Létající cirkus

Python tutoriál

Jan Švec

14. února 2003

Obsah

Kapitola 1

Úvod

Téměř každý programátor vyzkoušel za svůj život vícero programovacích jazyků. A také téměř každý si vysnil svůj ideální. Jednoduchý, robustní, hezký, přenositelný. A poněvadž se v poslední době čím dál více mluví a píše o jazyku, který tyto představy více než naplňuje, podíváme se na něj v našem seriálu. Dámy a pánové, přichází pan Python...

Jazyk Python začal vznikat v roce 1989 ve výzkumném ústavu v Amsterodamu. Při jeho zrodu stál Guido van Rossum a je vidět, že u návrhu dostatečně přemýšlel. Vznikl promyšlený jazyk, který je stále ve vývoji. Jméno dostal podle pořadu BBC *Monty Python's Flying Circus*. V současné době běží na mnoha platformách (Linux, Win, Mac, WinCE, OS/2, Java). Stejně tak programy v něm napsané lze na těchto systémech téměř vždy spouštět bez úprav.

A jaký Python vlastně je? Čistý objektový jazyk se správou výjimek, kompilací do bytecodu, mnoha vysokoúrovňovými typy (řetězce, seznam, asociativní pole), plně podporující Unicode, lze jej doplnit o vlastní vestavěné typy a funkce pomocí C/C++, nebo naopak lze interpret začlenit do programu v jiném jazyce. Základní balík obsahuje velké množství modulů, které lze ihned používat ve vašem programu. Jmenujme moduly pro přístup k databázím, GUI, službám operačního systému, HTTP, FTP, POP3, SMTP a mnohým jiným protokolům. Samozřejmostí jsou regulární výrazy. Definuje také několik modulů pro přístup k vnitřním mechanismům Pythonu (garbage collector, parser, kompiler). V Pythonu je taktéž napsán debugger a profiler tohoto jazyka. Slibuji, že v některém z příštích dílů se na ně také podíváme.

Domovskou stránkou Pythonu je `www.python.org`; najdete zde velké množství dokumentace, zdrojové tarbally i předkompilované balíky pro různé systémy. Aktuální verze je 2.2 (přesněji řečeno existuje momentálně release candidate pro verzi 2.2, vydání finální verze je očekáváno každým dnem – pozn. redakce), která obsahuje množství změn oproti starším verzím, zmiňme jen možnost dědit od vestavěných typů. Pro pochopení základů Pythonu budou stačit i starší verze (1.5.2 nebo 2.0), které jsou ponejvíce obsaženy v nejnovějších distribucích.

Nyní, pokud již máme všechny balíčky nainstalované, si poprvé spustíme interaktivní interpret Pythonu. Na příkazové řádce stačí zadat příkaz `python` (za předpokladu, že ho máte v cestě) a okamžitě se nám objeví výzva příkazového řádku ">>>". Python zná ještě sekundární výzvu "...", kterou používá při zadávání složených konstrukcí. Python ihned vyhodnocuje zadávané výrazy, pouze v případě, že jde o složenou konstrukci, vyčká až na její ukončení. Interpret ukončíme kombinací `Ctrl-D` nebo zadáme:

```
import sys
sys.exit()
```

Komentáře začínají vždy znakem `#`. Stejně jako v jiných interpretovaných jazycích lze na začátek souboru s kódem programu uvést magickou sekvenci `#!/usr/bin/env python` a v případě, že soubor má nastaven executable bit a ve vaší cestě se nachází příkaz `python`, spustí se tento soubor stejně jako jakýkoli skript. Zavedenou konvencí jsou zdrojové soubory Pythonu s příponou ".py". Po spuštění zdrojového kódu ze souboru Python transparentně zkompiluje celý soubor na bytecode, který uloží do souboru s příponou ".pyc". Ten je použit pro urychlení příštího startu programu, odpadá parsování a kompilace. Jestliže soubor ".pyc" svým obsahem neodpovídá souboru ".py", je ignorován a vytvořen znovu. Soubory ".pyc" je možné spouštět samostatně, tj. je možné vyvíjet i closed-source programy :(Používají se ještě soubory s příponou ".pyo", ve kterých je uložen optimalizovaný bytecode. V současných verzích se ale žádných velkých optimalizací nedočkáte, jsou pouze vynechány některé ladící informace.

Čísla jsou chléb programátora. Python rozlišuje mezi čísly celými (integer a long integer), reálnými a imaginárními:

```
num1 = 123                          # (1)
num2 = 123456789L                   # (2)
num3 = 987.654                      # (3)
num4 = 321j                         # (4)
num5 = 1 + 3.1415j                  # (5)
num6 = num7 = 0                     # (6)
num1 += 10                          # (7)
```

Řádek (1) ukazuje použití klasického 32bitového integeru. Skvělou věcí je dlouhý integer (long integer) na řádku (2), který se od klasického liší připojením písmena l (nebo L) za číslo. Jeho velikost je omezena pouze velikostí dostupné operační paměti. Zápis reálných čísel (3) je stejný jako v jazyce C. Imaginární číslo (4) rozlišuje písmeno `j` připojené za hodnotu čísla, komplexní číslo dostaneme sečtením reálného a imaginárního čísla podobně jako na řádku (5). Reálnou i imaginární složku komplexního čísla získáme zápisem `num5.real` a `num5.imag`. Jak reálná, tak imaginární složka je reálného typu. Absolutní velikost komplexního čísla vrátí funkce `abs(num5)`.

Nabídka operátorů, jejich priorita a zápis jsou stejné jako v jazyce C. Chybí ale ternární operátor a operátory inkrementace a dekrementace. Operátor přiřazení je stejný jako v céčku a lze použít i vícenásobné přiřazení (6). Stejně tak se může používat spojení operátoru a přiřazení (7). Informace o operátorech, jejich prioritě apod. je možné najít v dokumentaci dodávané s Pythonem.

Při vyhodnocování aritmetických výrazů se hodnoty nejprve převedou na společný typ podle následujících pravidel:

1. Je-li některý z argumentů komplexní číslo, ostatní argumenty jsou zkonvertovány na komplexní čísla.

2. Je-li některý z argumentů reálné číslo, ostatní argumenty jsou zkonvertovány na reálná čísla.

3. Je-li některý z argumentů dlouhý integer, ostatní argumenty jsou zkonvertovány na dlouhé integery.

4. Jinak jsou všechny argumenty obyčejné integery a žádná konverze není třeba. Výsledek je stejného typu jako společný typ.

```
>>> 2 + 3.0                    # (1)
5.0
>>> 3 / 2                      # (2)
1
>>> 3.0 / 2                    # (3)
1.5
>>> y = 3 * 2                  # (4)
>>> _                          # (5)
1.5
```

Proč interpret zobrazil po zadání výrazu jeho výsledek? Není-li hodnota nějakého výrazu přiřazena proměnné, *interaktivní* interpret ji uloží do speciální proměnné _. Výsledek řádku (4) nebyl zobrazen, protože byl přiřazen proměnné `y`. Ze stejného důvodu zobrazí Python na řádku (5) hodnotu 1.5 místo 6. Pozor: proměnná _ se používá pouze v interaktivním procesoru. Spouští-li se skript ze souboru, není definována a její použití vede k chybě.

Všechny proměnné a atributy objektů v Pythonu jsou netypové, což znamená, že proměnným můžeme přiřadit různé typy. Ke zjištění typu proměnné je vyhrazena funkce `type()`, vracející typ argumentu, který jí byl předán. Proměnné se nedeklarují, k jejich vytvoření dojde tehdy, když je jim přiřazena hodnota. Použití nedefinované proměnné vede k chybě.

Řetězce v Pythonu je možné zapsat několika způsoby, přičemž lze použít jak jednoduché, tak dvojité uvozovky pro uvození řetězce. Nelze je ovšem kombinovat (viz řádek (8)):

```
>>> str1 = 'zluta'                          # (1)
>>> str2 = "cervena"                        # (2)
>>> str3 = '''modra                         # (3)
... a zelena'''                             # (4)
>>> str4 = 'znak noveho radku: \n'          # (5)
>>> str5 = r'toto jsou dva znaky: \n'       # (6)
>>> str6 = u'Unicodovy\u0020retezec'        # (7)
>>> str7 = 'chybny zapis"                   # (8)
```

Řádky (1) a (2) jsou podobné jako v jiných jazycích. Na řádku (3) je vidět víceřádkový řetězec. Všimněte si, že jsou do něho zahrnut i znaky "# (3)". Víceřádkový řetězec začíná `'''` (nebo `"""`) a pokračuje, dokud není ukončen dalšími třemi uvozovkami. Proto také interpret zobrazil sekundární prompt na řádku (4). V Pythonu lze podobně jako v C užívat speciální znaky v podobě \n, \t atd. (5). Pro vložení bytu lze použít sekvence \000, kde místo 000 je index znaku v ASCII tabulce v osmičkové soustavě a podobně zápis v šestnáctkové soustavě vypadá takto: \x00. Výjimkou jsou raw řetězce uvozené písmenem `r` (nebo `R`) (řádek (6)), v nichž se ruší význam escape znaků a řetězec je uložen stejně, jako byl zapsán. Na řádku (7) je vidět příklad unicodových řetězců. Ty jsou uvozeny písmenem `u` (nebo také `U`). Protože unicode používá 16bitové kódování, je tu pro zadání 16bitového znaku sekvence \0000. Unicodové řetězce také mohou být zapsány jako raw. Podobně víceřádkové řetězce můžou být také raw a unicode. V Pythonu neexistuje zvláštní typ char, jeden znak je prostě řetězec o délce 1. Délku řetězce (počet znaků) zjistíme pomocí funkce `len()`, které jako argument předáme řetězec.

Řetězce v Pythonu lze spojovat použitím operátoru `+`. Jsou-li dva řetězce umístěny hned za sebou, dojde k jejich spojení. Podobně jako v shellu, i zde je možné rozdělit jeden dlouhý řádek za použití zpětného lomítka:

```
>>> 'Linux: ' + 'try' ' or ' 'die'
'Linux: try or die'
>>> 'Dobry ' \
... 'den'
'Dobry den'
```

Nováčky v tomto jazyce si ihned získá možnost vynásobit řetězec celým číslem n. Výsledkem je původní řetězec nakopírovaný n-krát za sebe. Trošku předběhnu, celým číslem lze vynásobit jakýkoli sekvenční typ (tuple, seznam), více ale v dalším dílu našeho seriálu.

```
>>> 'trikrat ' * 3
'trikrat trikrat trikrat '
```

Další velice používanou konstrukcí je `print`. Ta postupně vyhodnotí všechny výrazy, které jsou za ní zapsány, oddělené čárkou, poté je převede na řetězce a nakonec vypíše na standardní výstup. Je-li za posledním výrazem umístěna čárka, nedojde k vložení znaku nového řádku.

```
>>> print 'soucet 2 + 3 je:', 2 + 3
'soucet 2 + 3 je: 5'
```

Jak vlastně dojde k převedení výrazu na řetězec? V Pythonu je pro tyto účely určena funkce `str()`, která vrátí svůj argument převedený na řetězec. Další možností je uzavřít výraz mezi obrácené apostrofy:

```
>>> print 'toto je retezec: Ahoj a toto je cislo: ' +
  `2 + 5` toto je retezec: Ahoj a toto je cislo: 7
```

Pozor: Použití `str()` a ``` `` ``` není ekvivalentní. K rozdílu dojde, je-li argument řetězec. Zatímco `str()` ho vrátí beze změny, ``` `` ``` ho „obloží“ uvozovkami.

V další kapitole se podíváme na složitější typy: tuple, seznam, asociativní pole, vysvětlíme si slice konstrukce a možná si povíme něco o definování funkcí.

Kapitola 2

Sekvenční typy

Mezi tyto typy patří typ tuple a seznam. Jde o konečnou množinu prvků, které jsou indexovány od 0 do n-1, kde n je počet prvků sekvence. Dělí se dále na immutable (neměnné) sekvence a mutable (proměnné) sekvence. U neměnných sekvencí nelze po vytvoření objekt tohoto typu již změnit, lze pouze vytvořit nový z již existujících objektů, kdežto proměnné sekvence lze měnit i po jejich vytvoření.

2.1 Tuple

Je prvním ze sekvenčních typů. Patří mezi tzv. *immutable sekvence*, po jeho vytvoření nelze jeho prvky již změnit.

```
>>> t1 = (1, 2, 3, 4)                    # (1)
>>> t2 = 1, 2, 3, 4                      # (2)
>>> t3 = (1, 'jedna', 'one')             # (3)
>>> t4 = ()                              # (4)
>>> t5 = (2, )                           # (5)
>>> tuples = (t1, t2, t3)                # (6)
```

Na předchozích příkladech vidíme několik ukázek zápisu tuple. Jednotlivé prvky tuple se zapisují oddělené čárkou (1), přičemž uzávorkování není povinné (2). Všechny složené typy samozřejmě mohou obsahovat prvky různých typů (3). Prázdný tuple získáme zapsáním dvou kulatých závorek (4) – toto je také samozřejmě jediný případ, kdy není možné vynechat závorky kolem prvků. Na řádku (5) je tuple o jediném prvku. Nesmíme ale zapomenout, že za tímto prvkem je čárka! Interpretr takto rozlišuje mezi tuple a obyčejným výrazem uvedeným v závorce. Samozřejmostí je, že do jednoho tuple může být vložen jiný (6).

2.2 Seznam (list)

Je obdobou typu tuple. Jediným rozdílem je, že seznam je sekvence proměnná, tj. jednotlivé prvky seznamu můžeme měnit.

```
>>> s1 = [1, 2, 3, 4]                    # (1)
>>> s2 = []                              # (2)
>>> s3 = [2]                             # (3)
```

Seznam se zapisuje podobně jako tuple (1), pouze s tím rozdílem, že namísto kulatých závorek píšeme závorky hranaté a nelze je vynechat (při jejich vynechání dostaneme typ tuple, což nechceme). Seznam o jednom prvku dostaneme zápisem hodnoty mezi dvě hranaté závorky, přičemž nemusíme uvádět čárku za prvkem (3). Prázdný seznam získáme podobně jako prázdný tuple zápisem `[]` (2).

Každý objekt typu seznam má i několik metod (`append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, `sort`). Pro bližší informace vás odkážu do dokumentace Pythonu, zde uvedu jen malou ukázku použití:

```
>>> jmena = ['Zbysek', 'Ctirad', 'Emil', 'Adolf']
>>> jmena.insert(0, 'Jenda')
>>> jmena.extend(['Blaza', 'Katka'])
>>> jmena
['Zbysek', 'Jenda', 'Ctirad', 'Emil', 'Adolf', 'Blaza', 'Katka']
>>> jmena.sort()
['Adolf', 'Blaza', 'Ctirad', 'Emil', 'Jenda', 'Katka', 'Zbysek']
```

2.3 Přístup k prvkům sekvenčních objektů

```
>>> seznam = [1, 2, 3, 4, 5]                    # (1)
>>> seznam[0]                                   # (2)
1
>>> seznam[3] = 5                               # (3)
>>> seznam                                      # (4)
[1, 2, 3, 5, 5]
>>> seznam[0] = seznam                          # (5)
>>> seznam                                      # (6)
[[...], 2, 3, 5, 5]
>>> seznam[0][1]                                # (7)
2
```

K prvkům objektů typu tuple i seznam se přistupuje stejně, pomocí indexů zapsaných do hranatých závorek za proměnnou (2). U seznamu můžeme přiřadit prvku na určitém indexu novou hodnotu podobně jako na řádku (3). Prvkem seznamu může být opět seznam. Dokonce lze vložit seznam i do sebe samého, Python pak zobrazí místo nekonečné posloupnosti znaky '...'. Seznam ale pořád pracuje tak, jak má, a proto výraz na řádku (7) má hodnotu 2 (nultý prvek ze seznamu 'seznam' je opět tentýž seznam, z něho prvním prvkem je dvojka). Je-li index mimo platný rozsah, dojde k výjimce.

```
>>> seznam = [1, 2, 3, 4, 5]                    # (1)
>>> seznam[-1]                                  # (2)
5
>>> seznam[-5]                                  # (3)
1
>>> seznam[-6]
```

Zajímavé ale je to, co se stane při použití záporného indexu. Nedojde k výjimce, ale prvky se začnou indexovat „odzadu“, takže poslední prvek má index -1 atd. Překročíme-li rozsah pole, opět dojde k výjimce (řádek (4)).

2.4 Slice konstrukce

Další velice užitečnou konstrukcí je *slice*, která umožňuje jakýkoli sekvenční typ doslova „rozřezat“ na kousky. Slice podobně jako indexy používá hranaté závorky, v nichž jsou ale uvedena dvě celá čísla oddělená dvojtečkami.

```
>>> seznam = (1, 2, 3, 4, 5)                    # (1)
>>> seznam[3:5]                                  # (2)
(4, 5)
>>> seznam[0:4]                                  # (3)
(1, 2, 3, 4)
>>> seznam[0:50]                                 # (4)
(1, 2, 3, 4, 5)
>>> seznam[:3]                                   # (5)
(1, 2, 3)
>>> seznam[1:]                                   # (6)
(2, 3, 4, 5)
>>> seznam[:]                                    # (7)
(1, 2, 3, 4, 5)
>>> seznam[-1:]                                  # (8)
(5,)
```

První z čísel znamená dolní hranici pro výběr prvků, druhé je horní hranice, přičemž není chybou, leží-li tato čísla mimo rozsah platných indexů (4). Je-li první číslo vynecháno, nahradí se číslem 0, je-li vynecháno druhé, dosadí se za něj hodnota proměnné `sys.maxint` (defacto maximální velikost integeru) (5)(6)(7). I zde je možné používat záporná čísla podobně jako u indexů (8). Nejsou-li v rozsahu žádné prvky, vrátí slice konstrukce prázdnou sekvenci. Zde je třeba zmínit, že slice vždy vrací novou sekvenci toho typu, na nějž se aplikuje. I když se slice konstrukcí vybere jediný prvek, vždy se jedná o sekvenci! Podobně při použití slice s vynechaným dolním i horním indexem dojde k vytvoření *nové* sekvence!

2.5 Řetězce

V předchozím dílu jsem vám záměrně zatajil jednu důležitou vlastnost řetězců, ty totiž patří také mezi sekvenční immutable typy, z čehož vyplývá, že na ně lze aplikovat stejné funkce a konstrukce jako na tuple a seznamy. Totéž platí samozřejmě i pro unicodové řetězce.

```
>>> str = 'linux: try or die'
>>> str[:5]
'linux'
>>> len(str)
17
>>> str[5]
```

```
':'
```

Sekvence stejného typu lze spojovat pomocí operátoru `+`. Vynásobíme-li sekvenci celým číslem `n`, získáme sekvenci, v níž se původní sekvence opakuje `n`-krát.

```
>>> x = (1, 2)
>>> y = (3, 4)
>>> (x + y) * 3
(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
```

Objekt typu tuple lze zkonvertovat na seznam a naopak. Děje se tak pomocí vestavěných funkcí `list()` a `tuple()`. Podobně i řetězec lze zkonvertovat na tuple nebo seznam.

```
>>> t = (1, 2, 3, 4)
>>> list(t)
[1, 2, 3, 4]
>>> tuple(_)
(1, 2, 3, 4)
>>> list('linux')
['l', 'i', 'n', 'u', 'x']
```

Kapitola 3

Mapované typy

Další skupinou typů jsou mapované typy. Jde o konečnou množinu prvků indexovaných libovolnou množinou indexů. Mezi mapované typy patří asociativní pole a jako mapovaný typ je implementován např. i přístup k databázím dbm apod.

3.1 Asociativní pole (dictionaries)

Jde o pole, kde se pro indexování prvků používá libovolných objektů. Jediné omezení na tyto objekty (klíče) je, aby z nich bylo možno určit jejich hash. To lze ale pouze pro objekty neměnné (immutable sekvence, řetězce, čísla a další, viz dokumentace k Pythonu).

```
>>> znamky = {'Voprsalek': 3, 'Macha': 5,      # (1)
... 'Novak': 1}                                # (2)
>>> znamky['Voprsalek']                        # (3)
3
>>> znamky['Macha'] = 4                        # (4)
>>> znamky['Svec'] = 2                         # (5)
>>> znamky                                     # (6)
{'Svec': 2, 'Macha': 4, 'Voprsalek': 3, 'Novak': 1}
```

Na řádcích (1)(2) jsme vytvořili nové asociativní pole. To se zapisuje jako páry `klíč: hodnota` oddělené čárkou mezi složenými závorkami. Pro přístup k hodnotám se používá klíč zapsaný ve hranatých závorkách za proměnnou. Přistupujeme-li k indexu, který není v asociativním poli, dojde opět k výjimce. Vytvoření nového páru `klíč: hodnota` pomocí přiřazení je samozřejmostí (5).

Také objekt typu asociativní pole má několik metod (`clear`, `copy`, `get`, `has_key`, `items`, `keys`, `setdefault`, `update`, `values`). Zde vás opět odkážu na dokumentaci.

