycena nějakým jiným příkazem `try`.

- Pokud výjimka nebude odchycena a rozšíří se až do hlavního programu a odtud ještě výše, odchytí ji interpretr, který vytiskne chybové hlášení (*traceback*) a ukončí program.

Jak jsme si řekli o pár odstavců výše, může mít příkaz `try` více než jednu větev `except`, každou definovanou pro jiné výjimky. V každém případě ale bude spuštěn vždy nejvýše jeden (tj. buď žádný nebo jeden) handler obsluhující výjimku. Pokud u dvou různých větví `except` budou uvedeny stejné typy výjimek, Python to nevyhodnotí jako chybu, ale je třeba dát pozor, že ta větev, která je ve zdrojovém souboru uvedena jako druhá nebude na tuto výjimku reagovat!

Větev `except` může specifikovat více jak jednu výjimku formou tuple[2] několika typů výjimek.

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

U poslední větve můžeme dokonce vynechat výčet výjimek úplně. Pak tento handler funguje jako žolík - zpracuje všechny výjimky, které nebyly odchyceny předchozími větvemi. Ovšem i zde platí známé "všeho s mírou", protože unáhleným použitím `except` bez výpisu výjimek můžeme skrýt nějakou chybu kódu, se kterou jsme předem nepočítali.

Použití samotného `except` se přímo nabízí v případě, kdy chceme výjimku sice odchytit, následně vypsat nějaké chybové hlášení a výjimku znovu pustit do světa (více o vyvolání výjimek se dozvíte níže):

```
import string, sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Nemohu převést data na celé číslo."
except:
    print "Neznámá chyba:", sys.exc_info()[0]
    raise
```

Podobně jako jiné příkazy, i příkaz `try` má volitelnou větev `else`. Ta se zapisuje po všech handlerech a její kód je spuštěn v případě, kdy žádná výjimka nenastala:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'nemohu otevřít', arg
    else:
        print arg, 'obsahuje', len(f.readlines()), 'řádků'
        f.close()
```

[2] Připomeňme, že při zápisu tuple, v případě, kdy nemůže dojít k dvojznačnostem, můžeme vynechat kulaté závorky.

Pokud nějaká výjimka vznikla, pak může mít přiřazenu nějakou hodnotu nazývanou *argument výjimky*. Jeho typ a význam se řídí typem výjimky. Pokud má nějaká výjimka přiřazen argument, můžeme uvést ve větvi `except` za jménem výjimky i jméno proměnná, které se v případě odchycení výjimky tímto handlerem přiřadí argument výjimky. Podle tohoto argumentu můžeme rozhodnout o příčinách chyby a zajistit následnou akci programu na její odstranění:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'jméno', x, 'není definováno'
...
jméno spam není definováno
```

V případě, kdy výjimka má nějaký argument, je tento použit jako popis výjimky (*detail*) v chybovém hlášení interpretru.

Jelikož se výjimka šíří v zásobníku volaných funkcí vždy směrem vzhůru, odchytává příkaz `try` nejen výjimky vzniknuvší v bloku "jeho" kódu, ale i ve všech funkcích, které volá. Například:

```
>>> def chyba():
...     x = 1/0
...
>>> try:
...     chyba()
... except ZeroDivisionError, detail:
...     print 'Běhová chyba:', detail
...
Běhová chyba: integer division or modulo
```

8.4 Vyvolání výjimek

Příkaz `raise` umožňuje programově vyvolat výjimku určitého typu, přičemž lze specifikovat i argument výjimky:

```
>>> raise NameError, 'Ahoj'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Ahoj
```

V našem příkladě bylo jméno výjimky `NameError` a její argument řetězec '`Ahoj`'. Je zvykem všechny výjimky nazývat různými variacemi na téma *Error*, např. `TypeError`, `SyntaxError`, `ImportError` a další. Význam a typ argumentu je, jak jsme si již řekli, závislý na typu výjimky.

V případě, o němž jsme hovořili výše (viz `except` bez výčtu výjimek) použijeme příkaz `raise` bez argumentů. V takovém případě vyvolá poslední výjimku, ke které došlo, tedy:

```
>>> try:
...     raise NameError, 'Ahoj'
... except NameError:
...     print 'Došlo k výjimce!'
...     raise
...
Došlo k výjimce!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: Ahoj
```

8.5 Výjimky definované uživatelem

Program může definovat svoje vlastní výjimky jako třídy odvozené od třídy `Exception` (více o třídách a objektovém programování se dozvíte z následující kapitoly), např.:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return 'self.value'
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'Vyskytla se výjimka s hodnotou:', e.value
...
Vyskytla se výjimka s hodnotou: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Třída výjimky může být definována i jako potomek jiné třídy než `Exception`, v praxi se to ale nepoužívá. Výjimka je plnohodnotnou třídou, může definovat množství atributů a metod. V praxi těchto možností ale příliš nevyužijeme, výjimky by totiž měly být jednoduché třídy sloužící hlavně k informování programu o chybě a její příčině.

Pokud píšete nějaký modul, který používá množství výjimek, je vhodné definovat společného předka těchto výjimek, třeba s názvem `Error` a teprve od něj odvozovat všechny ostatní chyby. Pokud totiž uživatel vašeho modulu bude chtít odchytit všechny výjimky, které mohou ve vašem modulu vzniknout, může napsat jedinou větev `except jméno_modulu.Error`. Například:

```
class Error(Exception):
    """Základní třída všech výjimek v tomto modulu."""
    pass

class InputError(Error):
    """Výjimky vyvolané pro chyby vstupu.

    Atributy:
        expression -- vstupní výraz, ve kterém došlo k chybě
        message -- popis chyby
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Vyvolána, když se program pokusí o nepovolenou změnu stavu modulu.

    Atributy:
        previous -- stav před změnou
        next -- nový stav
        message -- vysvětlení, proč daná změna není povolena
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Mnoho standardních modulů Pythonu definuje vlastní třídy výjmek, jejichž hiearchie je podobná výše uvedené. Pokud jste zde uvedenému výkladu tříd neporozuměli, nelámejte si s tím hlavu. Přečtěte si následující kapitolu věnovanou objektově orientovanému programování a pak se k této části vraťte.

8.6 Definování clean-up akci

Příkaz `try` může mít místo větví `except` jedinou větev `finally`. Ta se používá pro tzv. clean-up akce, tj. kód, který by měl být spuštěn za všech podmínek a který se stará o korektní ukončení těla příkazu `try` (např. zavře otevřené soubory nebo smaže dočasná data). Ukázkovou clean-up akci, reprezentovanou příkazem `print` vidíte v následujícím příkladě:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Mějte se, lidi!'
...
Mějte se, lidi!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

Větev `finally` je spuštěna za všech okolností, pokud v bloku chráněném příkazem `try` dojde k výjimce, je nejprve vykonána větev `finally` a až poté je standardním způsobem vyvolána výjimka. Clean-up akce je však spuštěna i

v případě, kdy k žádné výjimce nedošlo a blok je opuštěn normálním způsobem, případně pomocí příkazu `return` apod.

Častou chybou začátečníků je pokus zkombinovat větve `except` a `finally`. Příkaz `try` vždy musí mít buď pouze větve `except` nebo pouze větve `finally`, v žádném případě ne obě zároveň. Pokud chcete výjimku odchytit a zároveň definovat nějakou clean-up akci, použijte dva do sebe vložené příkazy `try`, přičemž jeden bude definovat větve `except` a druhý větev `finally`.

KAPITOLA

DEVÁTÁ

Třídy

Jeden z hlavních rysů jazyka Python jsme dosud nezmínili. Jedná se o objektově orientované programování (OOP). Tento "moderní" trend[1] se v poslední době čím dál tím více rozmáhá. Některé úlohy si bez něho dokážeme pouze těžko představit (např. komplikované uživatelské rozhraní implementované pomocí strukturovaného programování by byla profesionální sebevražda). Předešleme, že Python je v otázce objektově orientovaného přístupu k návrhu programu velice pokročilý a obzvláště ve verzi 2.2 najdeme mnoho nových vlastností.

Obecně lze říci, že Python díky svému akademickému návrhu (vznikl na univerzitě v Amsterdamu) používá to nejlepší z jiných objektově orientovaných jazyků. Hlavní mechanismy byly inspirování jazykem C++ a Modula-3. Toho všeho bylo dosaženo za použití minima nové syntaxe, což umožňuje snadné pochopení objektového návrhu pomocí jazyka Python.

Python tedy jmenovitě podporuje tyto vlastnosti: *dědičnost* (samozřejmostí je vícenásobná dědičnost), *polymorfismus* (libovolnou metodu může potomek předefinovat, slovy jazyka C++ jsou *všechny* metody virtuální), *zapouzdření* - všechny předchozí vlastnosti musí podporovat každý jazyk, který si chce říkat objektově orientovaný. Další vlastnosti, které buď přímo nebo nepřímo souvisí s objektově orientovaným programováním jsou: *přetěžování operátorů*, *mechanismus metatříd*, *dynamické atributy a mechanismus vlastností* (properties), *kooperativní volání metod*, *možnost odvodit třídu od interního typu* atd. V dalších verzích jazyka lze počítat s dalším vývojem objektového modelu jazyka, lze předeslat, že jazyk bude podporovat *rozhraní* (interfaces) a ještě více se v těchto verzích setře rozdíl mezi interními typy a uživatelskými objekty.

9.1 Použitá terminologiie

Za *objekt* je v jazyce Python libovolná entita, např. řetězec, číslo, slovník, seznam, modul ... to všechno jsou objekty. Každý objekt může mít svoje *atributy*. Jak jsme již poznali, k přístupu k atributům se používá *tečková notace*, čili zápis ve stylu `objekt.jméno_atributu`. A poněvadž jsme před chvílí řekli, že objekt je každá entita, i atribut je opět objekt. Proto můžeme psát i daleko rozsáhlejší konstrukce, např. `window.button1.OnClick.handlers` apod.

Mezi atributy má zvláštní postavení *metoda*. Jde o funkci, která určitým způsobem patří pouze k tomuto objektu. Metody lze používat obvyklým způsobem jako klasické funkce. Jsou však většinou určeny pro modifikování vnitřního stavu objektu. Pro pochopení metod je důležité si uvědomit, že každá metoda modifikuje pouze "svůj" objekt. Podívejme se na následující kód:

[1] Přestože se zdá, že jde o poměrně nový přístup, byl poprvé použit již v sedmdesátých letech v jazyce Simula. Od té doby do nynějška se objevilo množství jazyků pyšnících se nálepkou objektově orientovaný.