Kapitola 4

Konstrukce pro řízení toku programu

Žádný programovací jazyk se bez nich neobejde. A ani Python není výjimkou. Tento programovací jazyk používá pouhé tři konstrukce, kterými lze ovlivnit, co se bude v programu dít. Jsou to `if`, `for` a `while`.

4.1 Konstrukce `if`

Je základní konstrukce pro řízení toku programu. Předpokládám, že tento článek čtou lidé, kteří již mají nějaké zkušenosti s programováním, a proto pouze popíši, jakým způsobem se používá.

```
>> D = 2                                    # (1)
>>> if D > 0:                               # (2)
...     print 'rovnice ma 2 reseni'         # (3)
... elif D == 0:                            # (4)
...     print 'rovnice ma 1 reseni'         # (5)
... else:                                   # (6)
...     print 'rovnice nema reseni'          # (7)
...                                         # (8)
rovnice ma 2 reseni
```

Toto je klasická školní ukázka, jak určit počet kořenů kvadratické rovnice v oboru reálných čísel. Všimněte si, že v Pythonu se nepoužívá žádný druh uvození bloků programu jako třeba v C/C++ nebo Pascalu. Jednotlivé bloky jsou určeny odsazením od kraje. Nezáleží na tom, kolik odsadíte a jestli použijete mezery nebo tabulátory, rozhodující je, aby odsazení bylo v celém bloku stejné. Pozor při kopírování řádků kódu např. z webového browseru do editoru. V editoru vypadá vše, jak má, kód funguje, ale doplníte-li nějaký vlastní kód, interpretr si stěžuje na chybu v syntaxi. Háček je v tom, že při kopírování se text odsadil jen mezerami, a když jste doplnili text v editoru a odsadili jste ho tabulátorem, začal si Python stěžovat. Odsazení od kraje je geniální věc, nedochází k žádným nejednoznačným konstrukcím, kdy se např. neví, k jakému `if` daný `else` patří. A tento způsob formátování také rozhodně prospěje vzhledu celého zdrojového kódu programu. V Pythonu platí jedna zásada: končí-li jeden řádek

na dvojtečku, následující musí být odsazen. Chceme-li, aby bylo tělo bloku prázdné, nelze ho vynechat, musíme místo bloku uvést konstrukci `pass`, jde o prázdnou konstrukci, nedělá vůbec žádnou akci, je zde pouze pro uspokojení parseru. U konstrukcí if nemá praktický význam, hodí se ale u konstrukcí `try ... except`, když chceme, aby byla výjimka odchycena, ale neprovedla se žádná akce. Doplňme, že stejně jako bloky v konstrukci `if` se odsazují i bloky v cyklech `for` a `while`, při definicích funkcí a tříd, u konstrukce `try ... except` apod.

4.2 Konstrukce `for`

Tato konstrukce se liší od svého jmenovce v jazyce C. V Pythonu cyklus `for` prochází objekt sekvenčního typu a jednotlivé prvky dosazuje do řídící proměnné. Stejně jako v jiných jazycích, i zde můžeme použít konstrukce `break` a `continue`. Zvláštností Pythonu je, že i cyklus `for` může mít část `else`, která se vykoná po zpracování všech prvků sekvence. Je-li ovšem cyklus přerušen pomocí `break` nebo v jeho tělo dojde k výjimce, část `else` se neprovede. Chceme-li, aby cyklus `for` iteroval přes množinu celých čísel (podobně jako třeba v Pascalu), musíme tuto množinu nejprve vytvořit jako sekvenci pomocí vestavěné funkce `range()`.

```
>>> soucet = 0
>>> for cislo in range(50):
...     soucet += cislo
... else:
...     print 'soucet cisel od 0 do 49 je', soucet
...
soucet cisel od 0 do 49 je 1225
```

Funkce `range()` vrací seznam, naplněný čísly. Je-li jí předáno jako argument jedno číslo n, obsahuje seznam čísla 0 až n-1. Jsou-li jí předána čísla dvě, řekněme m a n, seznam bude naplněn čísly m až n-1. Jsou-li jí předána tři čísla x, y a z, vrátí čísla od x až po y-1 s krokem z. Seznam je v paměti alokován najednou, proto se u velkých rozsahů, kde by bylo zbytečné uchovávat jednotlivé prvky v paměti, používá funkce `xrange()` se stejným rozhraním.

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

Je-li jako sekvence použita jakákoli proměnná sekvence, je třeba dát pozor, zda se s jednotlivými prvky a s jejich pořadím v těle cyklu nemanipuluje, pak by totiž jeden prvek mohl být zpracován dvakrát, zatímco jiný by byl vynechán. V tomto případě je vhodné vytvořit dočasnou kopii seznamu pomocí slice konstrukce.

```
>>> seznam = [1, 2, 3]
>>> for item in seznam:
...     print 'pridavam', item
...     seznam.append(item)
```

```
... else:
...     print 'seznam byl zdvojen'
...
```

Předchozí příklad vede k nekonečnému cyklu. Nápravou je vytvoření dočasné kopie seznamu pomocí slice. Nyní vše již pracuje tak, jak má:

```
>>> seznam = [1, 2, 3]
>>> for item in seznam[:]:
...     print 'pridavam', item
...     seznam.append(item)
... else:
...     print 'seznam byl zdvojen'
...
```

4.3 Konstrukce `while`

Je obdoba cyklu `while` v C. Její tělo je vykonáváno, dokud je výraz – podmínka vyhodnocen jako `true`. I zde je možné použít nepovinnou část `else`, která se vykoná při ukončení cyklu. Podobně jako u cyklu `for` lze použít konstrukce `break` a `continue` pro ukončení cyklu, respektive pro pokračování další iterací. Při ukončení cyklu pomocí `break` se taká samozřejmě neprovede část `else`.

```
>>> cislo = 0
>>> soucet = 0
>>> while cislo < 50:
...     soucet += cislo
...     cislo += 1
... else:
...     print 'soucet cisel od 0 do 49 je', soucet
...
soucet cisel od 0 do 49 je 1225
```

Kapitola 5

Vstup od uživatele

Abyste mohli napsat jednoduchý interaktivní program, potřebujete ještě vědět, jakým způsobem se může program zeptat uživatele na data. Úplně nejzákladnější funkcí je funkce `raw_input()`, která přejímá jeden nepovinný argument. Ta se zeptá uživatele na data, která pak vrátí jako řetězec. Přitom použije argument jako výzvu. Existuje ještě funkce `input()`, která zadaný řetězec nejprve vyhodnotí jako výraz jazyka Python a až poté vrátí hodnotu tohoto příkazu. Bližší informace najdete v dokumentaci.

Kapitola 6

Logické výrazy

Každý výraz jazyka Python může být zároveň i logickým výrazem. Podobně jako v jazyce C zde neexistuje žádný speciální typ určený pro logické proměnné. I tady se místo něho může používat obyčejný integer (nulová hodnota je `false`, cokoli jiného `true`). Ale i výrazy jiných typů mohou být vyhodnoceny jako logický výraz. Například prázdná sekvence má hodnotu `false`, hodnota `None` je vyhodnocena jako `false`, hodnota `Ellipsis` jako `true` atd. Blíže se o tomto problému zmiňuje dokumentace k jazyku.

V Pythonu, stejně jako v céčku a jiných jazycích, můžeme nalézt tyto relační operátory: $<$, $>$, $==$, $!=$, $>=$, $<=$. Mimo nich existuje ještě $<>$, ale samotní autoři doporučují místo něj raději používat ekvivalentní $!=$. Každé porovnání vrací pouze hodnoty 1 (`true`) nebo 0 (`false`). Python umožňuje zřetězení operátorů, zápis $a < b < c$ je tady ekvivalentní zápisu $a < b \ and \ b < c$. V prvním případě se ale výraz `b` vyhodnotí pouze jednou, kdežto ve druhém dvakrát. Lze zapsat i $a < b > c$, pak tento matematický „patvar“ vede k zápisu $a < b \ and \ b > c$.

Python umožňuje porovnávat nejen čísla, ale i řetězce (i unicodové) a obecně jakékoli jiné sekvence. Porovnávání sekvencí se děje lexikograficky (tj. nejprve nultý prvek s nultým, poté první s prvním atd., dokud není nalezen první rozdíl nebo jedna ze sekvencí neskončí). Lahůdkou je možnost porovnávaní asociativních polí. Interpret si nejprve převede asociativní pole na sekvenci z prvků (klíč, hodnota), kterou si setřídí, a následně obě sekvence porovná. Jazyk dokonce umožňuje i porovnávání různých typů. Výsledek takového porovnání záleží čistě na vůli interpretu, ale je *vždy* stejný, takže třeba číslo je vždy menší než seznam, ten je zase menší než řetězec atd. Čísla různých typů se nejprve převedou na společný typ a pak se teprve porovnají. Pro bližší informace viz. dokumentace k jazyku.

```
>>> (1, 2, 3) < (1, 2, 4)
1
>>> [1, 2, 3] < [1, 2, 4]
1
>>> 'Brezen' < 'Cerven' < 'Cervenec'
1
>>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
1
>>> 1 == 1.0
1
```

Logické operátory, jak je známe z jazyka C, bychom hledali marně. Python pro logický součin, součet a negaci používá místo symbolů &&, || a ! slova `and`, `or` a `not`. Důvod je prostý: opomenutím jednoho „andu“ jsme dostali bitový součin, a ne logický. Ze stejných důvodů Python také nedovoluje použití přiřazení ve výrazu. Nejednoho začátečníka v C zaskočila záměna = a == například v podmínce `if`. Programátor v Pythonu si prostě musí zvyknout na to, že funkčnost někdy musela ustoupit estetice a přehlednosti.

Operátory `and` a `or` pracují odlišně od jiných jazyků. Zde výsledek nemusí být jen 1 nebo 0, ale může mít obecně jakoukoli hodnotu a typ:

- Při vyhodnocování výrazu `x and y` se nejprve vyhodnotí výraz `x`, je-li `false`, je vrácena jeho hodnota, jinak se vyhodnotí výraz `y` a vrátí se výsledek výrazy `y`:

```
>>> 1 and 4
        4
>>> 0 and 4
        0
```

- Při vyhodnocování výrazu `x or y` se nejprve vyhodnotí výraz `x`, je-li `true`, je vrácena jeho hodnota, jinak se vyhodnotí výraz `y` a vrátí se výsledek výrazu `y`:

```
>>> 1 or 4
        1
>>> 0 or 4
        4
```

Na první pohled chování stejné jako v jiných jazycích, ale... Python totiž dokáže vyhodnotit libovolný výraz jako logický (viz. začátek dnešního dílu) a výše zmíněné hodnoty se přímo nabízejí i k jinému způsobu použití, než jen k vytváření logických výrazů, třeba k dosazení implicitní hodnoty:

```
>>> answer = raw_input('Zadejte text: ')                    (1)
                                                            (2)
>>> print 'Zadali jste ' + (answer or 'prazdny retezec')    (3)
Zadali jste prazdny retezec
```

V Pythonu můžeme, stejně jako v jiných jazycích, ovlivňovat prioritu operátorů pomocí závorek. To jsme provedli i na řádku (3), jelikož operátor `+` má vyšší prioritu než logický operátor `or`. O prioritě operátorů se více zmiňuje dokumentace jazyka.

Další operátory, které nemají v céčku obdobu, jsou `in` a `not in`. Ty mají dva operandy `m`, `n`. Operátor `in` testuje, je-li `m` prvkem sekvence `n`, `not in` je jeho pravým opakem.

```
>>> 'l' in 'linux'
1
>>> 1 in (2, 3, 4, 5)
0
```

Na konec povídání o operátorech se ještě zmiňme o operátoru `is`. Ten testuje, jsou-li jeho dva operandy jedním a týmž objektem. Vytváříte-li nový objekt proměnného typu, je vždy v paměti vytvořen objekt nový, ne tak při vytváření objektu neměnného typu. Zde již interpret *může* (ale *nemusí*) použít nějakou předchozí instanci v paměti. Je jen na jeho libovůli, učiní-li tak, nebo ne.

```
>>> a = []
>>> b = []
>>> a is b
0
>>> a = ()
>>> b = ()
>>> a is b
1             # může být i 0, záleží na interpretru
```

Kapitola 7

Proměnné a prostory jmen

Proměnné, jak je známe z jiných kompilovaných jazyků, v Pythonu nenajdeme. Zde je každá hodnota, funkce, třída, metoda, instance třídy nebo modul objektem v paměti interpretru a proměnná je jen odkazem na tento objekt. Správa paměti je obsluhovaná automatickým garbage collectorem, založeným na počítání odkazů. Díky tomu se v Pythonu vůbec nemusíme starat o přidělenou paměť. Každý objekt má počítadlo odkazů, které je při vytváření objektu vynulováno. Při vzniku odkazu na tento objekt (tedy nejen přiřadíme-li objekt nějaké proměnné, ale vložíme-li ho třeba do seznamu apod.) je hodnota počítadla objektu zvýšena o 1, při zrušení odkazu je hodnota snížena o 1, a až pokud je hodnota tohoto počítadla rovna nule, dojde k fyzickému odstranění objektu z paměti (u instancí tříd také k zavolání destruktoru). V Pythonu nenajdeme žádné deklarace, novou proměnnou prostě vytvoříme, přiřadíme-li jí nějakou hodnotu. Použití neexistující proměnné ale vede k výjimce, a proto nedochází ke zmatkům jako např. v BASICu, kde jsme proměnné "pocet" přiřadili hodnotu a dále jsme třeba počítali s neexistující proměnnou "poce".

Mapování jmen proměnných na objekty mají na starosti prostory jmen. Ty jsou implementovány jako asociativní pole, jejichž klíčem je jméno proměnné a hodnotou je vlastní objekt, na který proměnná odkazuje. Python používá tři prostory jmen (namespace) – globální, lokální a builtin (česky snad „vestavěný“, ale já se budu držet anglického originálu).

- *Globální* – zde jsou uloženy globální proměnné. Program má k těmto proměnným přístup jen pro čtení. Potřebujeme-li přiřadit globální proměnné nějakou hodnotu, musíme proměnnou deklarovat jako globální konstrukcí `global` (viz. následující odstavce). Neučiníme-li tak, dojde k vytvoření nové proměnné v lokálním prostoru jmen a hodnota globální proměnné změněna nebude!

- *Lokální* – zde jsou uloženy lokální proměnné. Všechny nově vytvářené proměnné jsou umístěny v tomto prostoru jmen (nejsou-li však deklarovány jako globální konstrukcí `global`).

- *Builtin* – všechny proměnné vestavěné v interpretru. Zaveden proto, aby se program na tyto proměnné mohl odkazovat přímo bez použití tečkové notace (blíže viz kapitola 9 na straně 33, která bude věnována modulům a balíčkům)

Prostory jmen jsou prohledávány v pořadí lokální, globální, builtin. Význam lokálních a globálních prostorů jmen se mění v závislosti na aktuálním kontextu, tj. na místě, kde se zrovna program nachází. Např. v těle modulu je globální a lokální prostor jmen totožný, v těle

funkce je globální prostor jmen stejný jako globální prostor jmen nadřízeného bloku, lokální prostor jmen se vytvoří vždy nový apod. Vše vám podrobně vysvětlí dokumentace jazyka Python.

7.1 Konstrukce `global` a `del`

Tyto dvě konstrukce těsně souvisí s proměnnými, ta první deklaruje proměnné jako globální, druhá odstraní proměnné z prostoru jmen. Jejich syntaxe je velice jednoduchá, za klíčovým slovem `global` nebo `del` následuje seznam proměnných oddělených čárkou, které chcete deklarovat jako globální, resp. odstranit:

```
>>> a = b = c = 10
>>> global a, c     # a, c jsou globalni promenne
>>> del a, b        # odstranime promenne a, b
```

Deklarování globální proměnné platí od místa, kde tak bylo učiněno, do konce aktuálního bloku kódu. Blok kódu je: modul, tělo funkce, definice třídy, každý příkaz, který je zadán interaktivně, je také samostatný blok kódu, soubor s kódem a dále kód předaný konstrukci `exec` nebo funkcím `eval()`, `execfile()` a `input()`. Proměnné, která má být deklarována jako globální, nesmí být před konstrukcí global přiřazena žádná hodnota.

Kapitola 8

Definování uživatelských funkcí

Definováním funkce vytvoříme nový spustitelný objekt a nový záznam v lokálním prostoru jmen. Spustitelný funkční objekt obsahuje informace o svém vlastním kódu a hlavně o globálním prostoru jmen, který se použije, je-li jeho kód vykonáván. Definice funkce pouze vytvoří nový objekt, kód je vykonán až při jeho spuštění – zavolání funkce. Samozřejmostí je rekurze (maximální hloubka zanoření je omezena, její velikost zjišťujeme funkcí `sys.getrecursionlimit()` a nastavujeme funkcí `sys.setrecursionlimit()`, implicitně je nastavena na 1000 a překročení tohoto limitu vede k výjimce, ne k pádu interpretru), je také možné používat vnořené funkce. Ty ale nemají přístup k proměnným vnější funkce, dokumentace k jazyku se zmiňuje o možnosti obejít tento „nedostatek“, který vyplývá z filozofie dvou prostorů jmen, použitím nepovinných parametrů a jejich výchozích hodnot. Nakolik je ale toto řešení elegantní, nechám na vašem zvážení.

Jméno funkce v programu vystupuje jako proměnná, které je přiřazen funkční objekt. Funkci proto můžeme předávat jako parametr jiným funkcím, vložit do seznamu apod. Je to plnohodnotný objekt a jako takový se také chová. Funkční objekt má i několik atributů, jejichž popis však přesahuje rámec našeho seriálu. Případné zájemce odkážu do dokumentace k jazyku.

Pro definování uživatelských funkcí se používá konstrukce `def`. Za klíčovým slovem `def` následuje jméno funkce a v závorkách seznam formálních parametrů oddělených čárkou. Celý řádek je ukončen dvojtečkou. Na dalších řádcích následuje tělo funkce odsazené od kraje podobně jako tělo konstrukce `if` nebo `for`. Nejjednodušší definice funkce tedy může vypadat takto:

```
>>> def tiskni_text(answer, text):
...     print answer, text or 'prazdny retezec'
...
>>> tiskni_text('Parametr text je', '')
Parametr text je prazdny retezec
>>> tiskni_text('Parametr text je', 'www.root.cz')
Parametr text je www.root.cz
```

Funkce také nemusí mít žádné parametry, pak jsou místo výčtu formálních parametrů uvedeny pouze závorky, např. `getmax()`. V Pythonu existují i nepovinné parametry, které mohou být při volání funkce vynechány, mají zadánu výchozí hodnotu, která je použita, není-li tento formální parametr použit. Nepovinných formálních parametrů může být i více, ale vždy

musí platit: všechny nepovinné formální parametry v definici funkce následují po parametrech povinných. Je také třeba mít na paměti, že výraz, který přiřazujeme nepovinnému formálnímu parametru, je vyhodnocen jen při definici funkce (jeho hodnota je dále uchována ve funkčním objektu)!

```
>>> def tiskni_text(answer, text = 'nedefinovan', pocet = 1):
...     for foo in range(pocet):
...         print answer, text or 'prazdny retezec'
...
>>> tiskni_text('Parametr text je')
Parametr text je nedefinovan
>>> tiskni_text('Parametr text je', 'obycejny text')
Parametr text je obycejny text
>>> tiskni_text('Parametr text je', 'obycejny text', 3)
Parametr text je obycejny text
Parametr text je obycejny text
Parametr text je obycejny text
```

Výše uvedený způsob volání funkce je stejný jako v jiných jazycích, za jménem funkce následuje výpis skutečných parametrů. Tyto parametry nazýváme pozičními. Python zná ještě *keyword* parametry (parametry používající klíčová slova). Volání funkce `tiskni_text` můžeme za použití keyword parametrů zapsat takto:

```
>>> tiskni_text(answer = 'Parametr text je', text = 'obycejny text')
Parametr text je obycejny text
```

Keyword parametry můžeme psát i na přeskáčku, tj. v jiném pořadí, než jsou zapsány při definici funkce. Použijeme je, chceme-li hodnotu přiřadit jen určitému nepovinnému parametru. Chceme-li třikrát vytisknout text 'Parametr text je nedefinován', použijeme zápis:

```
>>> tiskni_text('Parametr text je', pocet = 3)
Parametr text je nedefinovan
Parametr text je nedefinovan
Parametr text je nedefinovan
```

Zde jsme zkombinovali poziční a keyword parametry. Opět musí platit několik pravidel. Především – zápis pozičních parametrů musí předcházet zápisu keyword parametrů. Dále hodnota předaná pomocí pozičního parametru nesmí být zároveň předána keyword argumentem. Zápis

```
>>> tiskni_text('Hello world', answer = 'Nazdar svete', pocet = 2)
```

vede k výjimce. Stejně tak musí být při volání naplněny všechny povinné formální parametry.