```
>>> list1 = [1, 3, 2, 0]
>>> list2 = ['ahoj', 'bbbbbbb', 'AAA', 'root']
>>> list1.sort()
>>> list2.sort()
```

Nadefinovali jsme si dva objekty – seznamy – a následně jsme pro ně zavolali jejich metody `sort()`. Všimněte si, že obě metody, i když se jmenují stejně, jsou různé, protože se každá váže k jinému objektu. Metoda tedy modifikuje pouze ten objekt, k němuž se váže, případně volá jiné metody jiných objektů. Může samozřejmě volat i jiné funkce, fantazii se meze nekladou. Pro důkladný výklad objektově orientovaného programování bych vám doporučil nějakou z odborných knich, které na toto téma vycházejí.

Pokud použijeme terminologii jazyka C++ jako základ terminologie používané v jazyce Python, můžeme říci, že všechny metody objektů jsou *virtuální* (čili libovolná odvozená třída může tuto metodu předefinovat, přičemž tuto novou verzi budou používat všechny ostatní metody tohoto objektu). Zároveň lze říci, že všechny atributy (tj. metody a data třídy) jsou *veřejné*, jinými slovy – má k nim přístup jakýkoli jiný objekt. Tato vlastnost je pro Python klíčová, nic před uživatelem neskrývá a uživatel může, pokud ví co dělá, modifikovat interní data a tím docílit požadovaného (např. nestandardního) chování objektu. Veřejné jsou i pseudo-privátní atributy, které využívají mechanismu náhrady soukromých jmen (více o této problematice si povíme dále).

V Pythonu neexistují žádné konstruktory a destruktory – namísto nich zde existuje dvojice metod `__init__` a `__del__`, které mají obdobnou, ne však stejnou funkci jako konstruktory a destruktory. Jazyk Python je zcela exaktní jazyk, čili máte pouze to, o co jste si řekli, prostředí téměř nikdy nemyslí za vás. Proto není možné z jedné metody volat *přímo* jinou metodu téhož objektu! Každá metoda proto přejímá jeden speciální argument (většinou pojmenovaný *self*), který reprezentuje objekt, jehož metoda je prováděna. Chceme-li tedy z nějaké metody vyvolat metodu téhož objektu se jménem řekněme `foo`, použijeme zápisu: '`self.foo()`'. Takto můžeme přesně specifikovat objekt, jehož metodu chceme volat a nemůže dojít k žádným nejednoznačnostem.

Obdobně jako v jazyce Smalltalk, i v Pythonu je třída také objekt. To ale platí v Pythonu obecně – libovolná datová struktura je objekt, objektem jsou i interní datové typy, moduly, bloky kódu, instance tříd i třídy samotné. Jako v již zmíněném Smalltalku, i Python umožňuje od verze 2.2 dědit třídy od interních datových typů, čímž lze implementovat specifické změny v chování tohoto typu (např. vytvořit datový typ odvození od seznamu, který bude sloužit pouze k uložení řetězců).

Z některých jazyků (např. C++) je známa možnost přetěžování operátorů. Python zavádí řadu "speciálních" metod, které slouží pro implementaci takto přetížených operátorů. Proto není problém nadefinovat třídu `Matice` a implementovat metodu `__add__`, která je volána při použití operátoru '`+`' a která tyto dvě matice sečte. Přetěžování operátorů tedy velice zprůhledňuje výsledný kód programu.

9.2 Prostory jmen

Ještě před samotným výkladem tříd si něco povíme o prostorech a oborech jmen v jazyce Python. Pochopení těchto mechanismů je důležité pro následné správné porozumění problematice tříd a objektů. Opět začneme definováním několika pojmů.

Jako *prostor jmen* se v Pythonu (a jiných jazycích) označuje zobrazení mezi jmény proměnných a objekty které reprezentují. Lze to chápat následujícím způsobem: objekty existují nezávisle na jejich jménech. Pokud vytvoříme nějakou proměnnou pomocí přiřazení, nevytvoříme tím nový objekt, pouze odkaz na objekt, který stojí na pravé straně přiřazení. Pokud si uvedeme následující příklad:

```
>>> d1 = {1: 'ahoj', 2: 'bla', 3: 'cvok'}
>>> d2 = d1
```

Pak skutečně ‘d1’ a ‘d2’ odkazují na tentýž objekt. Pokud nyní modifikujeme jeden z těchto objektů, změna se projeví i ve "druhém":

```
>>> d2[3] = 'cyril'
>>> print d1
{3: 'cyril', 2: 'bla', 1: 'ahoj'}
```

Jména lze tedy chápat i jako pouhé odkazy na objekty. Proto pokud předáváme nějaký objekt libovolné funkci, vždy se předává odkazem. Pokud tento objekt modifikujeme, změna se projeví i ve volající funkci.

Prostorů jmen existuje v prostředí Pythonu mnoho. Např. každý modul má svůj prostor jmen, každá definice třídy zavádí také nový prostor jmen, při každém volání funkce dojde k vytvoření nového (dočasného) prostoru jmen, dokonce každý objekt má svůj prostor jmen určený pro uchování jeho atributů atd. Jména uvnitř jednoho prostoru jmen musí být *jedinečná*.[2] V různých prostorech jmen se ovšem mohou nacházet stejná jména odkazující na různé objekty.

Při každém vyvolání funkce je vytvořen nový prostor jmen, do něhož se ukládají lokální proměnné a formální parametry. Tento prostor jmen zanikne po ukončení funkce. Při rekurzivním vyvolání funkce vznikne nový prostor jmen, tudíž jednotlivá volání *nesdílí* společné objekty.

Jak jsme si již řekli, objekty mohou mít mnoho *atributů*, jejichž jména má objekt uložen ve svém prostoru jmen. Z hlediska přístupu k atributům jsou všechny atributy *veřejné* a dále se člení na atributy *pouze pro čtení* a atributy *zapisovatelné*. To, o jaký typ atributu se jedná se dozvíte většinou z dokumentace tohoto objektu. Například všechny atributy modulů (tj. obsah modulu) je zapisovatelný, tudíž kód může měnit všechny atributy - hodnoty uvnitř tohoto modulu. Obecně je možné zapisovatelný atribut také smazat příkazem `del`.

Protože každé jméno proměnné je součástí nějakého prostoru jmen, používá Python tzv. *obory jmen*. Každému oboru jmen je přiřazem jmenný prostor a všechna jména z tohoto prostoru jmen je možné díky tomu, že jsou součástí oboru jmen, používat bez *kvalifikace*, tj. bez uvedení prostoru jmen. I když to zní poněkud krkolomně, je za tím skryto logické chování všech proměnných. Pokud programátor někde napíše jméno *promenna*, aniž by uvedl, že se jedná o atribut nějakého objektu, např:

```
>>> vysledek = 3 * promenna
```

začne interpretr prohledávat všechny obory jmen v přesně definovaném pořadí. Poněvadž obor jmen není nic jiného než jmenný prostor, může obsahovat proměnné. Pokud první obor jmen hledanou proměnnou neobsahuje, je prohledáván druhý obor jmen a pak další atd.

V počátcích jazyka Python existovaly přesně vymezené tři obory jmen: *lokální*, *globální* a *interních jmen*. Jim přiřazené prostory jmen se liší v závislosti na právě prováděném kódu. Tak třeba při vyvolání funkce obsahuje lokální obor jmen lokální jmenný prostor funkce, čili obsahuje lokální proměnné, globální obor jmen pak odpovídá jmennému prostoru modulu, který danou funkci obsahuje, čili definuje globální proměnné. Interní obor jmen vždy obsahuje jmenný prostor modulu `__builtin__`, který definuje interní funkce (např. `abs()` nebo `file()`).

Každé přiřazení vytváří novou proměnnou v lokálním oboru jmen, uvnitř funkce tedy každé přiřazení vytváří novou lokální proměnnou. Zde si všimněte jisté asymetrie, přestože Python umožňuje používání jak lokálních, tak globálních a interních jmen zároveň, každé přiřazení vytvoří pouze lokální proměnnou, nelze proto přiřadit hodnotu globální proměnné nebo dokonce interní.

K tomu je v Pythonu příkaz `global`[3], jenž umožňuje specifikovat, že tato proměnná je globální a tudíž i každé přiřazení této proměnné změní hodnotu odpovídající proměnné v globálním oboru jmen:

[2] Ve skutečnosti je většina prostorů jmen implementováno jako datový typ slovník, jehož klíče tvoří jména proměnných a hodnoty jsou objekty, ne něž tyto proměnné odkazují

[3] Spíše než o příkaz se jedná o direktivu.

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

Vše co jsme si řekli o přiřazení platí i o jiných příkazech, které vytvářejí nová jména proměnných. Například jde o příkaz `import` nebo o příkazy vytvářející nové pojmenované objekty jako `def` nebo `class`. Obdobně se chová i příkaz pro odstranění jména proměnné `del`.

V současnosti Python používá novou architekturu oborů jmen, kdy ke standardním třem oborů přibyl čtvrtý, který je umístěn mezi lokální a globální. Jde o tzv. *nadřízený* obor jmen. Představte si následující situaci: máme funkci `f()`, která ve svém těle definuje funkci `g()`. Chceme-li nyní, aby funkce `g()` používala proměnné z nadřízené funkce `f()`, budeme k nim přistupovat zrovna přes nadřízený obor jmen. Jde tudíž o obor jmen obsahující prostory jmen volajících funkcí.

9.3 Třídy poprvé ...

Abychom mohli používat své vlastní objekty (v mluvě objektově orientovaných jazyků se jim říká *instance tříd*) musíme si nejprve nadefinovat *třídu*, z níž lze poté jednoduchým způsobem vytvořit instanci.

V některých jazycích se třída chová podobně jako datový typ, v Pythonu však třída je klasický objekt, stejně se s ní pracuje a všechno co platí pro objekty platí i pro třídy.

Co je pro třídy ale obzvláště důležité – dávají všem instancím *stejné vlastnosti*. V praxi to znamená, že např. metodu si nadefinujeme uvnitř třídy a automaticky jí získají i všechny instance této třídy. Názorně si tuto situaci můžete představit třeba takto: máme obecnou třídu `auto`, která definuje společná vlastnosti, třeba metodu `jezdi()` nebo `tankuj()`. Ve skutečnosti je ale třída `auto` jakási automobilka, teprve její instance jsou skutečná auta. Automobilka pouze určí, jak se budou auta chovat (to odpovídá implementování metod) a až samotná auto budou toto chování vykonávat (což neznamená nic jiného zavolání metody na nějaké instanci).

Třída je tedy jakási šablona určující chování všem jejím instancím. Proto lze změnou definice třídy dosáhnou změny chování všech její instancí – objektů.

9.3.1 Definování tříd

Python pro definici třídy zavádí nový příkaz `class`, který vytvoří novou třídu. Příkaz `class` se chová podobně jako definice funkce, která vytvoří funkci z příkazů, které obsahuje její tělo. Obdobně i příkaz pro vytvoření třídy, který vytvoří novou třídy se zadaným jménem, jejíž chování (metody, atributy apod.) budou určovat příkazy v jejím těle:

```
class Jméno_třídy:
    <příkaz-1>
    .
    .
    .
    <příkaz-N>
```

Podobně jako u funkcí, i u tříd musí být před prvním použití třídy tato třída zadefinována. Jinými slovy, musí být spuštěna její definice. Z toho nám plynou některé užitečné vlastnosti Pythonu, například definici třídy můžeme umístit do větve příkazu `if` a podle nějaké podmínky se můžeme rozhodnout, jak třídu definujeme.

Uvnitř těla definice třídy jsou specifické příkazy, kterými se definuje chování třídy. Většinou se jedná o definice funkcí, uvnitř těla třídy se ale příkaz `def` chová poněkud odlišněji od standardního příkazu definice funkce, tak jak ho známe z předchozího povídání. Nenabývejte ale dojmu, že zde není povoleno nic více než definice metod, Python dovoluje do těla třídy umístit libovolné příkazy, které budou vykonány při spuštění definice této třídy.

Při vykonání definice třídy je vytvořen nový prostor jmen, který je použit jako lokální obor jmen uvnitř těla třídy. Z vlastností oborů jmen je nám již jasné, že všechna jména proměnných, která uvnitř těla třídy vytvoříme budou umístěna v prostoru jmen této třídy.

Pokud se definice třídy provede normálně (tj. nedojde v těle k žádné výjice), pak je vytvořen *objekt třídy*, čili Pythonový objekt reprezentující tuto třídu. Objekt třídy je uložen pod jménem, které bylo zadáno při definici funkce. Tento objekt, kromě jiného, vytváří jakýsi obal nad vytvořeným prostorem jmen. Po vytvoření objektu třídy je obnoven lokální obor jmen, který je nastaven zpět tak, aby obsahoval stejný obor jmen jako před spuštěním definice funkce.

9.3.2 Objekty typu třída

Třídní objekty obsahují, jak již bylo řečeno, prostor jmen uchovávající objekty, které byly definovány v těle funkce. Kromě toho ale podporuje i další dva druhy operací – *zpřístupnění atributů* a *vytvoření instance*.

Nejprve se budeme věnovat atributům třídy. Jak jsme si řekli hned na začátku této kapitoly, každý objekt může mít mnoho atributů a ani třídy nejsou výjimkou. Jména atributů odpovídají jménům ve jmenném prostoru třídy. Pokud tedy budeme uvažovat třídu `MojeTřída`, která je definována takto:

```
class MojeTřída:
    "Jednoduchá třída"
    i = 12345
    def f(self):
        return 'ahoj'
```

pak `MojeTřída.i` a `MojeTřída.f` jsou atributy třídy `MojeTřída`. První z nich odkazuje na objekt čísla 12345 a druhý na funkci `f`. Všechny atributy třídy jsou jak pro čtení, tak je do nich umožněn i zápis.

Všimněte si řetězce, který je uveden na druhém řádku definice třídy. Jde o dokumentační řetězec, který funguje úplně stejným způsobem jako dokumentační řetězec nějaké funkce. Tento řetězec je možné zpřístupnit jako atribut `__doc__`.

Vytváření instancí používá zápis funkčního volání. Nenechte se však zmýlit, operátor volání funkce a vytvoření instance má úplně jiný význam a funkci. Pokud použijeme třídu `MojeTřída` a její definici, kterou jsme si uvedli výše, pak instanci této třídy můžeme vytvořit pomocí příkazu:

```
x = MojeTřída()
```

Nyní již bude jméno `x` odkazovat na *instanci třídy*. Tato instance, jinými slovy též *instanční objekt* je prázdná, nemá

nastaveny žádné atributy. Pokud bychom tento objekt chtěli hned po vytvoření nastavit do nějakého stavu, jinými slovy nastavit nějaké jeho atributy, můžeme definovat speciální metodu s názvem `__init__()`. Prozatím bych poprosil čtenáře o trochu shovívavosti, pojem *metoda* si rozvedeme dále v této kapitole.