Co se ale stane, když funkci předáme nějaké poziční nebo keyword parametry navíc? U syntaxe, kterou jsme si ukázali v předchozích odstavcích, dojde k výjimce. Je ale možné použít dvě speciální formy zápisu formálních parametrů: `*identifikator` a `**identifikator`, přičemž je nemusíme použít vůbec nebo můžeme použít jen jeden, popř. můžeme použít oba dva. Do

proměnné identifikator jsou při použití syntaxe `*identifikator` uloženy všechny přebývající poziční parametry, při syntaxi `**identifikator` všechny přebývající keyword parametry. Poziční parametry jsou uloženy jako typ tuple, keyword parametry jako asociativní pole, jehož indexy jsou jména keyword parametrů a hodnotami jsou vlastní skutečné parametry předané funkci formou keyword parametrů.

```
>>> def tiskni_text(answer, text = 'nedefinovan', *args, **kwargs):
...     print answer, text or 'prazdny retezec'
...     print 'zbyvajici pozicni parametry:', args
...     print 'zbyvajici keyword parametry:', kwargs
...
>>> tiskni_text('Parametr text je', 'linux', 'Ahoj', server = 'www.root.cz')
Parametr text je linux
zbyvajici pozicni parametry: ('Ahoj',)
zbyvajici keyword parametry: {'server': 'www.root.cz'}
```

I při volání funkce můžeme jako skutečné parametry použít syntaxi `*identifikator` a `**identifikator`. Hodnoty z těchto parametrů pak budou „sloučeny“ s pozičními a keyword parametry předanými funkci. Zápis hvězdičkovaných parametrů musí být uveden po výčtu všech pozičních a keyword parametrů a nejprve musí být uveden (je-li použit) `*identifikator`, až poté (je-li použit) `**identifikator`. Klíči asociativního pole, které má sloužit jako `**identifikator`, musí být řetězce, jinak dojde k výjimce. Jako ukázka nechť slouží následující volání funkce `tiskni_text`:

```
>>> pozicni = ('prvni', 'druhy', 'treti')
>>> keyword = {'os': 'linux'}
>>> tiskni_text('Parametr text je', 'linux', 'Ahoj', server = 'www.root.cz', \
... *pozicni, **keyword)
Parametr text je linux
zbyvajici pozicni parametry: ('Ahoj', 'prvni', 'druhy', 'treti')
zbyvajici keyword parametry: {'server': 'www.root.cz', 'os': 'linux'}
```

8.1 Použití konstrukce `global`

Jak jsme si již řekli, proměnné z globálního prostoru jmen jsou přístupné pro čtení kdykoli, ale chceme-li jim přiřadit nějakou hodnotu, musíme je deklarovat jako globální. Největší význam to má právě při použití s funkcemi a metodami (popravdě řečeno, jiné použití mě ani nenapadá, jestli víte ještě o jiném, mně skrytém použití konstrukce `global`, podělte se s námi v diskusi).

```
>>> g = 0                               # (1)
>>> def f():                            # (2)
...     g = 1                           # (3)
...     print 'f(): g =', g             # (4)
...                                     # (5)
>>> f()                                 # (6)
f(): g = 1
```

```
>>> print 'global: g =', g              # (7)
global: g = 0
>>> def f2():                           # (8)
...     global g                        # (9)
...     g = 2                           # (10)
...     print 'f2(): g =', g            # (11)
...                                     # (12)
>>> f2()                                # (13)
f2(): g = 2
>>> print 'global: g =', g              # (14)
global: g = 2
```

Na začátku naší ukázky jsme si vytvořili proměnnou `g`. Provedli jsme to v hlavním bloku modulu, takže `g` je uloženo v globálním prostoru jmen. Na řádku (2) jsme si vytvořili novou funkci `f()`, která proměnné `g` přiřadí hodnotu 1 a ještě hodnotu `g` vytiskne. Na řádku (6) funkci `f()` zavoláme a zjistíme, že uvnitř jejího těla má proměnná `g` opravdu hodnotu 1. Na řádku (7) si zkontrolujeme hodnotu `g`, ale jaké je překvapení, že má pořád hodnotu 0. Vysvětlení je jednoduché. Přiřazení na řádku (3), místo aby změnilo hodnotu globální proměnné `g`, vytvořilo novou lokální proměnnou, která překryla globální, a proto i kontrolní tisk na řádku (4) vytiskl hodnotu 1. Hodnota globální proměnné `g` ale zůstala nezměněna. To řeší až funkce `f2()`, která používá konstrukci `global`. Pak již vše pracuje, jak má.

8.2 Návratová hodnota funkce

Návrat z uživatelské funkce provede konstrukce `return`, které mohou být předány argumenty, jež se použijí jako návratová hodnota funkce. Return bez argumentů, stejně jako ukončení funkce bez konstrukce `return`, vrátí hodnotu `None`, což je vestavěná proměnná s významem podobným, jaký má v céčku null.

8.3 Výjimky a funkce

Vznikne-li ve funkci neošetřená výjimka, rozšíří se z této funkce do funkce volající atd., dokud není odchycena. Nedojde-li k odchycení výjimky a ta se rozšíří až do hlavního modulu, je standardní akcí výpis volaných funkcí, informací o výjimce a ukončení programu. Více si o výjimkách povíme, až budeme probírat třídy a jejich používání. Výjimky totiž nejsou nic jiného než třídy, takže je u nich možno používat dědičnost apod.

Kapitola 9

Moduly

Pryč jsou doby, kdy se zdrojové kódy programů psaly do jednoho souboru bez ladu a skladu, pryč jsou časy, kdy nejsložitější programy měly pár stovek řádek. V dnešní době, kdy je kladen velký důraz na opakované použití kódu a na programy složené z komponent, je třeba program rozdělit do několika částí. V jazyce Python jde o *moduly* a balíčky modulů a další pokračování tohoto seriálu bude právě o nich.

9.1 Příklad

Jak jsme si již řekli v jedné z prvních částí našeho seriálu, program v Pythonu nemusíme psát přímo v prostředí interaktivního interpretu, ale lze ho zapsat i do zdrojového souboru, a ten pak dále spouštět. V dalším výkladu budeme dále pracovat s tímto zdrojovým souborem (module1.py):

```
#!/usr/bin/env python                                  # (1)
'Ukazkovy modul v jazyce Python'                       # (2)
import sys                                             # (3)
import os                                              # (4)
                                                       # (5)
default_path = os.environ['PATH']                      # (6)
                                                       # (7)
def print_environment(env = None):                     # (8)
    '''Vytiskne promenne prostredi v poli env,         # (9)
    neni-li zadano, uvazuje os.environ'''              # (10)
    env = env or os.environ                            # (11)
    for var in env.keys():                             # (12)
        print '$' + var + '=' + env[var]               # (13)
                                                       # (14)
def list_path(path = None):                            # (15)
   '''Vrati seznam vsech souboru v ceste path,         # (16)
    neni-li zadano, uvazuje globalni promennou         # (17)
    default_path'''                                    # (18)
    path = path or default_path                        # (19)
    path_l = path.split(':')                           # (20)
```

```
    retval = []                                     # (21)
    for el in path_l:                               # (22)
        try:                                        # (23)
            retval.extend(os.listdir(el))           # (24)
        except OSError:                             # (25)
            pass                                    # (26)
    return retval                                   # (27)
                                                    # (28)
if __name__ == '__main__':                          # (29)
    my_env = {'PATH': os.environ['PATH']}           # (30)
    print_environment(my_env)                       # (31)
    print_environment()                             # (32)
    print list_path()[:50]                          # (33)
```

Takto nějak může vypadat modul v jazyce Python. Na prvním řádku může být magická sekvence pro spuštění modulu přímo v Pythonu, má-li soubor nastaven executable bit. Následuje importování dvou modulů `sys` a `os`. Dále definujeme jednu globální proměnnou (globální v rámci tohoto modulu) a nakonec definujeme pár funkcí, které náš modul bude poskytovat. Výše uvedený soubor module1.py uložíme do libovolného pracovního adresáře a v tomto adresáři si spustíme interaktivní interpret příkazem `python`:

```
honza@localhost ~/tmp $ python
Python 2.0 (#1, Apr 11 2001, 19:18:08)
[GCC 2.96 20000731 (Linux-Mandrake 8.0)] on linux-i386
Type "copyright", "credits" or "license" for more information.
>>> import module1                                  # (1)
>>> dir()                                           # (2)
['__builtins__', '__doc__', '__name__', 'module1']
>>> dir(module1)                                    # (3)
['__builtins__', '__doc__', '__file__',
'__name__', 'default_path', 'list_path', 'os',
'print_environment', 'sys']
>>> module1.default_path                            # (4)
'/usr/bin:/bin:/usr/X11R6/bin:/usr/local/bin'
```

Každý soubor se zdrojovým kódem v jazyce Python a s příponou .py může zároveň být i modulem. Zavedení modulu (importování) se provádí (překvapivě :) konstrukcí `import` (řádek 1), která může mít mnoho tvarů. V každém případě ale import vytváří nový záznam (záznamy) v aktuálním lokálním prostoru jmen, jak ukazuje volání funkce `dir()` na řádku (2).

9.2 Tečková notace

Pro přístup k atributům libovolného objektu se v Pythonu používá tečková notace. Zápis `os.path` tedy znamená atribut '`path`' objektu '`os`'. Atributy mohou mít samozřejmě další atributy, například `os.path.split`.

9.3 Moduly

Importováním modulu vznikne nový objekt typu module, jehož atributy jsou proměnné v globálním prostoru jmen tohoto modulu. K těmto atributům se samozřejmě přistupuje pomocí tečkové notace. To můžeme vidět i na řádku (4) výše uvedeného příkladu.

Každý modul má kromě těch atributů, které mu přiřadil jeho tvůrce, i atributy, které mu dal do vínku interpret Pythonu. Jsou to hlavně:

__builtins__	asociativní pole, které má význam builtin prostoru jmen modulu
__doc__	dokumentační řetězec modulu
__file__	jméno souboru, ze kterého byl modul zaveden
__name__	jméno modulu

9.4 Dokumentační řetězce

Každý modul (a nejen modul, ale i třída, její metoda a libovolná funkce) má speciální atribut se jménem `__doc__`. Ten obsahuje takzvaný dokumentační řetězec, (řádky 2, 9–10 a 16–18 souboru example1.py), ve kterém může být zapsán komentář k modulu (třídě, metodě). Zapisuje se jako obyčejný řetězec hned na začátek souboru před vše ostatní (podobně následuje hned za definicí třídy nebo metody). Tyto komentáře jsou nepovinnou součástí definice, je ale zvykem je psát, neboť velice usnadňují práci v interaktivním interpretu.

9.5 Konstrukce `import`

Konstrukce `import` je snad nejvariabilnější konstrukcí v Pythonu. Následuje několik ukázek použití, pro kompletní přehled možných kombinací vám doporučuji referenční příručku jazyka.

```
>>> import module1                              # (1)
>>> import module1 as m                         # (2)
>>> import module1, os as opersys               # (3)
>>> from module1 import default_path, \         # (4)
... print_environment as pe                     # (5)
>>> from module1 import *                       # (6)
```

Konstrukce `import` nejprve najde modul, který je třeba importovat, poté spustí jeho tělo. Tím se definují globální proměnné (viz. proměnná `default_path` v module1) a také další objekty, které bude modul obsahovat (funkce print_environment, list_path). Takto se spustí úplně libovolný kód, např. na řádku (29) souboru example1.py vidíme testování jména modulu, je-li rovno `__main__`, a modul je tedy načten jako hlavní modul, dojde k vykonání ukázkových příkazů (podobné konstrukce se používají u většiny modulů, viz. zdrojové kódy jednotlivých modulů). Po spuštění těla modulu následuje vytvoření nových jmen v lokálním prostoru jmen bloku, který si vyžádal zavedení modulu.

Řádek (1) ukazuje nejčastější použití konstrukce `import`, pouze vytvoří v lokálním prostoru jmen proměnnou `module1` typu modul. Další řádek (2) ukazuje importování modulu pod jiným jménem. Importovat lze i více modulů najednou, konkrétně to ukazuje řádek (3), přičemž i v tomto případě lze některé moduly importovat pod jiným jménem. Atributem modulu `__name__` ale zůstává stále jméno původního modulu. Z modulů můžeme do lokálního prostoru

jmen importovat jen některé atributy (4)(5), opět je možné je importovat i pod jinými jmény, než pod jakými jsou obsaženy v modulu. Je-li místo výčtu atributů uvedena hvězdička, zavedou se do lokálního prostoru jmen všechny atributy modulu (s výjimkou jmen, která začínají znakem podtržítko _). Zde je třeba zmínit, že při použití klíčového slova `from` se nevytvoří proměnná odkazující na modul, ale jen odkazy na ty proměnné, které si programátor přeje naimportovat (a jejichž výčet následuje za slovem `import`).

Narazí-li interpret na konstrukci `import`, začne hledat modul, který odpovídá požadovanému jménu. Nejprve se podívá do interního seznamu dosud importovaných modulů, není-li již zaveden. Tento seznam je přístupný jako `sys.modules` (čili proměnná `modules` v modulu `sys`). Jde o asociativní pole, jehož klíči jsou jména modulů a hodnotami moduly, na něž jména odkazují. Do tohoto pole je umožněn i zápis, čili je možné takto vytvářet i nové moduly, aniž by byly fyzicky přítomny v souborovém systému (doporučuji si prohlédnout zdrojový kód modulu `os`, který takto vytváří modul `os.path`).

Není-li modul nalezen v interním seznamu, pokračuje se s jeho hledáním v aktuálním adresáři a pak v adresářích v seznamu `sys.path` (který se generuje z proměnné prostředí `$PYTHONPATH` a z cest závislých na instalaci jazyka). Pro modul 'module1' se nejprve hledá soubor 'module1.pyc', což je „zkompilovaná“ verze souboru 'module1.py', není-li nalezen, zavede se soubor module1.py. Je-li soubor module1.py novější než module1.pyc, předpokládá interpretr, že obsah module1.pyc je neaktuální a vytvoří ho znovu z module1.py. Zároveň se vytvoří i soubor 'module1.pyc', který urychlí příští zavedení modulu (není-li ho možné vytvořit, nic se neděje, pouze bude další start zpomalen kompilací). Je dobré vědět, že spuštěním hlavního modulu (přístupného jako `__main__`) nedojde k jeho kompilaci, proto je výhodnější napsat menší spouštěcí skript, jenž zavolá hlavní funkci ve velkém modulu, který se již zkompiluje. Dosáhneme tím rychlejšího spouštění programu. Taktéž je možné používat pouze zkompilované .pyc moduly, ze kterých se nedá zpět získat původní zdrojový kód, Python takto umožňuje vývoj i closed-source programů.

Není-li modul nalezen, dojde k výjimce `ImportError`, je-li modul nalezen, ale není syntakticky správně, vyvolá interpret výjimku `SyntaxError`. Vznikne-li výjimka při inicalizaci modulu, rozšíří se do volajícího bloku a modul zůstane *nezinicializovaný*! Přesto se vytvoří záznam v `sys.modules`. Proto si dávejte pozor a důsledně kontrolujte výjimky, které se mohou při importování modulu rozšířit, poněvadž pokusíme-li se importovat nezinicializovaný modul, interpret nám bez problémů vyhoví.

9.6 Balíčky

Pro velké kolekce modulů se vyplatí používat více modulů a i zde přichází Python s výraznou pomocí, kterou jsou balíčky. Jedná se vlastně o moduly uložené v adresářové struktuře. Narazí-li interpret na konstrukci, která požaduje importování modulu s hierarchickým jménem, např: `import package.module`, považuje 'package' za balíček a 'module' za modul, který je v něm uložený.

Nejjednodušším balíčkem je pouhý adresář umístěný v místech, kde se vyhledávají klasické moduly, a který obsahuje soubor `__init__.py` a samozřejmě moduly v něm uložené. Soubor `__init__.py` se spustí jako inicializační část balíčku, poté se přistoupí k importování modulů z balíčku. Soubor `__init__.py` může teoreticky být i pouhý prázdný soubor, v praxi se tak ovšem (většinou) neděje.

9.7 Proměnná __all__

Co se stane, když ale programátor požaduje konstrukcí `from package import *` naimportování všech modulů z balíčku? Dalo by se očekávat, že se naimportují všechny moduly v balíčku. Na různých platformách ale platí různé konvence pro zápis jmen souborů, konkrétně Windows (o DOSu nemluvě) nerozlišují velikost písmen v souborech a soubor `FOO.PY` by mohl být importován jako modul `FOO`, ale také jako `Foo` nebo `foo`. Proto se v inicializační části modulu nastavuje proměnná `__all__`, což je seznam řetězců, které explicitně určují, jaké moduly balíček obsahuje. Udržování této proměnné je pouze na autorovi balíčku, který ručí za to, že odpovídá skutečnosti.

Není-li proměnná `__all__` nastavena, je možné rovněž vykonat konstrukci `from package import *`, ale v aktuálním prostoru jmen se vytvoří odkazy jen na proměnné z globálního prostoru jmen souboru `__init__.py` balíčku 'package'. Chce-li programátor z modulu obsaženého v nějakém balíčku importovat jiný modul z téhož balíčku, musí jeho jméno zapsat plnou cestou. Máme-li balíček Languages, který obsahuje moduly Python, Perl a Ruby, a v modulu Python chceme používat modul Ruby, musíme ho importovat jako Languages.Ruby. Je samozřejmé, že balíčky mohou kromě modulů obsahovat další balíčky a lze tak vytvořit velice jemnou strukturu modulů.

Protože modul je stejně jako libovolný objekt pouze odkaz do paměti interpretu, lze ho tímto způsobem používat. Používáme-li v nějakém modulu jiné moduly, je slušností v době, kdy je již nebudeme potřebovat, odstranit z prostoru jmen konstrukcí `del` (např. `del Languages.Ruby`).

Kapitola 10

Objektové programování

Jak již bylo na mnoha místech napsáno, Python *není* plně objektový jazyk, obsahuje však ty nejdůležitější vlastnosti OOP[1] – *dědičnost* (samozřejmostí je i vícenásobná dědičnost), *polymorfismus* a *zapouzdření*. Objekt není v Pythonu chápán pouze jako instance třídy, ale jde o data určitého typu uložená v paměti. Pouze některé objekty jsou instancemi některé třídy, jiné mohou být uživatelské funkce, moduly nebo jednoduché typy (čísla, řetězce, tuple, seznamy...). Objektem je taktéž třída.

Každý objekt může mít množství atributů a metod. V Pythonu je každý atribut (jako atribut bereme i metodu) veřejný (public), čili libovolný objekt má přístup k atributům jiného objektu. Přeje-li si programátor třídy skutečně soukromé (private) atributy, musí třídu implementovat jako rozšiřující modul v jazyce C/C++.

10.1 Definování třídy

Třídu definujeme pomocí konstrukce `class`:

```
>>> class pes:
...     'Obecny domaci mazlicek'
...     def __init__(self, b = 'cerna'):
...         self.barva = b
...     def ukaz(self):
...         print 'Toto je pes:', self
...         print 'Barva srsti:', self.barva
...
```

Po klíčovém slově `class` následuje jméno třídy a celé tělo definice třídy je odsazeno, stejně jako při použití kterékoli jiné složené konstrukce (definice funkce, konstrukce `if` atd.). Pro tělo definice třídy se vytvoří nový lokální prostor jmen a tento prostor jmen je zachován i po opuštění těla definice.

Konstrukce `class` vytvoří nový objekt typu třída. Všechny objekty tohoto typu podporují dvě operace – odkazování se na atributy (pomocí tečkové notace) a vytváření instancí:

```
>>> pes.ukaz                          # (1)
```

[1] OOP – *O*bject *O*riented *P*rogramming (objektově orientované programování)

```
<unbound method pes.ukaz>
>>> pes.__doc__                          # (2)
'Obecny domaci mazlicek'
```

Atributy objektu třída jsou vlastně jména z lokálního prostoru jmen, který se vytvořil při definici funkce. Jak je vidět z řádku (2), i třídy mohou mít dokumentační řetězce. Každá třída má ještě pár speciálních atributů, které jí přiřadil Python:

__name__	Jméno třídy jako řetězec
__bases__	Tuple všech předků třídy
__doc__	Dokumentační řetězec třídy
__module__	Jméno modulu, ve kterém je třída definována

Instance třídy se vytvářejí „zavoláním“ třídy (podobně jako volání funkce), přičemž toto „volání“ vrátí instanci třídy:

```
>>> cerny = pes()                        # (1)
>>> bily = pes('bila')                   # (2)
>>> cerny.ukaz()                         # (3)
Barva srsti naseho pejska je cerna
>>> bily.ukaz()                          # (4)
Barva srsti naseho pejska je bila
>>> pes.ukaz(cerny)                      # (5)
Barva srsti naseho pejska je cerna
>>> pes.ukaz()       # CHYBNĚ!!!         # (6)
```

Každá metoda v Pythonu má ještě jeden argument navíc – většinou je pojmenován `self` – pomocí něhož se předává instance, pro kterou byla tato metoda zavolána. Tento argument musí být použit u každé (!) metody, která se definuje v těle třídy, a musí se jednat o její první argument. Python nepoužívá žádného zkráceného zápisu pro přístup k instančním atributům, vždy se musí používat zápisu `self.atribut` (viz řádek (4) v definici třídy pes). Toto opatření zvyšuje čitelnost kódu a zabraňuje kolizím mezi lokálními proměnnými a proměnnými instance.