```
def __init__(self):
    self.data = []
```

Metoda (funkce) `__init__()` je zavolána při vytvoření instance třídy, přičemž za argument *self* je dosazena tato instance. Novou ziniciovanou instanci získáme opět pomocí objektu třídy:

```
x = MojeTřída()
```

Metoda `__init__()` nám nabízí i další možnost – definovat ji s více argumenty, pak budou tyto argumenty vyžadovány při vytvoření instance, budeme je tedy muset předat objektu třídy, který následně zavolá metodu `__init__()` a všechny argumenty jí předá:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Instančí objekty

Již máme vytvořeny instance třídy. Škála operací, kterou nám instance nabízí je ale ještě užší než umožňují třídy, jde o pouhé zpřístupnění atributů. Jazyk Python rozlišuje dva druhy atributů, jednak jde o *datové atributy* a pak o tzv. *metody*.

Datové atributy

Datové atributy se nazývají v terminologii různých objektových jazyků různě, např. Smalltalk je nazývá "instanční proměnné" a C++ "datové členy". Datové atributy vznikají podobně jako proměnné – při definici, tj. při prním přiřazení nějaké hodnoty jejich jménu. Pokud budeme pokračovat v předchozím příkladu s třídou `MojeTřída` a budeme předpokládat, že *x* je jméno odkazující na instanci této třídy, pak následující fragment kódu vytiskne hodnotu `16`:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Metody

Kromě datových atributů rozlišuje Python ještě *metody*. Již několikrát v této publikaci zaznělo, že metoda je funkce patřící k určitému objektu. O pár řádků výše jsme si řekli, že i instance jsou objekty a proto třídy umožňují definování

metod, které se chovají naprosto stejně jako metody jiných objektů (např. seznamů).

Jména metod se specifikují v definici třídy. Python při rozlišování mezi datovými atributy a metodami postupuje velice jednoduše. Pokud je nějaký objekt v těle třídy funkce, pak jde o metodu, jinak jde o datový atribut.

Pokud se opět zaměříme na příklad s instancí `x` třídy `MojeTřída`, potom `x.f` je metoda, protože třída `MojeTřída` definuje funkci `f`, oproti tomu `x.i` je datový atribut, jelikož tělo třídy definuje `i` jako číslo `12345`.

Dejte si pozor na následující: Zatímco `x.f` je metoda `f` instance `x`, `MojeTřída.f` je funkce. Zapamatujte si proto, že o metodu se jedná pouze v případě, kdy k této metodě existuje i instance. Funkce `MojeTřída.f` je pouze jakýsi prototyp metody `x.f`, metoda z ní vznikne spojením s instancí.

Každá metoda se chová stejně jako funkce (jediný rozdíl oproti funkci je její provázanost s instancí), můžeme jí tedy volat jako funkci:

```
x.f()
```

Protože `MojeTřída.f` kdykoli vrací řetězec `'ahoj'`, pak i tato metoda vrátí `'ahoj'`. Metodu ale nemusíme volat přímo, můžeme jí uložit do nějaké proměnné a zavolat později. Zde si, prosím, uvědomte, že instance, pro kterou bude metoda volána se *neurčuje* podle objektu "před tečkou", ale je pevně zapsán v té které metodě. Proto je možné metody uchovávat pod jinými jmény a vůbec, pracovat s nimi jakoby to byly obyčejné funkce, což nejlépe demonstruje následující příklad:

```
xf = x.f
while 1:
    print xf()
```

Podívejme se tedy na to, co se stane, je-li metoda zavolána. Protože Python provádí roztřídění mezi datovými atributy a metodami ihned při vytvoření instance, má již v tuto chvíli "jasno" ohledně toho, že je 100% volána metoda.

Každá metoda je složena z odpovídající funkce a instance třídy. Běhové prostředí Pythonu proto z metody tyto dvě vlastnosti získá a následně zavolá právě vyextrahovanou funkci s těmito atributy: první bude instance (je vám již jasno, proč každá metoda má první argument `self`?) a pak následují ty atributy, které byly předány při volání metody.

Proto je nutné každou metodu definovat s jedním argumentem "navíc". Za tento atribut se pak při volání metody dosadí instance, pro kterou byla metoda zavolána. A co víc, pomocí tohoto argumentu se odkazujeme na datové atributy instance. Instancí totiž může být vytvořeno mnoho, ale při běhu metody díky výše popsanému mechanismu máme jistotu, že vždy používáme tu správnou instanci (tj. tu, pro níž jsme metodu volali).

Python dokonce umožňuje toto rozložení na instanci a funkci nechat na programátorovi, který může tuto funkci zavolat a místo prvního argumentu jí předat rovnou instanci:

```
x.f()
MojeTřída.f(x)
```

Tyto dva zápisy jsou ekvivalentní. Dobře si je zapamatujte, protože v některých situacích musíte metody volat takto explicitně, protože chcete použít jinou funkci, než kterou by použil interpretr.

Poznamenejme, že jako první argument můžete funkci, která de facto reprezentuje metodu předat pouze instanci její třídy. Pokud se pokusíte o jakýkoli jiný postup, nekompromisně dojde k výjimce:

```
MojeTřída.f('fooo')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unbound method f() must be called with instance as first argument
```

9.4 Třídy podruhé ...

Nyní následují náhodné poznámky, které se (převážně) týkají problematiky objektově orientovaného programování v jazyce Python.

Poněvadž datové atributy a metody sídlí ve stejném jmenném prostoru, dojde při přiřazení nějaké hodnoty nějakému jménu uvnitř těla promněnné k přepsání metody, která měla stejné jméno. To nejlépe demostruje následující fragment kódu:

```
class Trida:
    def foo(self, x):
        return x + 1

foo = 'novy text'
```

Pak již atribut instance třídy `Trida` neobsahují metodu `foo`, nýbrž datový atribut třídy `Trida.foo`. Proto je vhodné používat určitou konvenci pro pojmenovávání datových členů a metod. Podle jedné z nich můžete např. datové atributy psát s malým a metody s velkým počátečním písmenem. Další možností je pojmenovávat datové atributy podstatnými jmény a metody slovesy. Podobných konvencí existuje mnoho a jistě si brzy osvojíte tu "svoji".

Datové atributy nějaké instance jsou přístupné jak metodám této instance, tak i jakémukoli jejímu uživateli (*klientovi*). Python nepodporuje modifikátory atributů, kterými by bylo možné určité datové atributy prohlásit za soukromé. Je možné používat pouze pseudo-soukromé atributy o nichž si budeme povídat dále v této kapitole.

Z předchozího odstavce plyne i nemožnost vytvoření čistě abstraktních datových typů v jazyce Python, protože žádná třída nedokáže skrýt svoje atributy před zraky ostatních objektů. Pokud však tuto třídu přepíšete do C, implementační detaily se skryjí a používá se pouze rozhraní této třídy. V jazyce C se ale programuje velice nepohodlně (v porovnání s Pythonem) a tudíž se tento krok nedoporučuje.

Přestože všechny datové atributy instancí jsou veřejné, měli by je klientské objekty používat s obezřetností. Klient totiž může porušit složité závislosti mezi daty uvnitř instance a metody instance by pak nemusely pracovat správně. Pokud ovšem víte, že váš přístup nenaruší konzistenci instančních dat, pak vám nikdo nebrání tyto datové atributy modifikovat a dosáhnout tak požadovaného chování objektů. Klienti instance dokonce mohou do jejího jmenného prostoru ukládat své vlastní atributy, čímž lze např. označkovat určité instance apod. Jediným předpokladem je, že použijete unikátní jméno atributu. Tím se vyhnete kolizi s datovými atributy instance.

Jak jsme si již jednou řekli, v Pythonu neexistuje žádná zkratka, kterou se lze odkazovat na datové atributy, případně jiné metody z metody této instance. Vždy musíte tyto atributy a metody kvalifikovat, což neznamená nic jiného, než před ně uvést jméno `self` a tečku. Toto opatření vysoce zvyšuje čitelnost kódu, i při letmém pohledu na zdrojový kód nelze zaměnit lokální a instanční proměnnou.

Podle ustálených konvencí je první argument metod vždy pojmenován `self`. Přestože jde pouze o konvenci a nikdo vám nebrání v používání vlastního jména, nedoporučuje se to. Kdykoli, kdy tuto konvenci porušíte, přinejmenším zmatete čtenáře zdrojového kódu. Některé programy, které umožňují interaktivně procházet datovými strukturami a objekty definovanými v prostředí interpretru, mohou být také zmateny.

Každé jméno uložené ve jmenném prostoru třídy a odkazující na funkci, znamená zavedení nové metody pro instance této třídy. Přitom není nutné, aby tato funkce byla přímo uložena v těle definice třídy. Lze jí například definovat před třídou samotnou a v těle třídy pouze tento funkční objekt přiřadit nějakému jménu:

```
# Funkce definovaná mimo třídu
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'ahoj'
    h = g
```

Po provedení těchto definici jsou `f`, `g` a `h` atributy třídy `C`, odkazují však na jedinou funkci `f1`. Zároveň jsou `f`, `g` a `h` jména metod instancí třídy `C`. Každá z těchto metod však má stejnou implementaci — funkci `f1`. Těmto praktikám se pokud možno vyhněte, protože dokáží dokonale zmást, v některých případech je ale jejich použití ospravedlnitelné zjednodušením kódu případně zvýšením funkčnosti výsledného programu.

Jak již bylo několikrát řečeno, jedna metoda může volat druhou pomocí argumentu `self`. Ten reprezentuje instanci pro níž byla metoda zavolána a tudíž obsahuje všechny jeho metody:

```
class Bagl:
    def __init__(self):
        self.data = []
    def pridej(self, x):
        self.data.append(x)
    def pridej_dvakrat(self, x):
        self.pridej(x)
        self.pridej(x)
```

Protože kromě speciální chování metod, které umožňuje jejich provázání s instancemi jde o obyčejné funkce, lze z nich používat i globální proměnné a další vymoženosti funkcí jako jsou argumenty pro proměnný počet parametrů apod. Podobně jako u funkcí, i metody mají jako svůj globální obor jmen jmenný prostor modulu, jež obsahuje definici třídy (zde pozor na častý omyl: jmenný prostor třídy nikdy *není* globálním oborem jmen!).

9.5 Třídy potřetí (Dědičnost) ...

Jedna ze tří základních vlastností objektově orientovaného programování se nazývá *dědičnost*. Protože tato publikace nechce být učebnicí programování, ale pouze průvodcem jazykem Python, nebudeme zde zacházet do detailů, to ponecháme fundovaným učebnicím objektově orientovaného programování.

Proto si pouze řekneme, co nám dědičnost umožňuje. Jedná se vlastně o odvození nové třídy z nějaké původní, přičemž nová a původní třída jsou v určitém vztahu a předpokládá se, že odvozená třída definuje všechny metody svého předka a ještě může definovat některé metody navíc. Tyto nově definované metody mohou rozšiřovat nebo dokonce úplně předefinovávat chování předka.

Pokud tedy budeme uvažovat třídu `Bagl` ze závěru předchozí části této kapitoly a budeme jí chtít rozšířit tak, aby při přidání nějakého nového prvku vytiskla ještě hlášení o tom, jaký prvek přidává, použijeme následující definici třídy:

```
class Bagl_s_tiskem(Bagl):
    def __init__(self):
        print 'iniciuji bagl'
        Bagl.__init__(self)

    def pridej(self, x):
        print 'pridavam prvek %s' % x
        Bagl.pridej(self, x)

    def pridej_dvakrat(self, x):
        print 'pridavam dvakrat:'
        Bagl.pridej_dvakrat(self, x)

    def pridej_trikrat(self, x):
        print 'pridavam trikrat:'
        self.pridej_dvakrat(x)
        self.pridej(x)
```

Tím jsme definovali novou třídu, která je odvozená od třídy `Bagl` (viz hlavička definice). V této třídě jsme předefinovali chování původních metod tak, že nejprve vytisknou hlášení a poté zavolají metodu tak, jak byla definována v předkovi (viz zápis pomocí atributu nadřízené třídy '`Bagl.pridej(self, x)`').

Velice důležitou vlastností, kterou by mělo podporovat také každé objektově orientované prostředí je *polymorfismus*. V Pythonu je jeho podpora samozřejmostí. Proto si všimněte metody `Bagl_s_tiskem.pridej_dvakrat()`. Ta nejprve vytiskne klasické hlášení a posléze zavolá metodu svého předka. Nyní se podívejte na to, jak je definována metoda `Bagl.pridej_dvakrat()` — dvakrát zavolá metodu `pridej()`. Tato metoda je ovšem hledána v instanci, kde se jako první narazí na metodu `Bagl_s_tiskem.pridej()`. Tím vlastně původní kód vyvolá a použije naší *novou* metodu. Podařilo se nám ji dokonale předefinovat.

Hledání atributů (je jedno jestli datových atributů nebo metod) instance se děje velice jednoduchým způsobem. Nejprve se prohledá třída této instance, pokud je atribut definován zde, končíme a použije se jeho hodnota. Pokud se zde atribut nenajde, zkusí Python prohledat předky této třídy, pokud nějací existují. Existuje totiž možnost, že nějaká metoda, kterou v potomkovi nechceme předefinovat, bude definována v předkovi. A protože předek opět může být potomkem nějakého prapředka, zkusí se atribut najít v případě neúspěchu i v tomto prapředkovi. Pokud se dospěje ke kořeni stromu tříd, ale žádný odpovídající atribut není nalezen, dojde k výjimce `AttributeError`.

Protože metoda definovaná v potomkovi většinou pouze *rozšiřuje* funkčnost původní metody, je často nutné volat metodu předka. Python nepodporuje přímé vyvolání metody předka, tudíž musíte použít druhou možnost volání metody, který využívá funkčního objektu odpovídajícího metodě. V potomkovi proto zavoláme tuto funkci definovanou v předkovi a jako argument mu předáme argument `self` a další argumenty. Takto nemusíme volat přímo metodu přímého předka, postup lze použít i na prapředky apod.

Naše vzorová odvozená třída dokonce rozšiřuje paletu metod svých instancí, protože definuje metodu `pridej_trikrat()`, která ke své implementaci využívá metody `pridej()` a `pridej_dvakrat()`. Obecně lze říci, že instance zděděné třídy je svojí funkčností *nadmnožinou* instance svého předka, což znamená, že definuje vše co předek a ještě nějaké metody nebo datové atributy navíc. Proto lze většinou instance potomka používat na místech, kde jsou očekávány instance předka.

Je-li nějaký objekt instancí určité třídy nám umožňuje zjistit interní funkce `isinstance()`, které jako první argument předáme objekt a druhý argument třídu. Funkce pak vrátí 1, jestliže objekt je instancí třídy, 0 je-li tomu jinak. Zároveň platí, že instance nějaké odvozené třídy je vždy i instancí rodičovské třídy:

```
>>> batoh1 = Bagl()
>>> batoh2 = Bagl_s_tiskem()
>>> isinstance(batoh1, Bagl)
1
>>> isinstance(batoh2, Bagl)
1
>>> isinstance(batoh1, Bagl_s_tiskem)
0
>>> isinstance(batoh2, Bagl_s_tiskem)
1
```

9.5.1 Vícenásobná dědičnost

V Pythonu je samozřejmostí vícenásobná dědičnost. Třídu s více předky definujeme podobně jako třídu s jedním rodičem:

```
class OdvozenáTřída(Rodič1, Rodič2, Rodič3):
    <příkaz-1>
    .
    .
    .
    <příkaz-N>
```

Základní a jediné pravidlo, které musíte uvažovat při práci s instancemi tříd, které mají více předků je *"nejprve celý strom, až pak další třídy zleva doprava"*. Tato magická věta nám dáva tušení o tom, jak funguje vyhledávání atributů u tříd s více předky. Pokud totiž atribut není nalezen ve třídě této instance, hledá se u jejích předků. Výše uvedené pravidlo pak určuje pořadí v jakém jsou prohledávány rodičovské třídy. První část věty znamená jednoduše, že nejprve Python bude prohledávat *první* rodičovskou třídy (v našem případě `Rodič1`) a *její předky*. Až pokud daný atribut není nalezen (může jím být jak datový člen tak metoda), prohledává se druhá *rodičovská* třída (`Rodič2`) a *její předci*. A v případě selhání i zde se pokračuje poslední *rodičovskou* třídou (`Rodič3`) a *jejími předky*. Může nastat situace, kdy daný atribut není nalezen ani v této třídě, pak ale dojde k výjimce `AttributeError` a běh programu se přeruší.

Součástí každé učebnice, která se alespoň okrajově zmiňuje o objektově orientovaném programování v libovolném jazyce, jenž podporuje vícenásobnou dědičnost, je téměř vždy varování před zneužíváním tohoto mechanismu. Ani naše publikace v tomto směru nebude výjimkou. Nadměrné používání vícenásobné dědičnosti vede dříve nebo později ke zmatení jak čtenářů kódu, tak samotného programátora. Platí zde více než kdekoli jinde nutnost dodržování předem daných konvencí. Jedině tak lze udržet kód přehledný a snadno udržovatelný. Dalším známým problémem je tzv. *diamantová struktura* tříd, jak je nazýván případ, kdy od jednoho předka jsou odvozeny dvě třídy a od těchto dvou tříd zároveň je odvozena třetí. V tomto případě si s mechanismy zde popsanými těžko pomůžete a na řadu musí přijít nějaká jiná, sofistikovanější metoda.

9.6 Pseudo-soukromé atributy

Jak již několikráte zaznělo v textu výše, Python podporuje tzv. pseudo-soukromé atributy. Jako pseudo-soukromý atribut je používán každý identifikátor ve tvaru `__identifikátor`. Tento speciální zápis je tvořen jménem identifikátoru, jemuž předcházejí minimálně dvě podtržítka a je ukončeno nejvýše jedním podtržítkem. Každý výskyt takového identifikátoru je nahrazeno novým identifikátorem `_jménotřídy_identifikátor`, kde `jménotřídy` je jméno aktuální třídy. Jako aktuální třída se použije třída, v jejímž kódu (tj. kódu její metody) se takovýto zápis vyskytl. Takto lze zajistit, že dvě různé třídy mohou definovat dva stejné soukromé atributy a přesto nedojde ke kolizi identifikátorů. Pomocí pseudo-soukromých atributů můžeme pojmenovávat jak datové členy, tak i metody. Co je však velmi důležité - tento mechanismus můžeme použít i pro uložení určitých informací instance jedné třídy uvnitř in-

stance druhé. Nakonec poznamenejme, že mimo kód třídy stejně jako v případě, kdy jméno třídy je tvořeno samými podtržítky, k žádnému nahrazování nedojde.

Název pseudo-soukromé atributy se používá proto, že přestože dojde ke změně jména, je toto jméno stále přístupné jako `_jménotřídy_identifikátor` a to pro úplně libovolný kód. Toho můžete využít například v případě, kdy chcete při ladění programu zkontrolovat hodnotu určité proměnné.[4]

Je třeba podotknout, že kód předaný příkazu `exec` a funkcím pro dynamické vykonávání kódu nepovažuje jméno volající třídy za jméno aktuální třídy a tudíž se neprovádí rozšiřování identifikátorů. S obdobným problémem se potkáte v případě používání funkcí `getattr()`, `setattr()` a `delattr()`, podobně se chová i přístup ke jmennému prostoru nějaké instance pomocí atributu `__dict__`. V těchto případech vždy musíte provést nahrazení ručně!

9.7 Poznámky

Někdy se programátorovi může hodit datový typ podobný pascalovskému záznamu nebo strukturám v jazyce C. Tyto typy obsahují několik datových členů k nimž se přistupuje na základě jejich jmen. Tento typ získáme jednoduše definováním prázdné třídy:

```
class Zamestnanec:
    pass

honza = Zamestnanec() # Vytvoří prázdný záznam o zaměstnanci

# Vyplní jednotlivé položky záznamu
honza.jmeno = 'Honza Švec'
honza.skupina = 'PyCZ'
honza.plat = 0
```

Kód, který předpokládá, že mu předáte určitý abstraktní datový typ (tj. typ u nějž je známo pouze rozhraní a nezáleží na jeho implementaci) lze často "ošálit" a předat mu instanci jiné třídy, která však implementuje požadované metody. Například si představte funkci, které předáte souborový objekt, ona pomocí jeho metody `read()` přečte jeho obsah a vrátí ho jako řetězec, v němž nahradí všechny výskyty tabulátoru čtyřmi mezerami. Pak místo souborového objektu můžeme předat i libovolnou jinou třídu, která má metodu `read()` definovánu stejným způsobem jako souborový objekt.

Objekt metody, která je svázána s určitou instancí má také svoje vlastní datové atributy. Jsou jimi `im_self`, který reprezentuje instanci, na níž se metoda váže a `im_func` reprezentující funkční objekt, který metodu implementuje. Jde o funkci, která je definována v těle třídy a která se vyvolá při spuštění metody.

9.7.1 Výjimky mohou být třídy

Uživatelsky definované výjimky již v Pythonu dávno nejsou omezeny pouze na řetězce. Nesrovnatelně lépe totiž mohou posloužit třídy. Díky nim můžeme vytvořit rozsáhlou hiearchii výjimek, ve které se mezi třídami dědí atributy a jejich případné metody apod.

Kvůli možnosti používání tříd byl rozšířen i příkaz `raise`. Jeho syntaxe nyní vypadá následovně:

```
raise třída, instance

raise instance
```

V prvním případě musí být proměnná `instance` instancí třídy `třída` nebo třídy, která je od ní odvozená. Druhý

[4] To je také jediný důvod, proč jsou pseudo-soukromé atributy veřejně přístupné. Nebyl by totiž problém je úplně skrýt před nepovolanými zraky.

tvar je pak zkráceninou zápisu:

```
raise instance.__class__, instance
```

Větve `except` příkazu `try` mohou uvádět třídy stejným způsobem jako řetězce. Daný handler se spustí, pokud výjimka je instancí jedné ze tříd uvedených za příslušným klíčovým slovem `except`. Pro ilustraci můžeme uvést následující vyumělkovaný příklad. Ten vytiskne písmena '`B`', '`C`' a '`D`' přesně v tomto pořadí:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Pokud bychom větve `except` zapsali v opačném pořadí (tj. '`except B`' jako první), vytisklo by se '`B`', '`B`', '`B`', protože vždy by se spustil pouze první handler!

Pokud výjimka nebude odchycena a byde prohlášena za neobslouženou výjimku, bude jako popis chyby nejprve uvedeno jméno třídy následované dvojtečkou a řetězcovou reprezentací třídy, kterou vrátí interní funkce `str()`.

KAPITOLA

DESÁTÁ

Co nyní?

Pokud jste dočetli až sem, lze jen doufat, že váš zájem o jazyk Python ještě více zesílil a již jste dychtiví použít Python k řešení nějakých opravdových problémů. V této kapitole si popíšeme několik informačních zdrojů, pomocí nichž můžete získat další informace o Pythonu.

V první řadě byste si měli po přečtení této učebnice alespoň prolistovat dokument *Python Library Reference* (Referenční příručka standardních knihoven). Tato příručka je kompletním zdrojem informací. Najdete v ní popis všech standardních datových typů, funkcí a modulů. Všechny tyto moduly jsou navrženy tak, aby programátorovi co nejvíce usnadnili práci s Pythonem. Mnohokrát je programátor-začátečník vskutku zaskočen množstvím knihoven a jednoduchostí jejich používání. Nejednou jsem byl svědkem otázky "Co tedy mám programovat, když 'to' už všechno obsahuje?" Jen namátkou jmenujme seznam těchto modulů a balíčků, který začíná moduly, jež zpřístupňují standardní služby operačního systému definované normou POSIX, pokračuje moduly pro čtení e-mailových schránek v systémech UNIX, získávání dokumentů protokolem HTTP, přes modul pro generování náhodných čísel, modul pro získávání voleb předaných na příkazovém řádku a konče moduly pro psaní CGI programů. Pokud se do čtení příručky standardní knihovny nechcete pustit, alespoň si ji prolistujte, získáte představu, co vše vám může Python nabídnout.

Další zdroje informací jsou webové stránky různých sdružení a organizací. Mezi nimi je na prvním místě hlavní webová stránka Pythonu na http://www.python.org. Zde naleznete zdrojové kódy, připravené binární distribuce pro nejrozšířenější operační systémy, dokumentaci a odkazy na světové webové stránky věnující se Pythonu. Domácí stránky Pythonu jsou zrcadleny na různých místech po celém světě, proto se pokuste stahovat data vždy z toho nejbližšího, ušetříte tak přenosové kapacity páteřních linek a odlehčíte vzdáleným serverům.

Pro československého čtenáře budou pravděpodobně nejbližší stránky *Českého sdružení programátorů a uživatelů jazyka Python* (zkráceně PyCZ, adresa http://py.cz). Zde naleznete nejnovější verzi této učebnice a další dokumentaci přeloženou do češtiny. PyCZ také provozuje e-mailovou konferenci python@py.cz, kompletní instrukce jak se do konference přihlásit, jak se odhlásit a archív konference najdete na stránkách PyCZ.

Dalším zajímavým sídlem je http://starship.python.net/. Jedná se o informativní webové stránky. Mnoho uživatelů Pythonu zde má domovské stránky na nichž je publikováno množství programů, utilitek a modulů určených volně ke stažení.

Anglicky mluvícím uživatelům možná přijde vhod newsová skupina comp.lang.python a její e-mailové zrcadlo python-list@python.org. Obě tyto konference jsou propojeny, všechny příspěvky zaslané do jedné jsou automaticky přeposlány do druhé. Frekvence zpráv v této konferenci je kolem 120 nových zpráv denně a zahrnují otázky a odpovědi, návrhy nových vlastností a oznámení nových modulů.

Před zasláním otázky do e-mailové konference se nejprve přesvědčte, zda podobnou otázku nepoložil někdo před vámi. Projděte si proto archív konference (http://www.python.org/pipermail/) nebo se podívejte do nejčastěji kladených otázek (FAQ, http://www.python.org/doc/FAQ.html). Jejich obdobu najdete i v adresáři 'Misc/' libovolné *zdrojové* distribuce Pythonu. Možná v nich bude i řešení právě toho vašeho problému!

DODATEK

A

Interaktivní úpravy příkazového řádku a historie příkazů

Jak jsme si již říkali na samém začátku naší učebnice, interpretr Pythonu může být zkompilován s knihovnou *GNU Readline*, která zpřístupňuje rozšířené úpravy příkazového řádku. Tuto knihovnu používá například UNIXový shell *GNU Bash*. Tuto knihovnu můžete dokonce nakonfigurovat tak, že bude používat klávesové zkratky ve stylu obou nejpoužívanějších editorů - Emacs a vi. Přestože má tato knihovna svůj vlastní manuál, zmíníme zde základy její konfigurace, jež by se mohli hodit každému začátečníkovi.

A.1 Úpravy příkazového řádku

Po celou následující kapitolu budeme předpokládat, že váš interpretr je zkompilován s podporou knihovny *Readline*.

Rozšířené úpravy příkazového řádku pro zadání vstupu pro interpretr je možné používat kdykoli, přičemž nezáleží, zda interpretr zobrazil primární nebo sekundární výzvu. Aktuální řádka můžete upravovat např. za použití příkazů, které možná znáte z programu Emacs[1]. Jde především o klávesy `C-F` (`C-B`), jež slouží pro pohyb o jeden znak vpřed (vzad). Další dvojice kláves `M-F` a `M-B` posouvají kurzor o jedno slovo vpřed resp. vzad. Pro pohyb na konec případně začátek řádku použijete klávesy `C-E` a `C-A`. Klávesou `C-K` smažete vše od kurzoru do konce řádku, `C-Y` vloží naposledy smazaný řetězec. Konečně `C-podtržítko` vrátí poslední změnu na tomto řádku.

A.2 Historie příkazů

Historie příkazů pracuje naprosto stejně jako v libovolném jiném programu, který jako svoje rozhraní používá příkazový řádek. Klávesa `C-P` vás přesune na předchozí příkaz v historii, `C-N` na následující. Každý řádek, který jste vyvolali z historie může být upraven. Po dokončení vašich úprav stiskněte klávesu `Return` a aktuální řádek bude předán interpretru k opětovnému spuštění. Pro vyhledávání v historii použijte stisk kláves `C-R`. Pomocí kláves `C-S` spustíte vyhledávání v opačném směru.

A.3 Klávesové zkratky

Klávesové zkratky a další parametry knihovny *GNU Readline* můžete nastavovat pomocí příkazů v inicializačním souboru ‘~./inputrc’. V něm jsou klávesové zkratky zapisovány ve tvaru:

[1] V dokumentaci Emacsu znamená zápis C-B současné stisknutí kláves Control a písmena B. Podobně M-B stisk klávesy Meta (Alt) a písmena B.

```
jméno-klávesy: jméno-funkce
```

nebo:

```
"řetězec": jméno-funkce
```

Různé parametry se pak nastavují příkazy

```
set parametr hodnota
```

Zde si uvedeme jednu možnou konfiguraci knihovny Readline:

```
# Preferuji úpravy ve stylu vi:
set editing-mode vi

# Úpravy na jediné řádce:
set horizontal-scroll-mode On

# Upravíme nějaké klávesové zkratky:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Knihovna Readline obvykle používá klávesu `Tab` pro doplňování jmen souborů a příkazů, v Pythonu však slouží pro vložení znaku tabulátor. Jestliže si přejete používat doplňování vázané na klávesu `Tab`, zapište do svého souboru ‘~/.inputrc’ následující řádek:

```
Tab: complete
```

Za použití tohoto namapování kláves se ale připravíte o možnost přímo vkládat znaky tabulátor na příkazové řádce interpretru. Doplňování jmen proměnných a modulů je další rozšíření, které přináší knihovny *GNU Readline* do interpretru Pythonu. Pokud si přejete dolňování jmen používat, přidejte následující řádky do inicializačního souboru Pythonu:[2]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Tento kód sváže klávesu `Tab` s funkcí doplňování jmen (*complete*). Při doplňování jmen se prohledává seznam všech příkazů jazyka Python, aktuální lokální prostor jmen a jména modulů. Pro výrazy, které používají tečkovou notaci jako např. `string.a` se nejprve vyhodnotí výraz před poslední tečkou a pak se vyvolá doplňování jmen na výsledném objektu. Takto může dojít i k nechtěnému spuštění kódu programu. Stačí aby objekt definoval metodu `__getattr__()` a při prvním doplňování složitějšího výrazu, který se odkazuje na atributy takové instance se spustí kód této metody.

Inicializační soubor může nabízet i více možností. Jako příklad může posloužit následující výpis souboru. Podotkněme, že inicializační soubor se spouští ve stejném prostoru jmen, ve kterém budou později vykonávány interaktivní příkazy. Proto je slušností zanechat toto prostředí "čisté", kvůli čemuž skript po sobě snaže nepotřebná jména z tohoto jmenného prostoru odstraní:

[2] Python po svém startu spouští obsah souboru, na nějž ukazuje proměnná prostředí PYTHONSTARTUP. Více viz druhá kapitola.

```
# Přidává doplňování příkazů a ukládá historii příkazů vašeho interaktivního
# interpretru. Vyžaduje verzi Python 2.0 nebo vyšší, readline. Doplňování
# je svázáno s klávesou Esc (tuto vazbu můžete změnit - viz dokumentace
# knihovny readline).
#
# Uložte tento soubor jako ~/.pystartup a nastavte proměnnou prostředí tak,
# aby na něj ukazovala, tj. 'export PYTHONSTARTUP=/home/honza/.pystartup'
# v bashi
#
# Proměnná PYTHONSTARTUP _neexpanduje_ znak '~', takže musíte zadat celou
# cestu k vašemu domácímu adresáři.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser('~/.pyhistory')

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

V porovnání s prvními verzemi interpretru jsou tyto možnosti přímo pohádkové. Stále ale zbývá implementovat několik dalších vlastností, které by editování příkazového řádku udělaly ještě pohodlnější než je nyní.

Artitmetika v plovoucí řádové čárce: Problémy a jejich náprava

Tato kapitola se věnuje problematice čísel v plovoucí řádové čárce. Vysvětluje specifika tohoto datového typu, která se týkají nejen jazyka Python. Také vysvětluje, jakým způsobem přistupuje Python k reálným číslům.

Jak jistě dobře víte, v počítači jsou čísla uložena ve dvojkové soustavě (tj. číselné soustavě o základu 2). Stejně jsou uložena i desetinná čísla. Ta jsou reprezentována jako zlomky se jmenovatelem, jež je roven mocnimám čísla 2. Například desetinné číslo

```
0.125
```

má hodnotu 1/10 + 2/100 + 5/1000. Jde stejné číslo reprezentuje i binární zápis

```
0.001
```

Tento zápis znamená 0/2 + 0/4 + 1/8. Oba zápisy reprezentují stejné číslo, ovšem pokaždé zapsané v jiné soustavě.

Bohužel mnoho desetinný čísel zapsaných v desítkové soustavě nemá odpovídající binární tvar. Proto jsou tato čásla uložena pouze jako *přibližná* hodnota.

Tento problém si demonstrujeme na zlomku jedna třetina. V desítkové soustavě ho můžeme napsat pouze přibližně jako desetinné číslo:

```
0.3
```

nebo lépe:

```
0.33
```

nebo ještě lépe:

```
0.333
```

a tak dále. Nezáleží na tom, kolik číslic napíšete, výsledek nikdy nebude přesně 1/3, vždy půjde o přesnější a přesnější *aproximaci* čísla 1/3.

Podobně je ve dvojkové soustavě periodická jedna desetina. Vždy půjde nekonečný periodický rozvoj

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

Použitím pouze konečného počtu číslic získáme pouze přibližnou hodnotu. Díky tomu můžeme o interpretru získat výsledky podobně tomuto:

```
>>> 0.1
0.10000000000000001
```

Jak vidíte, Python vytiskl číslo 0.1 tak, jak ho on "vidí". Na některých platformách můžeme získat i jiný výsledek. Jednotlivé stroje se totiž mezi sebou (kromě jiného) liší i počtem bitů, které používají pro aritmetiku v plovoucí řádové čárce. Proto nás nesmí překvapit i aproximace podobná této:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Interaktivní interpretr Pythonu používá pro získání řetězcové reprezentace nějaké hodnoty interní funkce `repr()`. Ta desetinná čásla zaoukrouhluje na 17 platných cifer.[1] Tato cifra byla vypočítána programátory Pythonu, pokud by použili 16 cifer, nebyla by již výše uvedená podmínka platná pro všechna reálná čísla.

Zapamatujte si, že problém s reprezentací reálných čísel není problémem Pythonu. Také nejde o chybu ve vašem kódu. Se stejnými problémy se setkáte i v jiných jazycích, které používají klasické matematické funkce. Tyto jazyky však mohou být napsány tak, že rozdíly mezi jednotlivými reprezentacemi *nemusí* implicitně *zobrazovat*.

Oproti funkci `repr()` zde existuje i interní funkce `str()`, která vrací reálné číslo s přesností na 12 platných číslic. V některých případech ji proto můžete používat namísto funkce `repr()`, poněvadž její výstup vypadá v mnoha případech lépe:

```
>>> print str(0.1)
0.1
```

Musíte mít na paměti, že to co vidíte je jakási "iluze", hodnota ve skutečnosti není přesně 1/10, ale její aproximace *zaokrouhlená* na 12 platných číslic.

Můžete se dočkat i dalších překvapení. Pokud nepoučený programátor napíše 0.1 a uvidí následující výsledek

```
>>> 0.1
0.10000000000000001
```

může se pokusit výsledek zaokrouhlit pomocí funkce `round()`. Ta ale neprovede žádné zaokrouhlení!