Máme-li vytvořenu instanci třídy, odkazujeme se na její atributy pomocí tečkové notace. Voláme-li metodu této instance, postupujeme stejně jako při volání obyčejné funkce, přičemž za argument `self` interpret automaticky dosadí tuto instanci. Každé volání metody se ve skutečnosti převede na volání funkce, která je definována v těle třídy. Takže volání `cerny.ukaz()` se převede na `pes.ukaz(cerny)`. Je ale důležité si uvědomit, že `cerny.ukaz` a `pes.ukaz` není jedno a totéž, v prvním případě se jedná o metodu třídy pes, ve druhém o funkční objekt, definovaný uvnitř třídy `pes`. S metodami je možné pracovat úplně stejně jako s funkcemi, je možné je předávat v místech, kde se očekávají funkce apod. Není dovoleno volat metodu bez instance třídy, řádek (6) tudíž vede k výjimce. [2]

Při vytvoření nové instance třídy je zavolána speciální metoda této třídy `__init__`, jíž jsou předány ty argumenty, které třída dostala při žádosti o vytvoření nové instance. Jde o obdobu konstruktoru v C/C++. Python podporuje i další speciální metody (zde je uveden pouze krátký nástin problému, více viz dokumentace k jazyku). Stejný princip se používá i při přetěžování operátorů.

[2] Od verze 2.2 je možné již vytvářet třídní metody, které nevyžadují instanci třídy, viz dále.

__del__	Destruktor, volán při smazání objektu garbage collectorem
__str__	Zavolána při žádosti o získání popisku objektu vestavěnou funkcí `str()`
__getattr__	Zavolána při čtení atributu objektu
__setattr__	Zavolána při nastavení hodnotu atributu

Veškerá jména definovaná uvnitř prostoru jmen třídy se šíří i do jednotlivých instancí, proto `cerny.__doc__` je totéž jako `pes.__doc__`.

```
>>> class test:                                  # (1)
...     value = 'definovano ve tride'            # (2)
...     def __init__(self):                      # (3)
...         self.value = 'definovano v instanci' # (4)
...                                              # (5)
>>> obj = test()                                 # (6)
>>> test.value                                   # (7)
'definovano ve tride'
>>> obj.value                                    # (8)
'definovano v instanci'
```

Na řádku (8) je vidět, jak přiřazení hodnoty někde v instanci skryje hodnotu přiřazenou ve třídě. Přitom se původní hodnota definovaná uvnitř třídy nezmění, viz. řádek (7). Toho se využívá například při definici výchozích hodnot, které jsou společné pro všechny instance dané třídy.

10.2 Dědičnost

Nyní máme definovánu třídu `pes` a chceme od ní odvodit novou třídu, která bude navíc popisovat délku srsti psa.

```
>>> class jezevcik (pes):                     # (1)
...     'Dlouhosrsty jezevcik'                # (2)
...     def __init__(self, d = 4):            # (3)
...         pes.__init__(self, b = 'hneda')   # (4)
...         self.delka = d                    # (5)
...     def ukaz(self):                       # (6)
...         pes.ukaz(self)                    # (7)
...         print 'Delka srsti:', self.delka  # (8)
...
>>> punta = jezevcik()                        # (9)
>>> punta.ukaz()                              # (10)
Toto je pes: <__main__.jezevcik instance at 0x82a6a8c>
Barva srsti: hneda
Delka srsti: 4
```

V Pythonu je možné používat i vícenásobnou dědičnost, pak se v závorkách na řádku (1) objeví jednotlivé rodičovské třídy oddělené čárkou. Všechny metody v Pythonu jsou automaticky virtuální (viz. řádek (10), kde se volá metoda třídy `jezevcik` a ne třídy `pes`). Vyplývá

to ze způsobu, jakým Python postupuje při hledání atributu (a tudíž i metody) objektu. Ten nejprve prohlédne třídu objektu, pak první rodičovskou třídu této třídy, totéž udělá s dalšími předky této třídy; není-li zde atribut nalezen, pokračuje u další rodičovské třídy atd. Použije se první nalezený atribut (nebo metoda, což je totéž). Nenajde-li interpret žádný odpovídající atribut, dojde k výjimce.

Chceme-li volat metodu některé z rodičovských tříd, použijeme zápis podobný tomu, jako je na řádku (7), zavoláme funkci `ukaz` třídy `pes` pro objekt `self`. Podobný zápis jsme mohli vidět i na řádku (4), kde inicializujeme instanci pomocí rodičovského konstruktoru.

10.3 Pár slov o instancích

Jak instance vznikne, jsme si již řekli. Její odstranění není již tak jednoduchou záležitostí, protože Python má zabudovaný garbage collector založený na počítání odkazů. Instance třídy jsou proto odstraněny, až když na ně neexistuje žádný odkaz. Proto může každá třída definovat podobně jako metodu `__init__` i metodu `__del__`, která je volána při odstranění instance.

Instancím přiděluje interpret také speciální atribut `__class__`, který odkazuje na třídu objektu. Python také obsahuje dvě vestavěné funkce, které jsou užitečné v objektově orientovaném programování:

- `isinstance(object, classinfo)` – návratová hodnota je `true`, platí-li, že argument `object` (instance třídy) je potomkem (přímým i nepřímým) argumentu `classinfo` (třídy), ve verzi 2.2 může `classinfo` být i tuple, který obsahuje třídy, pak je návratová hodnota `true`, je-li `object` potomkem alespoň jedné třídy v `classinfo`.

- `issubclass(class1, class2)` – vrátí `true`, platí-li, že třída `class1` je potomkem třídy `class2`. Je-li `class1` rovno `class2`, funkce vrátí také `true`.

10.4 Dědění od vestavěných typů

Od verze 2.0 prodělal Python množství změn a bylo odstraněno několik „neduhů“, které mu byly vytýkány. Jedním z nich byla nemožnost dědit od vestavěných typů. V současné době je každý vestavěný typ třídou a je možné předefinovat jeho chování klasickou cestou pomocí speciálních metod.

Jako příklad si ukážeme definování vlastního typu `MyInt`, který bude potomkem klasického integeru, při převodu na řetězec funkcí `str()` však vrátí upravený text. Veškeré vestavěné typy (ty, které definuje samotný interpret a které jsou implementovány v C/C++) najdeme uvnitř modulu `types`. Příklad je všeříkající a nemá smysl ho vysvětlovat:

```
>>> import types
>>> class MyInt (types.IntType):
...     def __str__(self):
...         return 'toto je integer s hodnotou ' + `self`
...
>>> cislo = MyInt(21)
>>> str(cislo)
'toto je integer s hodnotou 21'
```

10.5 Třídní a statické metody

Novinkou tak horkou, že není uvedena ani v originální dokumentaci, jsou *třídní a statické metody*. Třídní metoda je taková metoda, která dostane jako první argument místo instance třídu samotnou, přičemž je jedno, je-li volána jako metoda instance nebo jako funkce, která je definována ve funkci. Pro vytvoření třídní metody vytvoříme uvnitř třídy klasickou funkci, jejímž prvním argumentem bude odkaz na třídu, a poté ji pomocí funkce `classmethod()` „zkonvertujeme“:

```
>>> class c:
...     value = 'class'
...     def __init__(self):
...         self.value = 'instance'
...     def print_instance(self):
...         print self.value
...     def print_class(cls):
...         print cls.value
...     print_class = classmethod(print_class)
...
>>> ins = c()                           # (1)
>>> ins.print_instance()                # (2)
instance
>>> ins.print_class()                   # (3)
class
>>> c.print_class()                     # (4)
class
>>> c.print_instance()    # CHYBNĚ!!!   # (5)
```

Z řádků (3) a (4) je vidět, že je úplně jedno, voláme-li třídní metodu jako metodu instance nebo jako metodu třídy. Vždy dojde k tomu, že se za argument `cls` funkce `print_class` dosadí třída `c`. Odvodíme-li nyní od třídy `c` novou třídu `d` a zavoláme pro ni (nebo pro její instanci) metodu `print_class`, dosadí se za atribut `cls` třída `d`. Na řádku (5) je vidět chybný zápis, kdy voláme instanční metodu, jako by to byla metoda třídy. Tento zápis vede k chybě, kdy si interpret stěžuje na chybějící instanci třídy.

Výše nastíněný mechanismus třídních metod doplňují ještě *statické metody*. To jsou přesně ty statické metody, jak je jistě znáte z C++ nebo z Javy. Ty nepřebírají žádný argument reprezentující instanci nebo třídu. Jde o jakési funkce definované uvnitř třídy. Vytvářejí se podobně jako třídní metody funkcí `staticmethod()`, které předáme funkční objekt definovaný někdy dříve.

10.6 Mechanismus vlastností – properties

Věcí, která se před příchodem 2.2 musela složitě programovat a kterou řešil téměř každý programátor, byly *vlastnosti* (properties, netřeba představovat například programátorům v Delphi), k jejichž hodnotám je přístup zprostředkován pomocí metod. Při čtení atributu je vrácena návratová hodnota přiřazené funkce, podobně je tomu i při zápisu a smazání atributu.

Jednoduchý příklad:

```
>> class prop(object):
...     def __init__(self): self._value = None
...     def _get_value(self):
...         print '_get_value()'
...         return self._value
...     def _set_value(self, value):
...         print '_set_value()'
...         self._value = value
...     def _del_value(self):
...         print '_del_value()'
...         print '_value nesmazana'
...     value = property(_get_value, _set_value, \
... _del_value, 'Ukazka properties')
...
>>> obj = prop()
>>> obj.value = 'text'
_set_value()
>>> obj.value
_get_value()
'text'
>>> del obj.value
_del_value()
_value nesmazana
>>> prop.value.__doc__
'Ukazka properties'
```

Poměrně rozsáhlý příklad, ale veškeré jádro pudla je obsaženo v jediných dvou věcech:

- Třída, v níž chceme properties používat, musí být potomkem typu `object`. Bez splnění této podmínky nebudou properties pracovat správně.
- Vlastnost vytvoříme voláním funkce `property`, která přebírá tyto nepovinné argumenty:
  `fget` – metoda pro získání hodnoty vlastnosti
  `fset` – metoda pro nastavení hodnoty vlastnosti
  `fdel` – metoda, která zařídí vymazání vlastnosti z prostoru jmen
  `doc` – dokumentační řetězec vlastnosti

Nebude-li některý argument nastaven, bude daná akce zakázána a při pokusu o její provedení dojde k výjimce.

10.7 Metody `__getattr__` a `__setattr__`

Jak jsme si řekli v minulém dílu našeho povídání, je možné vytvořit vlastní metody, které zprostředkovávají přístup k atributům objektu. Na stejném principu je založen i mechanismus vlastností, o kterém jsme se již zmínili. Nejprve si ukážeme příklad:

```
>>> class Double:                                    # (1)
```

```
...     def __init__(self, val = 0):              # (2)
...             self.value = float(val)               # (3)
...                                               # (4)
...     def __getattr__(self, name):              # (5)
...             if name == 'double':                  # (6)
...                     return self.value * 2             # (7)
...             raise AttributeError, name            # (8)
...                                                   # (9)
...     def __setattr__(self, name, value):       # (10)
...             if name == 'double':                  # (11)
...                     self.value = value / 2.0          # (12)
...                     return                            # (13)
...             self.__dict__[name] = value           # (14)
...
>>> i = Double(5)
>>> i.value
5.0
>>> i.double
10.0
>>> i.double = 20
>>> i.value
10.0
>>> dir(i)
['__doc__', '__getattr__', '__init__', '__module__',
'__setattr__', 'value']
```

Definovali jsme si třídu `Double`, která má dva atributy, první `value` je skutečným atributem, druhý `double` je dynamickým atributem počítaným z hodnoty value pomocí metody `__getattr__`. Všimněte si, že dynamické atributy (to jest ty, které jsou zpřístupněny pomocí metody `__getattr__`), nejsou obsaženy ve výpisu atributů pomocí vestavěné funkce `dir()`.

Metoda `__getattr__` společně s metodami `__setattr__` a `__delattr__` jsou nástroji pro vytváření dynamických atributů. Každá třída může definovat libovolnou z těchto metod. Metoda `__getattr__` přebírá kromě odkazu na instanci i jeden argument name, jenž je jménem, které chce interpret zpřístupnit. Tato metoda je spouštěna pouze tehdy, je-li požadován přístup k nějakému atributu, který není nalezen standardním postupem (čili není přímo součástí instance nebo třídy). Důvodem pro takové chování je především rychlý přístup k atributům instancí tříd, které používají jak klasické, tak dynamické atributy. Odkazujeme-li se uvnitř metody `__getattr__` na další atributy téže instance a ty nejsou nalezeny standardním postupem, je `__getattr__` volána znova. Proto je třeba dávat veliký pozor na ukončení rekurze. Jako konečná hodnota atributu se použije návratová hodnota metody `__getattr__`. Není-li atribut nalezen, je slušné vyvolat výjimku (viz řádek (8)), jinak by se jako hodnota atributu použila hodnota `None`, což je implicitní návratová hodnota, nepoužila-li metoda klíčové slovo `return` nebo bylo-li použito bez parametrů.

Metoda `__setattr__` je volána *vždy* při nastavení hodnoty určitého atributu. Zde je určitá asymetrie mezi metodami `__getattr__` a `__setattr__` a je dobré ji mít vždy na paměti. Metoda `__setattr__` přebírá dva argumenty. Argument `name` je jméno atributu, který se má změnit, a `value` obsahuje novou hodnota atributu. Přiřadíme-li uvnitř metody `__setattr__` nějakou

hodnotu libovolnému atributu téže instance, je metoda `__setattr__` volána znovu. Jestliže by výše uvedená metoda `__setattr__` vypadala například takto:

```
def __setattr__(self, name, value):
        if name == 'double':
                self.value = value / 2.0
                return
        if name == 'value':
                self.value = value        # !!!
        self.__dict__[name] = value
```

došlo by k nekonečnému cyklu. Proto se musí používat atribut instance `__dict__`, což je vlastně asociativní pole, které obsahuje jména atributů (klíč) a jejich hodnoty. Klíčové slovo `return` je použito pro předčasné ukončení metody, aby nedošlo k zařazení atributu `double` do pole `__dict__`, a tím k vytvoření klasického atributu (metoda `__getattr__` by pak již nikdy nebyla pro tento atribut volána).

Existuje ještě metoda `__delattr__`, která je volána vždy při smazání atributu konstrukcí `del`. Ta přebírá, podobně jako `__getattr__`, jeden argument `name`, obsahující jméno atributu, který se má smazat. Opět není možné použít přímé smazání hodnoty konstrukcí `del self.attribute`, ale musí se smazat odpovídající položka atributu `__dict__`.

Metody `__*attr__` musí být známé již v době překladu kódu, toto opatření je zavedeno kvůli zvýšení rychlosti běhu při interpretaci takového kódu. Závěrem je důležité dodat, že programování dynamických atributů je poměrně obtížné, daleko výhodnější a jednodušší je používat mechanismus vlastností, které přináší verze 2.2.

O problematice objektů se lze více dozvědět z originálního tutoriálu k Pythonu. Ovšem nedozvíte se z ní o nejnovějších rysech jazyka. U rysů, jako jsou třídní metody nebo vlastnosti, jsem čerpal z dokumentačních řetězců funkcí `classmethod()`, `staticmethod()` a `property()`. Verze 2.2 je stále ještě poměrně mladá, a proto prosím každého, kdo by můj výklad mohl upřesnit, případně odkázat na další zdroje, nechť mi dá vědět emailem.

Kapitola 11

Zpracování chyb

Až po předchozí díl jsme se o chybových stavech příliš nebavili. Nyní nastává čas si vše objasnit. Jazyk Python rozlišuje dva druhy chybových stavů: *syntaktické chyby* a *výjimky*. Oba druhy chyb mají ale stejný důsledek, dojde k vyvolání výjimky.

Začněme tím, jak se výjimky projevují. Poté, co dojde k určitému chybovému stavu, může program vyvolat výjimku. Tuto výjimku následně může program odchytit a přizpůsobit se chybě, která nastala. Každá chyba – výjimka – je určena svým identifikátorem, který specifikuje, o jaký druh chyby se jedná (např. `SyntaxError`, `AttributeError`, `OSError`). Výjimky mohou mít i své argumenty, které upřesňují, k jaké chybě došlo (výjimka `IOError` má tři argumenty: `errno` – číslo chyby, `filename` – jméno souboru, jehož se chyba týká, a `strerror` – textový popis chyby). Nejsou-li učiněny žádné kroky k zachycení výjimky, šíří se z volané funkce do volajících funkcí. Není-li výjimka vůbec odchycena, interpretr vypíše chybové hlášení sestávající z výpisu volaných funkcí a z informací o výjimce, tj. o jejím druhu a doplňujících informací. Takový výpis může vypadat třeba takto:

```
Traceback (most recent call last):
  File "igui/events.py", line 40, in Process
    consumer(self)
  File "test.py", line 30, in pokus
    window.events.OnClose.sender = 'foo'
  File "igui/events.py", line 54, in __setattr__
    raise AttributeError, "read-only atribute '%s'" % name
AttributeError: read-only atribute 'sender'
```

V tomto bodě se liší syntaktické chyby od výjimek. Při vzniku syntaktické chyby je vypsáno chybové hlášení a interpretr ještě ukáže přímo místo chyby na příslušné řádce (v tomto případě chybějící dvojtečka v konstrukci `for`):

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<string>", line 1
    for x in [1, 2, 3]
                     ^
SyntaxError: invalid syntax
```

11.1 Vznik výjimek

Výjimka je vyvolána konstrukcí `raise`. Za klíčovým slovem `raise` následuje identifikátor výjimky, případně ještě její argument. V Pythonu může být identifikátorem výjimky buď řetězec, nebo třída, nebo její instance. Nejjednodušší vyvolání výjimky může vypadat třeba takto:

```
>>> raise 'chyba vstupu/vystupu', {'soubor': '/home'}
```

V tomto případě je identifikátorem výjimky řetězec. Za tímto řetězcem může následovat i argument této výjimky. V počátcích jazyka byl toto jediný druh výjimek. Později k nim přibyla možnost používání tříd a jejich instancí. Je-li identifikátorem výjimky třída, nabízí se několik možností. Pokud je argument výjimky instance třídy, která byla použita jako identifikátor, případně instance třídy od ní odvozené, je jako výjimka použita tato instance. Není-li argumentem instance třídy, je z identifikátoru třídy vytvořena nová instance. Při vytváření jsou konstruktoru výjimky předány argumenty, které se odvodí z argumentů předaných konstrukci `raise`. Pro bližší informace vás odkážu do dokumentace jazyka Python. Je-li konstrukci `raise` předána instance třídy, nemohou již být předány další argumenty výjimky.

11.2 Odchycení výjimek

Výjimka se po svém vzniku šíří volajícími funkcemi, dokud není odchycena konstrukcí `try... except`. Při spuštění této konstrukce se nejprve začne vykonávat blok mezi slovy `try` a `except`. Dojde-li v tomto bloku k výjimce, vykonávání se přeruší a začne se porovnávat identifikátor výjimky s tuple identifikátorů uvedených za slovem `except` (je-li identifikátor jen jeden, nemusí se jednat o tuple, můžeme ho zapsat přímo, viz. následující příklad). Je-li nalezena shoda, spustí se blok následující po `except`. Shodou se rozumí:

- je-li identifikátor výjimky řetězec, ke shodě dojde, pokud se shoduje tento řetězec s řetězcem uvedený ze slovem `except`.
- je-li identifikátor výjimky třída, ke shodě dojde, pokud je tato třída odvozená od třídy uvedené za slovem `except` (termín odvozená zahrnuje i rovnost, viz. předminulý díl 10.3, odstavec o funkci `isinstance()`).
- je-li identifikátorem výjimky instance třídy, dojde ke shodě, je-li třída této instance předané odvozená od třídy uvedené za slovem `except`.

Bloků `except` může být uvedeno i více, každý s jiným identifikátorem výjimky (čili pro jiný druh chyby). Porovnávání identifikátorů výjimek se děje popořadě, tak, jak jsou jednotlivé větve `except` napsány ve zdrojovém kódu. Bloky `except` mohou být i pro společné identifikátory výjimky, ale po vykonání jednoho bloku `except` se již neprovádí další porovnávání identifikátorů výjimek a běh programu pokračuje za posledním blokem. Není-li nalezena odpovídající větev `except`, není výjimka odchycena a rozšíří se o úroveň výše v zásobníku volaných funkcí. Pokud není odchycena ani zde, šíří se výše atd., dokud nedojde k jejímu odchycení. Jestliže se ocitneme na vrcholu zásobníku, o obsloužení výjimky se postará interpretr, který vypíše *traceback* a ukončí se.

Seznam identifikátorů za slovem `except` můžeme vynechat, potom tato větev odpovídá jakékoli výjimce. Je-li součástí konstrukce `try...except` více větví `except`, musí být větev odpovídající každé výjimce uvedena jako poslední. Musím ale upozornit, že tato konstrukce je velice hazardním úsekem kódu a je třeba ho používat s velkou obezřetností. Může totiž skrýt některé chyby, o jejichž výskytu nejsme informováni. Vyskytne-li se v takto chráněném kódu třeba chyba dělení nulou, je odchycena konstrukcí `except` a programátor se nic nedozví (pokud není tělo `except` napsáno tak, že vypíše své vlastní hlášení o chybě apod.).

Nyní si již ukážeme příklad použití výjimek:

```
>>> while 1:                                        # (1)
...     x = raw_input("Soubor nebo 'quit': ")       # (2)
...     try:                                        # (3)
...         if x == 'quit':                         # (4)
...             raise 'quit'                        # (5)
...         print open(x).read()                    # (6)
...     except IOError:                             # (7)
...         print 'Chybne jmeno souboru!'           # (8)
...     except 'quit':                              # (9)
...         break                                   # (10)
...
```

Trochu strojená ukázka, ale jako příklad postačí. V nekonečném cyklu si postupně necháme zadat jméno souboru. Na řádku (3) začíná chráněný blok kódu. Zde nejprve otestujeme, zdali proměnná `x` není rovna řetězci 'quit'. Je-li tomu tak, vyvoláme výjimku 'quit', kterou má odchytit konstrukce `except` na řádku (9). Řádek (6) otevře soubor, který zadal uživatel, a vypíše jeho obsah na standardní výstup. Bylo-li zadáno chybné jméno souboru, vznikne ve funkci `open()` výjimka `IOError`, ta je odchycena konstrukcí `except` na řádku (7), a je vypsáno chybové hlášení. Veškeré výjimky, které používají vestavěné moduly a samotný interpretr jsou definovány v modulu `exceptions` a jsou automaticky zavedeny do vestavěného prostoru jmen modulu. Proto k nim můžeme přistupovat přímo uvedením jejich jména bez nutnosti importovat modul `exceptions`.

11.3 Argumenty výjimek

Těm všímavějším z vás jistě neuniklo, že náš program zatím nezískával žádné informace o příčinách výjimky. Příčiny, jak jsme si řekli na začátku dnešního dílu, nám prozradí argumenty výjimky. Argumenty jsou předávány konstrukci `raise`, které s jejich pomocí vytvoří instanci výjimky. Při výskytu výjimky hledá interpretr odpovídající `except` část, která, jak víme, je složena z tuple identifikátorů výjimek. Za tímto tuple může ještě následovat jedna nepovinná část – libovolné jméno proměnné, kterému je výjimka přiřazena, odpovídá-li identifikátoru:

```
>>> while 1:                                        # (1)
...     x = raw_input("Soubor nebo 'quit': ")       # (2)
...     try:                                        # (3)
...         if x == 'quit':                         # (4)
...             raise 'quit'                        # (5)
...         print open(x).read()                    # (6)
```

```
...     except IOError, details:                    # (7)
...         print 'Chybne jmeno souboru:'          # (8)
...         print details.filename                  # (9)
...     except 'quit':                              # (10)
...         break                                   # (11)
...
```

Příklad je obdobný jako výše uvedený. Na řádku (7) ale ke jménu `IOError` přibylo ještě jméno `details`, kterému se v případě výskytu výjimky `IOError` přiřadí výjimka. Jelikož `IOError` je třída, proměnné `details` je přiřazena její instance. Z dokumentace zjistíme, že tato instance obsahuje několik atributů a jedním z nich je i `filename`, což je jméno souboru, jehož se výjimka týká. Každá výjimka může mít množství atributů, záleží na jejím typu (rozuměj její třídě). Pro více informací nahlédněte do dokumentace k Pythonu. Je-li identifikátorem výjimky řetězec, jsou detaily o výjimce rovny argumentům předaným konstrukci `raise`. Nejsou-li tyto argumenty uvedeny, je dosazena hodnota `None`.

Došlo-li k nějaké výjimce, je obvyklým postupem výjimku odchytit a zpracovat. Někdy však programátor chce při výskytu výjimky provést nějakou akci, ale nechce výjimku odchytit. Proto je zde varianta konstrukce `raise`, která bez parametrů vyvolá naposledy nastavenou výjimku. Klasické použití je při odchytávání všech výjimek větví `except` bez parametrů. Při výskytu výjimky ji program odchytí, posléze vytiskne chybové hlášení a výjimku obnoví:

```
>>> while 1:
...     x = raw_input("Soubor nebo 'quit': ")
...     try:
...         if x == 'quit':
...             raise 'quit'
...         print open(x).read()
...     except IOError, details:
...         print 'Chybne jmeno souboru:'
...         print details.filename
...     except 'quit':
...         break
...     except:
...         print 'Doslo k~chybe!'
...         raise
...
```

Přejeme-li si výjimku odchytit, ale nechceme provést žádnou akci při jejím výskytu, můžeme použít konstrukci `pass`, která plní pouze funkci jakési výplně na místech, kde se očekává blok kódu.

Konstrukce `try...except` může mít ještě třetí nepovinnou část, blok `else`, který je vykonán při úspěšném vykonání těla bloku `try`. Výjimky, které vzniknou v části `else`, však již nejsou obslouženy bloky `except` na stejné úrovni.

11.4 Přístup k informacím o výjimce

Je-li použita část `except` bez tuple výjimek, není možné použít cílovou proměnnou, které se přiřadí výjimka. Z toho důvodu Python poskytuje další cestu, jak si zpřístupnit informace

o výjimce. Tyto informace nám zprostředkovává modul `sys` pomocí následujících proměnných:

- `sys.exc_type` – typ výjimky, defacto její identifikátor předaný konstrukci `raise`.
- `sys.exc_value` – argument výjimky
- `sys.exc_traceback` – traceback objekt, který obsahuje informace o volaných funkcích. Přístup k tomuto objektu je možný pomocí modulu `traceback`. Bližší informace viz dokumentace k jazyku.
- `sys.exc_info()` – funkce, která vrací tuple (`sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`)

11.5 Konstrukce `try...finally`

Další variantou klíčového slova try je konstrukce `try...finally`. Ta se používá při definování *cleanup* kódu (kódu, který je spouštěn vždy a používá se např. pro uzavření souborů). Při vykonávání této konstrukce se nejprve spustí blok následující po slově `try`. Při jeho opuštění se vykoná blok za klíčovým slovem `finally`, přičemž nezáleží na způsobu ukončení bloku `try` (může jím být použití konstrukce `return` pro návrat z funkce, nebo vznik výjimky). Klíčové slovo `finally` nelze použít zároveň s `except`. Oba typy chráněných bloků však mohou být do sebe libovolně zanořovány.

11.6 Doplnění

V Pythonu existují výjimky, kterých se týkají určité zvláštnosti. Tou první je `SyntaxError` – syntaktická chyba, pokud se vyskytne na nejvyšší úrovni (tj. v hlavním modulu), nelze ji odchytit. Pokud se však vyskytne v některém z modulů, které importujeme, chová se jako obyčejná výjimka a lze ji odchytit klasickou konstrukcí `try...except`.

Další výjimkou, která má zvláštní význam, je výjimka `KeyboardInterrupt`. Ta může být vyvolána v kterémkoli místě programu stisknutím kombinace kláves Ctrl-C. Chová se jako jakákoli jiná výjimka, a pokud není odchycena, způsobí stisk Ctrl-C ukončení programu.

Poslední výjimkou, kterou si zde uvedeme, je `SystemExit`. Ta se opět chová jako jakákoli jiná výjimka s tím rozdílem, že pokud není odchycena, způsobí ukončení interpretru (i interaktivního), aniž by vytiskla hlášení o chybě.

Kapitola 12

Pokročilá témata

12.1 Skládání a rozklad sekvencí

Co jsou sekvence, jsme si vysvětlili již v prvních dílech seriálu. Každá sekvence má konečný počet prvků. Proměnnými sekvenčními typy jsou seznamy, neměnnými pak tuple. Ke všemu si programátor může vytvořit vlastní třídy, které implementují rozhraní sekvenčních typů (čtení, nastavení a smazání prvku na určitém indexu). Sekvence jsou indexovány vzestupně čísly (první je podobně jako v C/C++ nula). Seznam vytvoříme pomocí výrazu, který se do češtiny nechá volně přeložit jako *konstruktor seznamu* (anglicky „list display“). Jeho nejjednodušší formou je výčet prvků oddělených čárkou a obklopený hranatými závorkami. Další možné formy zápisu si uvedeme později.

Tuple, jak již také víme, vytvoříme jako výčet prvků oddělených čárkou, volitelně uzavřený mezi kulaté závorky. Kulaté závorky můžeme vynechat, používají se v případech, kdy je třeba zabránit dvojznačnostem.

```
>>> 1, 2, 3
(1, 2, 3)
>>> (1, 2, 3)
(1, 2, 3)
>>> 1, 2, 3,
(1, 2, 3)
```

Toto vše jsou tuple o třech prvcích 1, 2 a 3. Vytváření tuple pomocí čárek se také nazývá *tuple packing* (skládání tuple). Opačnou operací je *sequence unpacking* (rozklad sekvencí). Ta využívá toho, že výčet prvků oddělených čárkou nemusí stát jen na pravé straně přiřazení, ale i na jeho levé straně:

```
>>> a, b, c = 1, 2, 3
>>> a, b = b, a
>>> a
2
>>> b
1
```

Toho se dá s výhodou využít pro přiřazení hodnot více proměnným najednou nebo pro záměnu hodnot dvou proměnných bez použití pomocné proměnné. Rozklad sekvencí, jak

již napovídá název, se může aplikovat na libovolnou sekvenci. Na pravé straně přiřazení tedy může stát i seznam. Použití rozkladu sekvencí se neomezuje jen na přiřazení, ale s výhodou ho můžeme použít i například při definici funkce nebo ve větvi `except` konstrukce `try...except`:

```
>>> import math                                    # (1)
>>> def vzdalenost_bodu((x1, y1), (x2, y2)):      # (2)
...     return math.sqrt((x1 - x2)**2 + \          # (3)
...                      (y1 - y2)**2)             # (4)
...                                                # (5)
>>> bod1 = (1, 0)                                 # (6)
>>> bod2 = (4, 4)                                 # (7)
>>> print vzdalenost_bodu(bod1, bod2)             # (8)
5.0                                                # (9)
```

Všimněte si především řádku (2). Funkce `vzdalenost_bodu` přebírá dva argumenty, které musí být dvouprvkové tuple. Tyto argumenty reprezentují souřadnice x a y dvou bodů. Při volání funkce `vzdalenost_bodu` s proměnnými `bod1` a `bod2` jako argumenty dojde právě na řádku (2) k rozkladu těchto tuple na jednoduché proměnné x1, y1, x2 a y2. Je třeba zdůraznit, že výčet prvků na obou stranách přiřazení musí být při aplikování rozkladu sekvencí *stejný*. Toto volání funkce `vzdalenost_bodu` vede k výjimce `ValueError`:

```
>>> bod3 = (1, 2, 3)
>>> print vzdalenost_bodu(bod3, bod2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 1, in vzdalenost_bodu
ValueError: unpack tuple of wrong size
```

12.2 Konstruktor seznamu

S konstruktorem seznamu jsme se již setkali. Byla to jeho nejjednodušší varianta, kterou jste poznali ve druhém dílu seriálu:

```
>>> s1 = [1, 2, 3, 4]                    # (1)
>>> s2 = []                              # (2)
>>> s3 = [2]                             # (3)
```

Jazyk Python ale nabízí i rozšířenou syntax, ve které máme přístupné zjednodušené konstrukce `for` a `if`. Jednoduché převedení tuple na seznam by se dalo provést třeba takto:

```
>>> tuple = (1, 2, 3, 4, 5)
>>> seznam = [i for i in tuple]
>>> seznam
[1, 2, 3, 4, 5]
```

V tomto případě je jako první v hranatých závorkách uveden výraz, jehož hodnoty budou tvořit prvky nově vzniklého seznamu. Tento konstruktor bere postupně prvky jeden za druhým z proměnné `tuple` a dosazuje je za řídící proměnnou `i`. Poté se vyhodnotí výraz před slovem `for` a z hodnot tohoto výrazu se nakonec složí nový seznam. Pro výpočet tabulky druhých mocnin čísel od 1 do 5 můžeme použít následující výraz:

```
>>> [i**2 for i in range(1, 6)]
[1, 4, 9, 16, 25]
```

Konstrukci `for` lze doplnit i o podmínku, kterou můžeme omezit prvky, jež se mají vyskytnout v novém seznamu. Pro více informací vám doporučuji nahlédnout do dokumentace k Pythonu. Seznam sudých čísel od 0 do 10 můžeme vytvořit třeba takto.

```
>>> [i for i in range(11) if i % 2 == 0]
[0, 2, 4, 6, 8, 10]
```

12.3 Anonymní lambda funkce

Pro definování funkcí zná Python, kromě klíčového slova `def`, i klíčové slovo `lambda`, kterým lze vytvořit takzvanou anonymní funkci. Tělem této funkce musí být jediný výraz! Návratová hodnota lambda funkce je rovna hodnotě výrazu. Lambda funkce mohou mít také parametry. Vytvoření nové anonymní funkce je výraz, takže abychom k funkci mohli přistupovat, musíme této funkci přiřadit nějaké proměnné:

```
>>> to_upper = lambda str: str.upper()
>>> print to_upper('python - batteries included')
PYTHON - BATTERIES INCLUDED
```

Lambda funkce jsou pouze jakousi zkratkou a její tělo musí opravdu tvořit jediný výraz. Nelze tedy používat konstrukce jako `if`, `for` nebo `print`. Velice se ale hodí jako argument pro funkce, které vyžadují určitou zpětnou vazbu (příkladem mohou být funkce `os.path.walk()`, `map()` nebo `reduce()` – pro jejich popis nahlédněte prozatím do dokumentace, k jejich výkladu se dostaneme v některém z dalších dílů).

12.4 Formátování řetězců

Jazyk Python obsahuje operátor `%`, který normálně znamená zbytek po dělení (modulo). Pro řetězce, a to jak normální, tak Unicode, má tento operátor jiný význam. Jedná se totiž o operátor formátování řetězce, obdoba C funkce sprintf(). Pro formátování řetězců musíme nejprve sestavit formátovací řetězec, v němž pomocí speciálních sekvencí znaků vyznačíme místa, kam se mají vložit data, která se předají operátoru `%`:

```
>>> 'Dnes je %s, aktualni cas %d:%02d' % ('patek', 15, 2)
'Dnes je patek, aktualni cas 15:02'
```

Pro ty, kteří nevědí o čem je řeč, ve zkratce shrnu podstatu formátování řetězců. Ta spočívá v prohledávání formátovacího řetězce (první operand operátoru `%`) a náhrada každé řídící sekvence jí odpovídajícími daty (druhý operand). Řídící sekvence začíná znakem `%` a následující znak určuje typ konverze, která se má s odpovídajícím prvkem dat provést (např. `%s` znamená převedení aktuálního argumentu na řetězec, blíže viz dokumentace k jazyku). Po zpracování řídící sekvence se začne vyhodnocovat další řídící sekvence, spolu s dalším prvkem dat. Jakýkoli řetězec, který nepatří mezi řídící, je do výstupního řetězce zkopírován

beze změny. Data jsou operátoru % předávána jako tuple, pouze v případě, kdy se jako data použije jediný výraz, je možné ho psát jako singleton.

Formátovací řetězce umožňují specifikovat i šířku jednotlivých polí, přesnost desetinné části, (ne)zobrazování znaménka `+` před kladnými čísly, zarovnání nalevo nebo napravo atd. Bližší popis by vystačil na samostatný díl seriálu. Protože jde ale o techniku všeobecně známou (právě z C/C++), nebudeme se jí dále věnovat. Případné zájemce odkáži do dokumentace k jazyku Python (protože tato stránka není uvedena v indexu dokumentace, hledejte v části Python Library Reference sekci číslo 2.2.6.2).

12.5 Iterátory

V této kapitole se budeme naposledy věnovat syntaktické stránce jazyka Python. V dalších dílech již bude následovat tolik žádaný popis modulů tohoto jazyka. Začněme jednou z vlastností, které přinesl interpret Pythonu verze 2.1 – *iterátory*.

Přidávání nových vlastností do jazyka Python se řídí takzvanými dokumenty [`http://python.sourceforge.net/peps/`] PEP (Python Enhancement Proposals). Každý PEP je jakousi obdobou internetových standardů RFC, každý má své číslo a jejich přijímání se řídí určitými pravidly – nejprve se musí najít někdo, kdo požaduje určité rozšíření jazyka nebo jakoukoli jinou změnu, která se týká jazyka. Aby tato změna dostala svůj vlastní PEP, musí se jednat o závažnější změnu nebo novinku, menší změny lze přímo zasílat jako patche. Poté tento člověk (autor PEPu, v anglické dokumentaci champion) napíše draft standardu, který zašle správci dokumentů PEP. Ten draft prohlédne a pokud dodržuje všechna doporučení, je autorovi přiděleno číslo PEP, draft je uveřejněn a je otevřena diskuse. Poté, když se zdá, že je požadavek dobře specifikován, se přikročí k druhému kroku – implementování této vlastnosti do jazyka samotného. Více informací o tomto pochodu vám nabídne PEP 001.

Stejnou cestou vznikly i iterátory (PEP 234). Od verze 2.1 interpretu již konstrukce `for` neiteruje po jednotlivých prvcích sekvence, ale nejprve požádá tuto sekvenci o iterátor a jednotlivé prvky, které bude dosazovat za řídící proměnnou, mu předává až iterátor. To umožňuje velice jednoduchým způsobem řídit, po kterých prvcích bude konstrukce `for` iterovat.

Iterátor je, jako cokoli jiného v jazyce Python, objekt. Tento objekt podporuje jednu jedinou operaci – vrací další hodnotu. Tuto operaci reprezentuje metoda `next()`, která nepřejímá žádný argument a její návratová hodnota reprezentuje další prvek.

Nejjednodušším způsobem, jak vytvořit iterátor, je vestavěná funkce `iter()`, která má dva tvary:

- `iter(obj)`, která vrátí iterátor příslušející k objektu `obj`
- `iter(callable, sentinel)`, kde `callable` je funkční objekt a `sentinel` je nějaká hodnota. Rovná-li se návratová hodnota funkce `callable` hodnotě `sentinel`, je ukončena iterace. Jako hodnoty jednotlivých iterací se použijí návratové hodnoty funkce `callable`.

Interpretr ale ještě potřebuje rozpoznat, kdy byl předán poslední prvek, a kdy se tudíž má ukončit cyklus `for`. Zavedení nové hodnoty, podobně třeba jako `None`, se neujalo, protože iterátor by měl být schopen vrátit jakoukoli hodnotu. Nakonec byl tento problém vyřešen použitím výjimky `StopIteration`. Tu vyvolá zavolání metody `next()` iterátoru, má-li dojít

k ukončení cyklu. Konstrukce `for` tuto výjimku odchytí a předá řízení za tělo cyklu, případně větvi `else`, pokud existuje.

Funkce `iter()`, které předáme jako objekt sekvenci, vytvoří iterátor vracející postupně všechny prvky sekvence:

```
>>> a = [1, 2, 3]

>>> i = iter(a)
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

Jak je vidět výše – po vrácení všech prvků sekvence vyústí další volání metody `next()` k výjimce `StopIteration`. To však není vše, iterátory lze pomocí funkce `iter()` vytvořit i k asociativním polím – pak iterátor vrací jako hodnoty klíče tohoto asociativního pole. Následující dva iterátory x a y si tedy jsou ekvivalentní:

```
>>> p = {1: 'a', 2: 'b', 3: 'c'}
>>> x = iter(p)
>>> y = iter(p.keys())
```

Iterátor ale lze vytvořit i pro soubory. Na tomto místě je však třeba dát pozor na současnou implementaci, která pracuje bezchybně pro obyčejné soubory, ale třeba pro roury tomu může být jinak – nejsou-li dostupná již žádná data, iterátor ukončí cyklus `for` výjimkou `StopIteration`, po ukončení cyklu ale mohou některá další data dorazit, cyklus je však již ukončen a pokud iterátor jednou vrátí `StopIteration`, má ho vracet pořád. Tím dojde ke ztrátě nově došlých dat. Guido van Rossum ale slibuje nápravu tohoto problému v dalších verzích.