```
>>> round(0.1, 1)
0.10000000000000001
```

V čem je problém? Binární desetinné číslo s hodnotou 0.1 je uloženo jako nejlepší aproximace zlomku 1/10. Pokusíte-li se jí nyní zaokrouhlit, lepší aproximaci již nezískáte!

Jiné překvapení — 0.1 není přesně 1/10! Sečtete-li tedy desetkrát číslo 0.1, nedostanete 1.0:

[1] 17 cifer je použito aby byla splněna rovnost `eval(repr(`*x*`)) ==` *x*.

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
```

Binární reprezentace reálných čísel obnáší ještě další podobné problémy. O některých jejich důsledcích a nápravách se dočtete na dalších řádcích. Kompletní výčet možných "chyb" najdete na adrese http://www.lahey.com/float.html.

Uvědomte si, že na tyto problémy nikdy neexistuje jednoznačná náprava. Aritmetiky v plovoucí řádové čárce se ale bát nemusíte! Chyby v plovoucí řádové čárce jsou závislé na použitém počítači a jejich výskyt je poměrně sporadický. To je pro většinu úloh naprosto postačující okolnost.

Vždy se ale snažte jednotlivé operace skládat tak, aby nedocházelo ke zvětšování chyby. Konkrétní postup se liší úlohu od úlohy. Pokud jste již zkušenější programátor, jistě si dokážete sami pomoci. A začátečníky nezbývá než odkázat na příslušnou literaturu či internetové stránky.

B.1 Chyby v reprezentaci čísel

Této podkapitole se blíže podíváme na to, jak jsou v počítači interně ukládána reálná čísla a jakým způsobem se vyvarovat většině problémů spojených s používáním reálných čísel na dnešních počítačových platformách.

Chyba v reprezentaci čísel znamená, že některá desetinná čísla v desítkové soustavě přesně neodpovídají binárním desetinným číslům. To je důvod proč někdy Python (nebo Perl, C, C++, Java, Fortran a další jazyky) nezobrazí přesně to číslo, které jste očekávali:

```
>>> 0.1
0.10000000000000001
```

V čem je problém? Jak jsme si již řekli, číslo 0.1 nemá v binární soustavě odpovídající zápis. V dnešní době téměř všechny počítače používají reálnou aritmetiku podle standardu IEEE-754 a téměř na všech platformách odpovídá Pythonovský datový typ float číslu s dvojitou přesností.

Tato čísla obsahují 53 bitů určených k reprezentaci hodnoty čísla (*matisa*). Při ukládání reálné hodnoty se tato v počítači nejprve přetransformuje na zlomek ve tvaru *J*/2***N*, kde *J* je celé číslo obsahující přesně 53 bitů. Přepíšeme-li

```
1 / 10 \~= J / (2**N)
```

jako

```
J \~= 2**N / 10
```

a za předpokladu, že *J* má přesně 53 bitů (tj. platí `>= 2**53` ale `< 2**53`), je nejlepší hodnotou pro *N* 56.

```
>>> 2L**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

56 je jediná hodnota *N*, která zajistí, že *J* je dlouhá přesně 53 bitů. Nejlepší možnou hodnotu *J* získáme zaokrouhlením:

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

Jelikož je zbytek větší než polovina z 10, nejlepší aproximaci získáme zaokrouhlením nahoru:

```
>>> q+1
7205759403792794L
```

Takže nejlepší možnou reprezentací 1/10 v IEEE-754 typu double precision je tato hodnota vydělená 2**56, tedy

```
7205759403792794 / 72057594037927936
```

Protože jsme zaokrouhlovali nahoru, je výsledek o trochu větší než 1/10. Pokud bychom zaokrouhlení neprovedli, výsledek by byl trochu menší než 1/10. Ale nikdy nebude přesně 1/10!

Takže počítače nikdy "neuvidí" 1/10: vždy bude používat zlomek 7205759403792794 / 72057594037927936:

```
>>> .1 * 2L**56
7205759403792794.0
```

Vynásobíme-li tuto hodnotu číslem 10**30, uvidíme hodnotu jejích 30 nejvýznamnějších dekadických číslic:

```
>>> 7205759403792794L * 10L**30 / 2L**56
100000000000000005551115123125L
```

Číslo uložené v počítači je tedy přibližně rovno dekadickému číslu 0.100000000000000005551115123125. Zaokrouhlením na 17 platných číslic získáme právě onu hodnotu 0.10000000000000001, kteroužto Python zobrazuje namísto hodnoty 0.1!

Licence

C.1 Historie Pythonu

Vývoj Python začal v roce 1990 na Stichtig Mathematisch Centrum (CWI, viz. http://www.cwi.nl/) v Nizozemí. U jeho zrodu stál Guido vam Rossum. Hlavní ideu Pythonu převzal z jazyku, který doposud vyvíjel, z jazyka ABC. Ačkoli Guido zůstal hlavním programátorem a ideologem Pythonu, použil v jeho interpretru mnoho kódu jiných programátorů.

V roce 1995 Guido pokračoval v práci na Pythonu na půdě Corporation for National Research Initiatives (CNRI, viz. http://www.cnri.reston.va.us/) v Restonu ve Virginii kde vydal několik verzí programu.

V květnu roku 200 založil Guido a další jemu blízcí vývojáři Pythonu tým pod hlavičkou společnosti BeOpen.com — tým BeOpen PythonLabs. V říjnu téhož roku se PythonLabs přesunuly pod firmu Zope Corporation (posléze Digital Creations, viz http://www.zope.com/). V roce 2001 byla založena nezisková organizace Python Software Foundation (PSF, viz http://ww.python.org/psf/), která vlastní všechno "intelektuální jmění" kolem jazyka Python. Společnost Digital Creations je i nadále jedním ze sponzorů PSF.

Všechny verze Pythonu mají otevřený kód (viz http://ww.opensource.org/). Většina, ale ne všechny verze Pythonu jsou kompatibilní s licencí GPL. Následující tabulka proto shrnuje informace o všech doposud vydaných verzích interpretru:

Verze	Odvozena z	Rok	Vlastník	GPL kompatibilní?
0.9.0 až 1.2	n/a	1991-1995	CWI	ano
1.3 až 1.5.2	1.2	1995-1999	CNRI	ano
1.6	1.5.2	2000	CNRI	ne
2.0	1.6	2000	BeOpen.com	ne
1.6.1	1.6	2001	CNRI	ne
2.1	2.0+1.6.1	2001	PSF	ne
2.0.1	2.0+1.6.1	2001	PSF	ano
2.1.1	2.1+2.0.1	2001	PSF	ano
2.2	2.1.1	2001	PSF	ano

Poznámka: Slůvko GPL kompatibilní neznamená, že Python je distribuován pod licencí GPL. Všechny licence Pythonu narozdíl od GPL umožňují distribuovat modifikované verze jazyka aniž byste byli nuceni uvolnit váš zdrojový kód. GPL kompatibilní lecence umožňují kombinovat na jednom mediu Python společně s dalším software uvolněným pod GPL. Ostatní licence toto *neumožňují*!

C.2 Poděkování ...

Na závěr této knihy bych chtěl poděkovat všem dobrovolníkům, kteří určitým způsobem přispěli k práci na jazyce Python. Rovněž bych chtěl poděkovat všem, kteří mi pomáhali s překladem. Dále lidem z Českého sdružení programá-

torů a uživatelů jazyka Python - PyCZ. Můj obdiv patří i všem lidem, kteří uvolnili zdrojové kódy svých programů ve smyslu hnutí OpenSource.

Mé speciální poděkování patří i Kačence Š.

C.3 Licence