Konstrukce `for` si od verze 2.1 vždy pomocí funkce `iter()` vytvoří nový iterátor pro objekt, jehož prvky má procházet, a pro tento iterátor volá metodu `next()`, jejíž návratové hodnoty dosazuje za řídící proměnnou. Dále je definicí iterátorů zaručeno, že pokud požadujeme iterátor nějakého již existujícího iterátoru, vrátí se ten původní beze změny. Takže následující příklady konstrukce `for` mají ekvivalentní funkci:

```
a = [1, 2, 3, 4, 5]
for i in a:
```

```
    print i
for i in iter(a):
    print i
```

Uživatelská třída může poskytovat svou vlastní metodu, která kromě argumentu self nepřebírá žádné jiné argumenty a která vrací iterátor instance této třídy. Tato metoda, podobně jako jiné metody, které mají co dočinění s „vnitřním chodem“ interpretu, má název `__iter__()` a je volána vždy při vytvoření nového iterátoru funkcí `iter()`.

12.6 Generátory

Mechanismus generátorů je geniálním rozšířením iterátorů. Poprvé byly uvedeny v nejnovější verzi 2.2 interpretu. Protože však generátory používají nové klíčové slovo `yield`, musíme si o jejich používání zatím říci sami – jak, to si povíme později. Nyní si vysvětlíme, co jsou generátory a k čemu se hodí.

Každý programátor jistě zná mechanismus Producent-Konzument. Producent získává data, Konzument je zpracovává. Jako Producent pracuje třeba parser zdrojového kódu, Konzumentem je v tomto případě jeho kompilátor. Producenta s Konzumentem je ale třeba nějakým způsobem propojit.

Nejjednodušším případem Producenta je jednoduchá funkce, která přebírá další argument navíc – zpětnou vazbu, Konzumenta. Napsat takového Producenta není složité, o to složitější je naprogramovat Konzumenta. Konzument totiž často potřebuje uchovávat informace o předchozích stavech. To lze například pomocí globálních proměnných. Problém se stává ještě složitějším používáním více vláken.

Další možnost – Producent vygeneruje všechna data a Konzument je posléze zpracuje. Jednoduché k implementování, ale velice náročné na paměť. Představte si, že Konzument vyhledává ve 100MB XML datech jediný řetězec. Producent tedy parsuje a natáhne do paměti všech sto megabajtů dat, aby Konzument nakonec zjistil, že se v nich hledaný řetězec nenachází.

Jednou z dalších možností je použití více vláken. Producent jedno vlákno, Konzument druhé. Problém se zdá být vyřešen. Python se ale od počátku potýká s vlákny. Veškeré vnitřní struktury Pythonu jsou chráněny jedním zámkem, takže na víceprocesorových strojích stejně běží v jednom čase pouze jedno vlákno. Navíc zde musíme používat zámky nebo jiné nástroje pro synchronizaci vláken. (Více o vláknech v dílu věnovaném modulu `threading`.)

Možností, jak tento problém řešit, je několik, ale Python přišel s elegantním a jednoduchým řešením, kterým jsou generátory. Generátor je speciální druh funkce, která místo klíčového slova `return` používá klíčové slovo `yield`. Nejjednodušší generátor může vypadat třeba takto:

```
>>> from __future__ import generators
>>> def gen():
...     yield 1
...     yield 2
...     yield 3
...
>>> gen()
<generator object at 0x81c5198>
>>> for i in gen():
```

```
...     print i
...
1
2
3
```

Protože generátory vyžadují nové klíčové slovo `yield`, bylo nejprve třeba učinit opatření, aby jeho zavedení nevedlo k chybné funkci existujícího kódu. Slovo `yield` se stane klíčovým slovem až po spuštění příkazu `import` tak, jak ho ukazuje první řádek příkladu. Bez jeho spuštění příklad nebude pracovat. Více o nových vlastnostech a „modulu“ `__future__` se dozvíte z dokumentace k jazyku Python nebo z PEP 236.

Zavoláním funkce `gen()` se nespustí její tělo, ale pouze se vytvoří nový generátor. Každý generátor má stejné rozhraní jako iterátor, takže je možné jej použít namísto iterátoru v konstrukci `for`.

Nejdůležitější na generátorech je to, že uchovávají svůj vnitřní stav. Po prvním zavolání metody next() nějakého generátoru se spustí jeho tělo, které běží, dokud interpret nenarazí na příkaz `yield`. Příkaz `yield` má podobnou funkci jako `return` – přeruší tělo generátoru a metoda `next()` vrátí hodnotu výrazu zapsaného za slovem `yield`. Důležité však je, co se stane dále – generátor nadále uchovává svůj stav, všechny lokální proměnné apod. jsou „zakonzervovány“ uvnitř generátoru.

Dalším zavoláním metody `next()` se generátor „vzbudí“ a jeho běh pokračuje na výrazu následujícím po příkazu `yield`, který způsobil „uspání“. Narazí-li se na další příkaz `yield`, generátor se opět „uspí“ a metoda `next()` vrátí nějakou hodnotu.

Takto celý cyklus pokračuje, dokud se nevykoná celé tělo generátoru, nebo dokud není generátor ukončen příkazem `return`. Pokud v těle generátoru vznikne výjimka, šíří se normálním způsobem všechny třídy výjimek kromě `StopIteration`. Ta má podobný (ne však úplně stejný) účinek jako příkaz `return` – vede k ukončení iterace.

Nakonec si ukážeme příklad trošku složitějšího generátoru, které simuluje házecí kostku a bude demonstrovat uchovávání lokálních proměnných generátoru:

```
>>> from __future__ import generators
>>> from random import choice

>>> def kostka(hodu):
...     cisla = range(1, 7)
...     for i in range(hodu):
...             print "Ziskavam %d. hod" % (i + 1)
...             yield choice(cisla)
...     print "Uz je dohazeno"
...     return
...
>>> k = kostka(3)
>>> for hod in k:
...     print "Hodil jste %d" % hod
...
Ziskavam 1. hod
Hodil jste 1
```

```
Ziskavam 2. hod
Hodil jste 3
Ziskavam 3. hod
Hodil jste 5
Uz je dohazeno
```

Jak vidíme, proměnná `cisla` a proměnná `i` se uchovává mezi jednotlivými stavy generátoru. Slovo `return` na konci generátoru není nutné, jak jsem již řekl; generátor běží, dokud se nevykoná celé tělo generátoru. Při programování generátoru musíte brát zřetel na to, že nevíte, kdy bude tělo příště spuštěno, bude-li vůbec spuštěno.

Kapitola 13

Moduly jazyka Python

13.1 Práce se soubory

Nejprve si probereme práci se soubory na úrovni obsahu. Podobně jako v jiných jazycích je soubor nejprve nutné otevřít a až poté je možné s jeho obsahem manipulovat. Soubor reprezentuje proměnná, pomocí níž se později přistupuje k jeho obsahu. Soubory mohou být otevírány v několika módech. Každý soubor má přiřazen ukazatel, jenž určuje, na kterém místě bude probíhat další čtení nebo zápis, a po ukončení práce se souborem je slušné ho uzavřít.

Všechny tyto postupy určitě znáte z jazyka C. V Pythonu reprezentuje soubor objekt typu „soubor“. Tento objekt získáme zavoláním funkce `file()`, která přejímá jako argument jméno souboru. Je možné jí předat i mód, v jakém má soubor otevřít ('r' pro čtení, 'w' pro zápis, 'b' binární mód, případně jejich kombinace); mód má stejný význam jako jako u standardní C funkce fopen(). Lze také uvést velikost bufferu pro práci se souborem.

```
>>> infile = file('source', 'r')
>>> outfile = file('target', 'w')
>>> while data:
...     data = infile.read(1024)
...     outfile.write(data)
...
>>> infile.close()
>>> outfile.close()
```

Tento příklad jednoduše ukazuje, jak zkopírovat jeden soubor do druhého. V zájmu kompatibility mezi jednotlivými platformami by bylo lepší předpokládat binární soubory a změnit mód přidáním písmena 'b'.

Jak si jistě někteří zkušenější všimli, funkce `file()` je od verze 2.2 upřednostňována před funkcí `open()`, která má stejné rozhraní a tutéž funkci. Objekt typu soubor nabízí několik metod, jsou to především:

- `read()` – metoda, která přečte data ze souboru a vrátí je jako řetězec. Je možné jí předat počet bytů, které má přečíst.
- `write()` – metoda přejímající jeden argument – řetězec, který má zapsat do souboru

- `flush()` – vyprázdní vyrovnávací paměť používanou pro přístup k souboru standardní C knihovnou
- `seek()` – přesune ukazatel aktuální pozice v souboru na zadaný offset
- `tell()` – vrátí offset aktuální pozice ukazatele v souboru od začátku souboru

Veškeré metody, které pracují se soubory při výskytu chyby, vyvolají výjimky `IOError`. Pro více informací můžete použít dokumentační řetězce těchto a dalších metod, které objekty typu soubor nabízejí.

Pro manipulaci se soubory na úrovni adresářů musíme použít modul `os`. Ten obsahuje jeden podmodul `os.path` (doporučuji přečíst zdrojový kód modulu `os`; dozvíte se, jak vytvořit modul, aniž by existoval jako soubor). Modul `os` umožňuje, kromě jiného, pracovat s aktuálním adresářem, získávat seznam souborů, vytvářet, přejmenovávat a mazat adresáře a odkazy.

```
>>> import os
>>> os.listdir('/')
['bin', 'dev', 'etc', 'lib', 'mnt', 'opt', 'tmp', 'var',
'usr', 'boot', 'home', 'proc', 'sbin', 'root']
>>> os.mkdir('/tmp/foo')
>>> os.symlink('/', '/tmp/foo/root')
>>> os.listdir('/tmp/foo/root')
['bin', 'dev', 'etc', 'lib', 'mnt', 'opt', 'tmp', 'var',
'usr', 'boot', 'home', 'proc', 'sbin', 'root']
```

Název většiny funkcí je shodný jako odpovídající příkazy shellu. Navíc Python nabízí funkce `os.makedirs()`, `os.renames()`, `os.removedirs()`, což jsou ekvivalenty funkcí pro práci s adresáři, které ale podle potřeby vytvářejí mezilehlé adresáře. Ukázkou vám může být následující příklad:

```
>>> os.makedirs('/tmp/Python/Tutorial/node1')
>>> os.renames('/tmp/Python/Tutorial/node1',
... '/tmp/python/node1')
>>> os.removedirs('/tmp/python/node1')
```

Modul os.path je určen pro manipulaci s cestami k souborům. Jedná se o platformově nezávislou implementaci, která zaručuje (při správném používání) funkčnost na všech podporovaných platformách (např. provádí konverzi normálních lomítek na zpětná v systémech DOS a Windows).

Modul `os` vytváří mimo jiné jednotné rozhraní ke službám operačního systému. Lze pomocí něho například vytvářet nové procesy (funkce `os.system()`, případně `os.fork()`), čekat na existující procesy (`os.wait()`) nebo pracovat s rourami (`os.popen()`).

Pomocí proměnné `os.environ` se přistupuje k proměnným prostředí - jedná se o obyčejné asociativní pole, které jako svoje klíče používá jména proměnných prostředí a jehož hodnoty odpovídají hodnotám proměnných, třeba obsah proměnné $HOME získáme jako `os.environ['HOME']`.

Všechny funkce modulu `os` v případě chyby vyvolají výjimku `OSError`. Instance této výjimky má atributy `errno` a `strerror`, které reprezentují číslo chyby a její textový popis. Z čísla proměnné zjistíme její popis pomocí funkce `os.strerror()`.

13.2 Modul __builtin__

Prvním modulem, který jsme používali, aniž bychom to věděli, je modul __builtin__. Tento modul reprezentuje builtin (interní) prostor jmen, a proto ho není nutné explicitně importovat příkazem `import`. Nabízí nám množství užitečných funkcí a některé další typy a proměnné.

Mimo jiné nabízí velice užitečné funkce pro práci se sekvencemi – funkce `map()`, `filter()` a `reduce()`. Každá z nich přejímá jako první argument funkci, které je aplikována na jednotlivé prvky sekvencí.

Funkce `map()` přebírá kromě funkce jednu nebo více sekvencí. Poté projde veškeré prvky sekvence a postupně je předává jako argumenty funkci. Nakonec vrátí seznam, který je vytvořen z jednotlivých návratových hodnot funkce. Je-li jí předáno více sekvencí, jsou odpovídající si prvky předávány zároveň jako tuple těchto prvků. Jednoduchým příkladem může být výraz, který vypočte součet odpovídajících si prvků sekvencí:

```
>>> prvni = [2, 5, 1]
>>> druhy = [3, 6, 8]
>>> def secti(a1, a2):
...     return a1 + a2
...
>>> map(secti, prvni, druhy)
[5, 11, 9]
```

Namísto funkce `secti` se častěji používají anonymní lambda funkce. Je-li jedna sekvence kratší než druhá, doplní se na stejnou délku hodnotami `None`. Zvláštním případem může být zápis, kdy funkce je nahrazena hodnotou `None`. Pak je totiž vrácen seznam tuple vytvořených z prvků sekvence.

```
>>> x1 = [1, 1, 2, 3, 5, 8]
>>> x2 = [1, 2, 3, 4, 5, 6]
>>> map(None, x1, x2)
[(1, 1), (1, 2), (2, 3), (3, 4), (5, 5), (8, 6)]
```

Funkce `filter()` prochází jí předanou sekvenci a pro jednotlivé prvky volá funkci, která jí byla předána. Je-li návratová hodnota funkce vyhodnocena jako logická jednička, je odpovídající prvek zahrnut do výsledného seznamu:

```
>>> cisla = range(20)
>>> def sude(x):
...     return not (x % 2)
...
>>> filter(sude, cisla)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Konečně funkce `reduce()` nejprve vezme první dva prvky sekvence a pro ně zavolá funkci, pak pro výsledek funkce a další prvek opět zavolá funkci, a tak pokračuje, dokud nevyčerpá všechny prvky sekvence. Konečnou hodnotu poté vrátí:

```
>>> cisla = range(10)
>>> reduce(lambda A, B: A~+ B, cisla)
45
```

Výsledkem je jediná hodnota. Funkci je možné předat i počáteční hodnotu. Více se o tomto triu funkcí můžete dočíst v dokumentaci modulu `__builtin__`.

Interních funkcí je daleko větší počet, než je možné zde uvést, proto jen telegraficky: pro nalezení minimální (maximální) hodnoty prvku nějaké sekvence je zde funkce `min()` (resp. `max()`), funkce `file()` je rovněž funkcí interní, stejně jako funkce `len()`, `str()`, `dir()`, `iter()`, `abs()` nebo `range()`.

V modulu `__builtin__` najdete i několik hodnot. Předně jsou to hodnoty `None` a `Ellipsis`. Zatímco první znamená „žádný“ objekt (obdoba hodnoty null v C) a má hodnotu logické nuly, druhá se používá u rozšířených slice konstrukcí a my se jí zde zabývat nebudeme. Do interního modulu jsou také zavedena jména všech standardních výjimek, takže je možné je ihned odkazovat bez použití tečkové notace.

13.3 Modul `sys`

Tento modul umožňuje přístup k samotnému jádru interpretru. S jeho pomocí je možné získávat informace o prostředí jazyka, o jeho verzi, nastavovat proměnné důležité pro ladění kódu a činit množství dalších věcí.

13.4 Informace o poslední výjimce

Nejdůležitější z této skupiny funkcí je funkce `sys.exc_info()`, která vrací informace o aktuální obsluhované výjimce. Aktuální obsluhovaná výjimka znamená výjimku, pro kterou již byla spuštěna větev `except` konstrukce `try`. Tato funkce vrátí tuple ve tvaru (typ, hodnota, traceback) – kde typ je typ výjimky, hodnota je její instance (obsahuje tedy argumenty výjimky) a traceback je interní objekt obsahující výpis posledních volaných funkcí. Jestliže neexistuje žádná aktuální výjimka, vrátí se trojice hodnot `None`: (None, None, None). Pomocí této funkce lze získat informace o *poslední* výjimce, nelze takto zjistit nic o žádné z předchozích.

Trojicí proměnných, které se budou velice hodit při ladění zdrojového kódu, jsou proměnné `sys.last_type`, `sys.last_value`, `sys.last_traceback`. Jsou v nich uloženy informace o poslední neobsloužené výjimce, což umožňuje spuštění post-mortem ladění.

13.5 Prostředí a platforma

Interpretr umožňuje získat informace o prostředí, v němž byl spuštěn. Název platformy lze získat jako proměnnou `sys.platform`, pořadí bytů stroje, na němž interpretr běží, obsahuje proměnná `sys.byteorder` (řetězce 'big' nebo 'little'), jméno spustitelného souboru interpretru najdeme v proměnné `sys.executable`.

Interpretr také poskytuje informace o své verzi – více o proměnných `sys.version`, `sys.version_info` a `sys.hexversion` najdete v dokumentaci jazyka Python.

Modul `sys` také nabízí dvě proměnné `sys.ps1` a `sys.ps2`, pomocí nichž se nastavuje primární a sekundární výzva (implicitně '>>>' a '...').

13.6 Implicitní kódování

Jak již víme, Python používá i Unicode řetězce. Aby bylo zřejmé, do jakého kódování je má konvertovat při použití funkce `str()`, je třeba toto kódování nastavit. Provádí to správce systému úpravou modulu `site`, kde volá funkci `sys.setdefaultencoding()`, poté je již tato funkce nepřístupná. Lze pouze získat hodnotu této proměnné pomocí funkce `sys.getdefaultencoding()`.

13.7 Standardní vstupně výstupní proudy

Tyto proudy, které používá interpretr při vstupu a výstupu, můžete najít jako proměnné `sys.stdin`, `sys.stdout`, `sys.stderr`. Umožňují, aby jim programátor přiřadil svůj vlastní objekt typu soubor (nebo libovolný jiný objekt se stejným rozhraním) a všechny příkazy a funkce (`print`, `input()`, `raw_input()`) s nimi budou spolupracovat.

Kapitola 14

Zpracování textu

14.1 Řetězce

Jak již bylo řečeno v jednom z prvních dílů tohoto seriálu, *řetězce* jsou typem neměnným, což znamená, že jejich obsah nelze změnit. Nový řetězec můžeme vytvořit složením z jiných řetězců. Řetězce jsou sekvence, takže pro jejich indexování můžeme použít klasické postupy včetně slice konstrukcí. Řetězce zapisujeme mezi jednoduché, popřípadě dvojité uvozovky. Uvnitř řetězců můžeme používat escape sekvence jako v jazyce C.

Každý řetězec má i několik metod, které umožňují s ním jednoduše pracovat. Pokud metoda nějakým způsobem mění řetězec, vrací vždy novou hodnotu. Mezi tyto metody patří hlavně tyto:

- Metody pro změnu velikosti písmen – `capitalize()`, `lower()`, `upper()`, `title()`
- Metody pro zarovnání řetězců – `ljust()`, `rjust()`, `center()` – přejímají jeden argument – šířku pole, do kterého se má řetězec zarovnat
- Metody pro dotazování řetězce – mezi tyto metody patří především vyhledávací metody `count()`, `find()`, `index()` a `rfind()` a metody, které vrací 1, odpovídá-li řetězec nějaké vlastnosti: `isalnum()`, `isalpha()` apod.
- Metody pro náhradu znaků – `replace()` a `translate()`

Řetězce mají ještě dvě užitečné a často používané metody: `split()` a `join()`. Metoda `split` rozdělí řetězec na sekvenci, přičemž jako oddělovač prvků bere první argument. Nepředáme-li jí žádný argument, použije se jako oddělovač libovolný „bílý“ znak. Metoda `join()` je metodou inverzní, to znamená, že provádí pravý opak. Jako argument přejímá sekvenci a její prvky spojí v řetězec za použití řetězce jako oddělovače:

```
>>> s~= 'Vitejte na serveru root.cz'
>>> s.split()
['Vitejte', 'na', 'serveru', 'root.cz']
>>> s.split('e')
['Vit', 'jt', ' na s', 'rv', 'ru root.cz']
>>> l = ['Python', 'only', 'Python']
>>> ' '.join(l)
```

```
'Python only Python'
>>> ' '.join('Python 2.2')
'P y t h o~n   2 . 2'
```

Na posledním řádku můžete vidět efektní způsob, jak „prostrkat“ řetězec. Takto se třeba mohou generovat elegantní nadpisy pro WWW stránky apod.

14.2 Unicode řetězce

Řetězce v kódování Unicode můžete vytvořit zapsáním písmena `u` před řetězec. Unicode řetězec lze také získat voláním interní funkce `unicode()`. Této funkci můžeme předat libovolný řetězec a kódování, ve kterém je, a Python z něj vytvoří Unicode řetězec. Pokud kódování nezadáme, použije se implicitní kódování tak, jak bylo iniciováno funkcí `sys.setdefaultencoding()`.

```
>>> unicode('ř', 'iso8859-2')
u'\u0159'
>>> unicode('ř', 'iso8859-1')
u'\xf8'
>>> unicode('ř', 'ascii')
```

V prvním případě vytvoříme ze znaku 'ř' (hodnota 0xF8) Unicode řetězec za použití „našeho“ kódování ISO8859-2. Podíváme-li se do tabulky kódů Unicode, zjistíme, že hodnota 0x0159 opravdu odpovídá malému ř. Použijeme-li jiné kódování, dojde k chybné interpretaci znaků (viz druhý příklad, kdy našemu písmenu 'ř' v kódování ISO8859-1 odpovídá písmeno 'ó' s hodnotou 0x00F8, ve třetím případě interpret dokonce odmítl znak 'ř' zkonvertovat, protože kódování ASCII používá pouze hodnoty 0x00 až 0x7F).

Namísto funkce `unicode()` můžeme použít i metodu `decode()` libovolného řetězce. Ta „dekóduje“ data v určitém kódování a vrátí výsledný Unicode řetězec, který odpovídá původnímu řetězci. Metodu `decode()` podporují pouze obyčejné řetězce, u Unicode ztrácí význam, protože data jsou již v univerzálním formátu:

```
>>> 'ř'.decode('iso8859-2')
u'\u0159'
>>> 'příšera'.decode('iso8859-2')
u'p\u0159\xed\u0161era'
```

Pro převedení Unicode, ale i obyčejného řetězce, do nějakého kódování použijeme metodu `encode()`, která je pravým opakem metody `decode()`, vezme data v univerzálním formátu a vrátí je v daném kódování:

```
>>> 'příšera'.encode('utf-7')
'p+AVkA7QFh-era'

>>> 'příšera'.encode('base64')
'cPjtuWVyYQ==\n'
```

Metodu `encode()` používá i funkce `str()` nebo příkaz `print` při tisku Unicode řetězců. V případě, že máte nastaveno implicitní kódování, nemusíte uvádět jméno kódování, rutiny Pythonu automaticky použijí toto kódování. Všechna jména kódování je doporučeno uvádět malými písmeny. Python podporuje mnoho kodeků pro konvertování řetězců, kromě jmenovaných i utf-7, utf-8, utf-16, base64, hex nebo zlib, přičemž můžete dopsat kodeky vlastní.

14.3 Regulární výrazy

Python pro práci s regulárními výrazy nezavádí žádnou novou syntaxi, vše řeší „svojí“ cestou – za použití modulů. Tento modul se jmenuje `re` a obsahuje několik funkcí a konstant, které jsou užitečné pro práci s regulárními výrazy. Modul `re` používá regulární výrazy ve stylu jazyka Perl. Existuje i modul `regex`, který používá výrazy programu Emacs, nicméně tento modul již byl dávno zavržen a jeho používání v nových programech není doporučováno. Kvůli omezenému prostoru si ukážeme pouze jednoduché aplikace regulárních výrazů. Pro více informací si přečtěte Regular Expresion HOWTO, které spolu s dalšími HOWTO jazyka Python najdete na `http://www.python.org/doc/howto`.

První funkcí, se kterou se setkáte, je funkce `re.match()`, která má dva argumenty – regulární výraz a řetězec. Odpovídají-li znaky na začátku řetězce regulárnímu výrazu, vrátí objekt, který umožňuje získat další informace o výsledku, jestliže si neodpovídají, vrátí `None`. Podobnou funkcí, která ovšem prohledává celý řetězec, je `re.search()`.

```
>>> import re

>>> print re.match('a[0-9]+', 'abcd')
None
>>> print re.match('a[0-9]+', 'a01234')
<_sre.SRE_Match object at 0x83626a0>
>>> print re.match('a[0-9]+', 'a0123b')
<_sre.SRE_Match object at 0x83633b8>
```

Výsledný objekt nám umožní získat další informace o výsledku hledání. Přiřaďme tedy výsledek posledního volání funkce `re.match()` nějaké proměnné:

```
>>> m = re.match('a[0-9]+', 'a0123b')
```

Výsledek prohledávání obsahuje některé užitečné informace, především můžeme zjistit, kde hledaný podřetězec začíná a kde končí:

```
>>> m.start()
0
>>> m.end()
5
```

Obrovským přínosem ale je používání skupin. Skupiny v regulárním výrazu vyznačíme párem závorek. Třeba pro vyextrahování uživatelského jména a domény z e-mailové adresy můžeme napsat tento regulární výraz:

```
>>> r = re.match('([a-zA-Z0-9_.]+)@([a-zA-Z0-9_.]+)', 'honza@py.cz')
>>> r.group(1)
'honza'

>>> r.group(2)
'py.cz'
```

Skupina s číslem 0 je celý řetězec, který odpovídá regulárnímu výrazu. Je-li index skupiny mimo platný rozsah, dojde k výjimce `IndexError`. Python umožňuje jednotlivé skupiny pojmenovat pomocí syntaxe (`?P<name>...`):

```
>>> r = re.match('(?P<user>[a-zA-Z0-9_.]+)@(?P<host>[a-zA-Z0-9_.]+)',
...              'honza@py.cz')
>>> r.group('user')
'honza'

>>> r.group('host')
'py.cz'
```

Spolu s indexováním skupin pomocí řetězců můžeme stále používat i indexování celým číslem, čili `r.group(1)` je totéž co `r.group('user')`.

Funkce `re.search()` se neomezuje pouze na začátek řetězce, ale prohledává řetězec celý. Pro nalezení e-mailové adresy můžeme psát:

```
>>> r = re.search(r'([\w.]+@[\w.]+)', 'Adresa honza@py.cz ...')
>>> r.group(1)
'honza@py.cz'
```

Jak vidíme, regulární výraz pracoval správně. Všimněte si především regulárního výrazu, který je zapsán jako raw řetězec. Vyhneme se tak zdvojení zpětných lomítek (první lomítko patří k množině znaků `\w` a druhým bychom museli uvést první, aby interpret escape sekvenci `\w` nenahradil odpovídajícím znakem). Ve výše uvedeném příkladě jsme použili množinu znaků `\w`, v Pythonu však můžete používat i další:

- `\d` odpovídá jakékoli číslici 0–9, čili je to totéž jako `[0-9]`
- `\D` odpovídá jakémukoli znaku kromě 0–9, totéž co `[^0-9]`, obecně jakákoli množina psaná velkým písmenem je doplňkovou množinou k té psané malým písmenem
- `\s` odpovídá bílému znaku, totéž co `[ \t\n\r\f\v]`, existuje i `\S`
- `\w` odpovídá libovolnému alfanumerickému znaku `[a-zA-Z0-9 ]`, doplňkem je `\W`

Funkce `search()` vrátí pouze *první* výskyt hledaného výrazu. Pro zjištění všech výskytů daného regulárního výrazu můžete použít funkci `re.findall()`, která vrátí všechny odpovídající nepřekrývající se podřetězce jako seznam. Obsahuje-li regulární výraz více než jednu skupinu, budou prvky seznamu tuple odpovídající jednotlivým skupinám:

```
>>> re.findall(r'[\w.]+@[\w.]+', 'Adresy: honza@py.cz, redakce@root.cz')
['honza@py.cz', 'redakce@root.cz']
>>> re.findall(r'([\w.]+)@([\w.]+)',
...            'Adresy: honza@py.cz, redakce@root.cz')
[('honza', 'py.cz'), ('redakce', 'root.cz')]
```

Chceme-li provést náhradu výskytu řetězce určeného regulárním výrazem jiným řetězcem, můžete použít funkci `re.sub()`. Ta jako první argument přebírá regulární výraz, druhým argumentem může být buď řetězec, nebo funkce a konečně třetím argumentem je řetězec, na nějž se má náhrada aplikovat.

Je-li druhým argumentem řetězec, pak je pro každý řetězec odpovídající výrazu provedena náhrada: všechny escape-sekvence jako `\n`, `\r` apod. jsou převedeny na odpovídající znaky a dále je provedena náhrada skupin – `\1` se nahradí řetězcem odpovídajícím první skupině. Namísto `\1` můžeme použít i `\g<1>`, rozdíl se projeví u víceciferných čísel: `\72` neznamená skupinu 7 následovanou dvojkou, ale skupinu 72, proto raději používejte `\g<7>2`. Pro vynásobení všech čísel desítkou můžete tedy napsat výraz:

```
>>> re.sub(r'([1-9][0-9]*)', '\g<1>0', '12, 34, 56')
'120, 340, 560'
```

Druhým argumentem může být i funkce, která je pak volána pro každý odpovídající řetězec. Musí mít jediný argument, za nějž se dosadí objekt, který má stejné rozhraní jako objekt získaný funkcí `re.match()` nebo `re.search()`. Náhradou pak je návratová hodnota této funkce.

14.4 Kompilované regulární výrazy

Chceme-li používat jeden regulární výraz na více řetězců, je efektivnější ho nejprve „zkompilovat“. Tuto práci za nás odvede funkce `re.compile()`, které regulární výraz předáme. Funkce provede jeho analýzu a vrátí objekt, jenž provádí efektivní vyhledávání zadaného vzorku. Tento objekt podporuje metody `match()`, `search()`, `sub()`, `findall()` a další, které jsme si ukázali výše.

14.5 Příznaky regulárních výrazů

Funkce `re.match()`, `re.search()` a `re.compile()` mohou přejímat ještě jeden nepovinný argument `flags`, který je bitovým součinem příznaků. Příznaky jsou definovány jako jména v modulu `re`:

- I, IGNORECASE – hledání nebude rozlišovat velikost písmen
- L, LOCALE – množiny `\w`, `\W`, `\b` a `\B` budou odpovídat aktuálnímu locale
- M, MULTILINE – ovlivňuje výrazy `^` a $, ty normálně odpovídají začátku, resp. konci řetězce, s příznakem MULTILINE odpovídají i začátku a konci řádku v řetězci.
- S, DOTALL – výraz `.` normálně odpovídá libovolnému znaku kromě znaku nového řádku, s příznakem DOTALL se bude uvažovat i znak nového řádku

- U, UNICODE – množiny `\w`, `\W`, `\b` a `\B` budou záviset na tabulce znaků Unicode
- X, VERBOSE – umožňuje psaní lépe vypadajících regulárních výrazů

Dnešní povídání nebylo ani zdaleka vyčerpávající. Věřím, že čtenář, kterého dnešní výklad zaujal, si sám nastuduje dokumentaci k modulu re a výše uvedené HOWTO. Rovněž prosím všechny regex-guru, aby tolerovali zápis mých výrazů – ty si nekladou za cíl být dokonalé, jako spíše ukázat možnosti, kterak lze pythonovské regulární výrazy používat.

Kapitola 15

Vlákna

Vlákna jsou prostředkem známým z mnoha operačních systémů. Lze je chápat jako samostatné procesy. Jednotlivá vlákna běží nezávisle na sobě, přičemž ale sdílí společný adresový prostor. Změna jedné proměnné v jednom vlákně se ihned promítne do ostatních vláken. Někdy nám však tato nezávislost může být na překážku. Jedno vlákno může třeba chtít testovat určitou proměnnou na hodnotu "1", proměnnou tedy načte a chce ji otestovat. Než to ale stihne, je přerušeno operačním systémem a v běhu pokračuje jiné vlákno. To mezitím změní danou proměnnou na "0". Pak je přerušeno a řízení se vrátí vláknu prvnímu. To má stále načtenu hodnotu "1", přestože proměnná již má hodnotu "0". Vlákna tudíž potřebují určité prostředky pro synchronizaci. Pomocí těchto prostředků lze zajistit, že k určitým prostředkům (proměnným, souborům atd.) bude mít přístup vždy pouze jedno vlákno.

Nejjednodušším synchronizačním prostředkem je zámek. Zámek má dva stavy (odemčený a zamčený) a umožňuje dvě operace (zamčení a odemčení samozřejmě). Jejich funkci si vysvětlíme na následujícím příkladu. Mějme dvě vlákna A a B, sdílenou proměnnou X a zámek Z. Nyní chce A přistupovat k X. Zamkne proto Z a začne pracovat s proměnnou X. Mezitím k téže proměnné chce přistupovat i vlákno B. Rozhodne se tedy zamknout Z. Ten je však již zamčen vláknem A, vlákno B proto musí čekat, dokud ho A opět neodemkne. Mezitím vlákno A dokončilo operace s X a zámek Z odemklo. Odemčení Z umožnilo vláknu B ho znovu uzamknout. B tak učiní a nadále pracuje výhradně s X. Nakonec zámek Z odemkne a pokračuje v činnosti. Oblasti, která je obklopena operacemi uzamčení a odemčení, se říká „kritická". Zámky zajišťují, že kritická oblast je prováděna vždy jen jedním vláknem a ostatní čekají, až tuto oblast opustí. S tím souvisí další problémy – musíme zabránit „dostihům" a „uváznutí" při používání více prostředků chráněných více zámky. Tato problematika je však již mimo rámec našeho seriálu. Více se o ní dozvíte z odborné literatury (například většina knih o programování zmiňující se o vláknech apod.).

Implementace vláken v Pythonu používá jeden zámek pro všechny interní proměnné interpretru, proto může vždy běžet pouze jediné vlákno interpretru. Na jednoprocesorových strojích je tento problém bezvýznamný (na jediném procesoru se může v jednom okamžiku vykonávat pouze jedna instrukce, tudíž jediné vlákno a jediný program), o to více je ale palčivější na víceprocesorových počítačích, které umožňují opravdový souběžný běh několika programů či vláken. Program v Pythonu používající vlákna pak běží v jednom okamžiku pouze na jednom procesoru a oproti jednoprocesorové konfiguraci není rychlejší ani o píď.

15.1 Modul `thread`

Tento modul nabízí nízkoúrovňová primitiva, pomocí nichž může programátor vytvářet nová vlákna a jednoduché zámky. Lze jej přikompilovat na všech systémech kompatibilních se standardem POSIX, dále pak ve Windows 95 a NT, SGI IRIX a Solaris 2.x.

Všechny funkce, které modul `thread` nabízí, jsou velmi nízkoúrovňové, proto bych vám doporučoval raději používat modul `threading`, který je vrstvou nad modulem `thread` a poskytuje opravdu komfortní rozhraní pro práci s vlákny.

Nejvýznamnější funkcí modulu `thread` je `start_new_thread()`. Té musíme jako první předat funkci, která se stane tělem vlákna. Pak musí následovat seznam pozičních argumentů jako tuple a nakonec lze uvést i nepovinné asociativní pole keyword argumentů. Každé vlákno má přiřazeno vlastní identifikační číslo. Toto číslo mají různá vlákna různé, ale po ukončení jednoho vlákna může být jeho číslo přiřazeno nějakému nově vzniklému vláknu. Hodnotu tohoto čísla získáme funkcí `thread.get_ident()`. Každé vlákno můžeme ukončit voláním buď funkce `sys.exit()`, nebo funkce `thread.exit()`. Stejný efekt má i neodchycená výjimka `SystemExit`.

Modul `thread` nám umožňuje vytvořit i ten nejjednodušší zámek. Slouží k tomu funkce `thread.allocate_lock()`. Ta je volána bez argumentů a vrátí objekt, který reprezentuje zámek. Zámek nám nabízí tři metody: `acquire()`, `release()` a `locked()`. Metoda `acquire()` zámek uzamkne. Pokud je zámek již uzamčen, čeká, dokud nebude odemčen jiným vláknem, a až poté ho uzamkne. Této metodě můžeme předat i počet vteřin, jak dlouho má maximálně čekat, pokud je zámek uzamknut. Uplyne-li tato doba, vrátí hodnotu `false`, jinak vrátí `true`. Metoda `release()` zámek odemkne, přičemž ho následně může uzamknout jiné vlákno čekající na jeho odemčení. Implementace zaručuje, že čeká-li na odemčení více než jedno vlákno, může ho znovu uzamknout vždy pouze jedno jediné.

15.2 Modul `threading`

Pomocí jednoduchého zámku lze implementovat i složitější synchronizační mechanismy jako reentrantní zámky, události, podmínky nebo semafory. Všechny tyto nástroje nám nabízí vysokoúrovňový modul `threading`, který je již celý napsaný v Pythonu a používá nízkoúrovňový céčkový modul `thread`. Všechny funkce tohoto modulu je možné importovat pomocí `from threading import *`.

Nejprve se podíváme na třídu Thread reprezentující vlákna. Od této třídy můžeme bez problémů odvodit svou vlastní třídu a předefinovat tak implicitní chování vlákna. To, jaký kód bude vlákno vykonávat, můžeme ovlivnit dvojím způsobem, buď předáním funkce konstruktoru třídy `Thread`, nebo předefinováním metody `run()` této třídy. V odvozené třídě byste měli přepsat pouze konstruktor `__init__()` a metodu `run()`, ostatní metody by měly zůstat beze změny:

```
>>> import threading

>>> import time
>>> def vlakno2():
...     for i in range(5):
...         print 'vlakno2'
...         time.sleep(0.2)
```

```
...
>>> def vlakno1():
...     for i in range(10):
...         print 'vlakno1'
...         time.sleep(0.1)
...
>>> v1 = threading.Thread(target = vlakno1)
>>> v2 = threading.Thread(target = vlakno2)
>>> v1.start(); v2.start()
```

Předchozí příklad byl jednoduchou ukázkou, jak vytvořit a spustit dvě vlákna. Objekty `v1` a `v2` reprezentují každé jedno vlákno. Objekt vlákna má také několik metod. Především je to metoda `start()`, která spustí tělo vlákna specifikované parametrem `target` jeho konstruktoru nebo metodou `run()`. Při spouštění vláken dávejte pozor na omyl, kdy chybně napíšete `v1.run()` a tělo se spustí v aktuálním (hlavním) vlákně místo vytvoření vlákna nového!

Objekty vláken mají ještě další užitečné metody. Především je to metoda `join()`. Je-li zavolána pro nějaké vlákno, pak vlákno, které ji spustilo, je zablokováno do doby, dokud vlákno, jehož `join()` bylo zavoláno, neskončí svou činnost. Další užitečná metoda je `isAlive()`, která vrátí `true`, pokud vlákno ještě běží.

Následující metoda `setDaemon()` přebírá logickou hodnotu. Je-li tato hodnota `true`, vlákno je klasifikováno jako démon. Program, používající vlákna, skončí, když svoji činnost dokončí všechna vlákna, která nejsou démoni. Démoni jsou pak automaticky ukončeni. Metoda `isDaemon()` vrací hodnotu příznaku nastavenou metodou `setDaemon()`, při vytvoření je vlákno klasifikováno jako normální („ne-démon“). Poslední dvojice metod `getName()` a `setName()` umožňuje jednotlivá vlákna pojmenovat. Vláknu můžeme přiřadit jméno i při jeho vytvoření předáním keyword argumentu '`name`' konstruktoru třídy `Thread`.

Modul `threading` obsahuje, kromě několika tříd, i některé funkce. Především jsou to `activeCount()`, která vrátí počet aktivních vláken, `currentThread()`, která vrací objekt aktuálního vlákna, a `enumerate()` vracející seznam všech aktivních vláken.

15.3 Synchronizační prostředky

Prvním synchronizačním prostředkem je zámek, jenž získáme voláním funkce `Lock()`, která je pouze odkazem na funkci `thread.allocate_lock()` (viz zdrojový kód modulu `threading`). Jednoduchým příkladem použití může být následující ukázka:

```
>>> import threading
>>> import time
>>> zamek = threading.Lock()

>>> def vlakno2():
...     zamek.acquire()
...     for i in range(5):
...         print 'vlakno2'
...         time.sleep(0.2)
...     zamek.release()
...
```

```
>>> def vlakno1():
...     zamek.acquire()
...     for i in range(10):
...         print 'vlakno1'
...         time.sleep(0.1)
...     zamek.release()
...
>>> v1 = threading.Thread(target = vlakno1)
>>> v2 = threading.Thread(target = vlakno2)
>>> v1.start(); v2.start
```

Mírná modifikace předchozího příkladu ukazuje použití zámku. Tato úprava zabrání pomíchání výstupu jednotlivých vláken. Kritickou oblastí v tomto případě jsou cykly pro výstup na obrazovku.

Reentrantní zámek je modifikací nejjednoduššího zámku. Tento zámek může být jedním vláknem uzamčen (a tedy i odemknut) více než jednou. Pokud se ovšem uzamčený zámek pokusí uzamknout jiné vlákno, musí počkat, až bude úplně odemčen. Tento zámek získáme voláním funkce `threading.RLock()`, vrácený objekt podporuje stejné metody jako jednoduchý zámek – `acquire()` a `release()`.

Semafor je nejstarší synchronizační primitivum. Semafor reprezentuje určitou hodnotu. Každé uzamčení tuto hodnotu sníží o jedničku, odemčení hodnotu o jedničku zvýší, přičemž hodnota se nesmí dostat do záporných čísel. Pokusí-li se některé vlákno semafor uzamknout a jeho hodnota je rovna nule, je toto vlákno zablokováno a musí čekat, dokud nějaký zámek semafor neodemkne. Semafor vrací funkce `threading.Semaphore()`, které předáme hodnotu určující počáteční nastavení semaforu. Metody jsou opět stejné jako u zámku. Semafor s počáteční hodnotou nastavenou nastavenou na 1 má téměř stejné chování jako zámek. Existuje jediný rozdíl: odemčení odemčeného zámku vyvolá výjimku, u semaforu pouze zvýší hodnotu o jedničku (tedy na hodnotu 2). Tomuto chování zabraňuje speciální případ semaforu – `BoundedSemaphore`, který vrací stejnojmenná funkce modulu `threading`. Ten pracuje stejně jako obyčejný semafor, má-li ovšem při volání metody `release()` dojít k překročení počáteční hodnoty, dojde k výjimce `ValueError`. Semafory zajišťují, že do kritické oblasti nevstoupí více než `n` vláken, kde `n` je počáteční hodnota semaforu. Bounded semafor dokáže navíc zabránit chybám, které vznikají vícenásobným odemčením semaforu.

Na více nám v dnešním díle této kapitole již nezbyl prostor. Proto pro popis dalších synchronizačních prvků a vlastností implementace vláken v jazyce Python nahlédněte, prosím, do dokumentace k modulům `thread` a `threading`.

Kapitola 16

Perzistentní objekty

16.1 Serializace

Nejeden programátor jistě zažil potřebu ukládat objekty určitého typu například do souboru a posléze je opět z tohoto souboru přečíst zpět. V dobách kralování jazyků Pascal nebo Basic (na nichž většina současných programátorů začínala) si tuto problematiku řešil každý po svém. Většinou ale bylo možné do jednoho souboru ukládat jen jeden typ dat. S příchodem interpretovaných jazyků a novějších postupů se začaly objevovat prostředky, které umožňují ukládat do souboru data různého typu a zpětně je z tohoto souboru přečíst. Při tomto procesu je třeba zajistit jednotný formát dat, čili u dat musí být uveden jejich typ a další informace potřebné pro zrekonstruování objektu. Proces převodu objektu do „jednotného“ formátu se nazývá *serializace*. Serializace se používá velice často. První způsob použití, který napadne téměř každého, může být ukládání dokumentu do souboru (každý objekt v dokumentu se jednoduše převede do jednotného tvaru a v tomto tvaru se objekty postupně zapíší do souboru, odkud je pak aplikace přečte zpět ve stejném pořadí, v jakém je tam zapsala). Serializaci provádí také jakýkoli mechanismus pro vzdálené volání metod (například CORBA, která serializuje argumenty a návratové hodnoty před jejich přenosem mezi aplikacemi). Obecně lze serializovaná data přenést do jiného prostředí (jiný proces klidně i na jiném počítači), kde se z nich obnoví původní objekt. Přenos se může dít za pomoci soketů, rour, souborů, případně dalších mechanismů.

Na závěr je třeba poznamenat, že serializace není totéž co mechanismus perzistentních objektů. Pro to, abychom mohli perzistentní objekty používat, by bylo potřeba vyřešit otázku pojmenování objektů a konkurenčního přístupu k objektům. Nicméně samotné ukládání dat perzistentního objektu by se řešilo za použití serializace. Ve zbytku této kapitoly se tedy podíváme na moduly určené pro serializaci objektů.

16.2 Marshal

Modul `marshal` je nejjednodušším modulem pro serializaci objektů. Jméno marshal je převzato z jazyka Modula-3, který pojmem marshalling nazývá proces konverze dat z interního do externího tvaru. Unmarshaling je pak proces opačný. Modul `marshal` používá především interpret Pythonu pro ukládání zkompilovaných .pyc souborů. Interní formát *není* dokumentovaný, protože verzi od verze interpretu se mění. Nelze proto spoléhat na jeho zpětnou kompatibilitu. Nicméně je nezávislý na platformně, a tudíž soubory zapsané na jednom počítači

můžete používat i na jiném používajícím tutéž verzi Pythonu (z toho vyplývá jedna užitečná vlastnost – .pyc soubory mohou být sdíleny počítači v síti).

Serializaci je možné většinou uplatnit na omezenou množinu datových typů. Modul marshal dokáže ukládat pouze ty typy, které jsou „nezávislé“ na prostředí interpretu. Jsou to tedy `None`, číselné typy, řetězce (včetně Unicode) a tuple, seznamy a asociativní pole, pokud tyto složené typy obsahují pouze typy, které dokáže `marshal` zpracovat. Posledním typem, kvůli kterému tento modul existuje především, je kódový objekt (tj. objekt, který obsahuje spustitelný kód).

Je třeba připomenout, že `marshal` je navržen pouze pro potřeby interpretu a my si ho vysvětlujeme pouze jako jednu z možných variant. Pro plnohodnotné používání tu je modul `pickle`.

Modul `marshal` exportuje čtyři užitečné funkce: `dump()`, `load()`, `dumps()` a `loads()`. Funkce `dump` přejímá dva argumenty, první je objekt, který má být uložen do souboru, a druhý je souborový objekt, kam bude serializovaný objekt uložen. Funkce `load()` zcela překvapivě načte zpět ze souboru, který jí formou objektu předáme jako jediný argument, jeden objekt, jenž pak vrátí. Funkce `dumps()`, která přejímá jediný argument – objekt k serializaci, vrátí takový řetězec, jaký by byl zapsán do souboru funkcí `dump()`. A nakonec `loads()` zrekonstruuje z řetězce, který jí předáme jako argument, původní objekt:

```
>>> import marshal
>>> obj1 = (None, 1, 2, 3)
>>> obj2 = ['jedna', 'dva', 'atd.']
>>> f = open('/tmp/marshal', 'wb')
>>> marshal.dump(obj1, f)

>>> marshal.dump(obj2, f)
>>> f.close()
```

Tento příklad ukazuje, jak uložit objekty do souboru /tmp/marshal. Ten je otevřen pro zápis a v binárním módu. Unixy nerozlišují mezi binárním a textovým módem, ve Woknech byste si ale s textovým módem ani neškrtli. Nyní si tyto objekty zrekonstruujeme (nevěřící Tomášové si mohou znovu spustit interpret):

```
>>> import marshal
>>> f = open('/tmp/marshal', 'rb')
>>> obj1 = marshal.load(f)
>>> obj2 = marshal.load(f)

>>> f.close()
>>> print obj1
(None, 1, 2, 3)
>>> print obj2
['jedna', 'dva', 'atd.']
```

16.3 Pickle

Jak již bylo řečeno v předchozích odstavcích, `marshal` je modul určený interpretu, proto zároveň s ním existuje ještě jeden modul – `pickle`. Ten je napsán v Pythonu a umožňuje

mnohem lépe řídit proces serializace objektů. Tento objekt již zaručuje zpětnou kompatibilitu a dokáže serializovat i instance uživatelských tříd a dokonce i třídy a funkce samotné.

Abychom předešli omylům – serializování tříd a funkcí neukládá žádný kód, modul pickle si pouze uloží jméno modulu a jméno třídy. Při načtení pak vrátí třídu (funkci) pouze podle těchto informací, přičemž nezáleží na tom, jestli jde stále o jednu a tu samou třídu. Neukládání kódu je bezpečnostní opatření, modul `pickle` se takto (zčásti) vyhnul nebezpečí zneužití pomocí trojských koní.

Modul `pickle` při vícenásobné serializaci jednoho objektu pouze uloží odkaz na posledně serializovaný objekt. Proto je třeba dávat pozor na změny objektů mezi jednotlivými serializacemi, pokud k nim dojde, uloží modul pickle do souboru (řetězce) odkaz na nezměněný objekt a změna nebude zaznamenána. Nicméně i toto chování lze obejít (viz dokumentace jazyka).

Základní rozhraní modulu `pickle` je shodné s modulem `marshal`. Opět zde najdeme čtveřici funkcí `dump()`, `load()`, `dumps()`, `loads()`. Navíc zde najdeme funkce `Pickler()` a `Unpickler()`, které přejímají argument typu souborový objekt a vrátí objekt, který podporuje metody `dump()`, resp. `load()`. Tímto můžeme zkrátit zápis, chceme-li do jednoho souboru uložit více objektů:

```
>>> import pickle
>>> obj1 = (None, 1, 2, 3)
>>> obj2 = ['jedna', 'dva', 'atd.']
>>> f = open('/tmp/pickle', 'wb')
>>> p = pickle.Pickler(f)
>>> p.dump(obj1)
>>> p.dump(obj2)

>>> f.close()

>>> import pickle
>>> f = open('/tmp/pickle', 'rb')
>>> u~= pickle.Unpickler(f)
>>> obj1 = u.load()
>>> obj2 = u.load()
>>> f.close()
>>> print obj1
(None, 1, 2, 3)
>>> print obj2
['jedna', 'dva', 'atd.']
```

Nyní je ten pravý čas zmínit se ještě o jedné důležité věci: existují dva moduly umožňující používat `pickle` rozhraní – Pythonovský modul `pickle` a céčkový modul `cPickle`. Modul `cPickle` je mnohonásobně rychlejší, ale nelze jej rozšiřovat, zatímco v modulu `pickle` jsou funkce `Pickler()` a `Unpickler()` implementovány jako třídy a není problém od nich odvodit vlastní třídu. Nicméně výsledná data po serializaci objektu mají u obou modulů stejný formát, takže není problém moduly zaměňovat (samozřejmě pokud neupravujeme chování při serializaci).

Nyní si ukážeme, jak postupovat, pokud potřebujeme serializovat instanci vlastní třídy.

Nejprve je třeba vědět, že modul `pickle` při obnově instance třídy standardně *nevolá* konstruktor této třídy (jen pro zajímavost: modul při obnově nejprve vytvoří instanci třídy, která nemá konstruktor, a poté změní třídu této instance na danou třídu přiřazením hodnoty atributu `__class__`).

Přeje-li si programátor před obnovením instance zavolat konstruktor její třídy, musí instance podporovat metodu `__getinitargs__()`, která musí vrátit tuple prvků, jež budou použity jako argumenty předané konstruktoru při obnově třídy. Metoda `__getinitargs__()` je volána *před* serializací instance a jí vrácené argumenty jsou uloženy zároveň se serializovanými daty instance.

Další užitečné metody, které může instance implementovat, je dvojice `__getstate__()` a `__setstate__()`. První je volána při serializaci instance a vrací objekt, obsahující interní stav instance. Tento objekt je následně serializován. Při obnově instance je tento objekt obnoven a předán metodě `__setstate__()`, která podle něj obnoví vnitřní stav instance. Metoda `__setstate__()` může být vynechána, pak `__getstate__()` musí vrátit asociativní pole, podle nějž se obnoví atribut `__dict__` této instance.

Nepodporuje-li instance ani metodu `__getinitargs__()`, ani `__getstate__()`, bude se serializovat pouze obsah atributu `__dict__`. Zde si ukážeme jednoduchou třídu, která implementuje frontu FIFO a zároveň podporuje serializaci, přičemž vnitřní stav beze změny uloží jako seznam prvků (protože první prvek je na posledním místě, musí se při naplnění fronty po obnovení obrátit pořadí prvků v seznamu), délka fronty se zase uchovává jako argument pro konstruktor a vrací ho metoda `__getinitargs__`:

```
class Fronta:
    __safe_for_unpickling__ = 1

    def __init__(self, delka):
        print 'Vytvarim Frontu o~delce %d' % delka
        self.delka = delka
        self.buffer = []

    def push(self, prvek):
        print 'Ukladam prvek "%s"' % prvek
        if len(self.buffer) < self.delka:
            self.buffer.insert(0, prvek)
        else:
            raise RuntimeError, 'buffer overflow'

    def pop(self):
        if len(self.buffer) > 0:
            return self.buffer.pop()
        else:
            raise RuntimeError, 'buffer underflow'

    def __getinitargs__(self):
        return (self.delka, )

    def __getstate__(self):
```

```
        return self.buffer

    def __setstate__(self, stav):
        print 'Naplnuji frontu'
        stav.reverse()
        for prvek in stav:
            self.push(prvek)
```

Nyní již můžeme vytvořit instanci této třídy a uložíme jí za pomoci serializace do řetězce:

```
>>> fronta1 = Fronta(5)
Vytvarim Frontu o~delce 5
>>> fronta1.push(1)
Ukladam prvek "1"
>>> fronta1.push('Ahoj')
Ukladam prvek "Ahoj"
>>> import pickle
>>> retezec = pickle.dumps(fronta1)
>>> retezec
"(I5\ni__main__\nFronta\np1\n(lp2\nS'Ahoj'\np3\naI1\nab."

>>> fronta2 = pickle.loads(retezec)
Vytvarim Frontu o~delce 5
Naplnuji frontu
Ukladam prvek "1"
Ukladam prvek "Ahoj"
```

Jak vidíme z následujících řádků, obsah bufferů obou front je stejný!

```
>>> fronta1.buffer
['Ahoj', 1]
>>> fronta2.buffer
['Ahoj', 1]
```

Modul pickle byl navržen i s ohledem na bezpečnost. Proto nelze přímo obnovit instanci libovolné třídy. Třídy, o kterých víme, že je budeme serializovat a poté obnovovat, musíme nejprve „označit“ pomocí atributu `__safe_for_unpickling__` (který musí být pravdivou logickou hodnotou).

16.4 Shelve

Modul `shelve` je mezistupněm mezi serializací a opravdovými perzistentními objekty. Řeší totiž otázku pojmenování objektů (konkurenční přístup ale stále není možný, konkrétně – vícenásobné čtení je podporováno, čtení a zápis zároveň ale neprojde).

Modul `shelve` používá pro ukládání `dbm` databázi, která se použije, je závislé na platformně a dostupných modulech. Modul `shelve` obsahuje funkci `open()`, která přejímá jeden argument – jméno souboru pro `dbm` databázi (tedy bez přípony) – a vrátí objekt, který má

podobné rozhraní jako asociativního pole. Jako klíče se mohou ovšem používat pouze řetězce, hodnotami pak mohou být libovolné objekty, které se podle potřeby serializují a obnovují zpět (pomocí modulu `pickle`). Nakonec se databáze uzavře voláním metody `close()`:

```
>>> import shelve
>>> d = shelve.open('/tmp/shelve')
>>> d['cislo'] = 1
>>> d['retezec'] = 'AHOJ'
>>> d['seznam'] = ['jedna', 1, 1.0]
>>> d.close()
```

Nyní můžeme interpret uzavřít a spustit znova:

```
>>> import shelve
>>> d = shelve.open('/tmp/shelve')
>>> d['cislo']
1
>>> d['retezec']
'AHOJ'
>>> d['seznam']
['jedna', 1, 1.0]
```

Stejně tak můžete používat i další metody dbm databází jako `keys()`, `get()` nebo `has_key()`. Pro více informací o všech v této kapitole probíraných modulech prosím nahlédněte do dokumentace jazyka Python.

Kapitola 17

Komunikace mezi procesy

17.1 Roury

Jak již řekl náš nadpis, *roury* jsou klasikou mezi UNIXy. Ještě než si řekneme, co to roury jsou, musíme si vysvětlit, jakými způsoby lze přistupovat k libovolnému souboru. Pro programátory v jazyce C jsou samozřejmostí dva různé přístupy k souborům. Pomocí přímých volání operačního systému a pomocí standardní C knihovny.

Přístup pomocí volání operačního systému je charakteristický používáním deskriptorů souborů. Deskriptor souboru je malé celé číslo, kterým operační systém reprezentuje otevřený soubor. Standardní C knihovna je vrstva vybudovaná nad operačním systémem a programátorovi nabízí luxusnější přístup k souborům. Automaticky se stará například o bufferování dat. Z hlediska programu se již nejedná o číslo, ale o strukturu, obsahující veškeré důležité informace o souboru.

V Pythonu namísto souborů standardní C knihovny používáme souborové objekty, které získáme voláním interní funkce `file()`, jak jsme si ukázali v dílu 13.1. U těchto souborových objektů můžeme přímo ovlivnit i velikost bufferu, který si C knihovna vytvoří pro přístup k souboru, mód souboru apod. Dále souborové objekty nabízejí i funkci `fileno()`, která vrátí číslo deskriptoru souboru.

Přímý přístup pomocí deskriptorů souborů je realizován funkcemi modulu `os`: `open()`, `close()`, `read()`, `write()`, `dup()` a dalšími. Jednoduchá ukázka použití:

```
>>> import os
>>> fd = os.open('/bin/bash', os.O_RDONLY)
>>> fd
3
>>> os.read(fd, 10)   # přečteme deset bytů
'\x7fELF\x01\x01\x01\x00\x00\x00'
>>> os.close(fd)
```

Protože Python je jazyk objektový, je zvykem na vše používat objektová rozhraní, a proto, pokud to vyloženě nevyžadujete, se nedoporučuje tyto nízkoúrovňové funkce používat. Přesto se jim nejspíše nevyhnete. Existuje však funkce, která dokáže z deskriptoru souboru vytvořit souborový objekt (obdobná funkce existuje i ve standardní C knihovně). Touto funkcí je `os.fdopen()`, které jako první argument předáme číslo deskriptoru souboru a další dva volitelné parametry mají stejný význam jako mód a velikost bufferu u interní funkce `file()`.

Nyní již k rourám. Rouru získáme voláním funkce `os.pipe()`, která vrátí tuple dvou hodnot. Tyto hodnoty jsou deskriptory souborů, které jsou zvláštním způsobem propojené – cokoli zapíšete do druhého deskriptoru, přečtete zpět z deskriptoru prvního. Toto zdánlivě obyčejné chování pochopíte až ve spojení s funkcí `os.fork()`, která „zdvojí“ aktuální proces. Od zavolání této funkce se jedná o dva samostatné procesy, přičemž potomek zdědí od svého rodiče všechny otevřené deskriptory souborů. Oba procesy se navzájem „poznají“ podle návratové hodnoty `os.fork()`, v potomkovi volání této funkce vrátí 0, v rodičovském procesu vrátí PID (číslo procesu) potomka. Pomocí rour se tedy dá realizovat komunikace mezi rodičem a potomkem:

```
#!/usr/bin/env python
import os
rfd, wfd = os.pipe()   # vytvoříme rouru

pid = os.fork()
if pid == 0:
   # jsme potomek
   data = os.read(rfd, 100)
   print 'Potomek: přečetl jsem %d bytů: "%s"' % (len(data), data)
   os.close(rfd)
else:
   # jsme rodič
   data = 'Ahoj, jak se máte?'
   bytu = os.write(wfd, data)
   print 'Rodič: zapsal jsem %d bytů' % bytu
   os.close(wfd)
```

Tento nástin mechanismu rour berte pouze jako informativní, pro detaily k této problematice doporučuji libovolnou knihu o programování pod POSIXem (třeba kniha nakladatelství Computer Press „Linux: Začínáme programovat“, kde se dočtete o rourách a další dnešní látce – soketech, a nejen o nich).

17.2 Sokety

Jak jste si jistě všimli, roury jsou prostředkem pro komunikaci mezi procesy na tomtéž počítači. Logickým rozšířením rour vznikly *sokety*, umožňující komunikaci mezi procesy na různých počítačích v síti. Podobně jako u rour, kde je přesně rozlišeno, který konec roury je pro čtení a který pro zápis, sokety rozlišují mezi klientem a serverem (i když samotné spojení je obousměrné). Typicky je server program, obsluhující klienty a poskytující jim nějaké služby. Většinou jeden server obsluhuje více klientů.

Server nejprve vytvoří soket, poté tento soket pojmenuje (čímž se stane přístupným pro ostatní procesy), následně vytvoří frontu pro příchozí spojení a začne čekat na klienty. Připojí-li se k serveru klient, server si jeho příchozí připojení převezme a pouze pro něj vytvoří nový soket, zatímco původní může nadále vyčkávat na další klienty. Nebudeme zabíhat příliš do detailů, a proto si ihned ukážeme, kterak vytvořit jednoduchý server:

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

```
import socket
HOST = ''
PORT = 50007
s~= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(5)
print 'server ceka na pripojeni'
while 1:
        conn, addr = s.accept()
        print 'pripojen klient:', addr
        while 1:
                data = conn.recv(1024)
                if not data: break
                conn.send(' '.join(data))
        conn.close()
```

Jak vidíme, soketové služby poskytuje modul `socket`. Ten obsahuje několik funkcí, které odpovídají jejich protějškům nabízeným standardní C knihovnou. Jak již řekl minulý odstavec, nejprve soket vytvoříme – to za nás udělá funkce `socket()` modulu `socket`, které předáme dva argumenty – rodinu protokolu a jeho typ. Dostupné rodiny protokolů jsou `AF_INET` (sítě IPv4) a `AF_INET6` (sítě IPv6). Na Unixech můžeme ještě použít `AF_UNIX`, což je lokální soket reprezentovaný souborem v souborovém systému. Typem soketu zase určíme, máme-li zájem o spojitou, nebo nespojitou cestu (v případě rodiny `AF_INET` je spojitá cesta implementována protokolem TCP, který zajišťuje spojitý „bezchybný“ kanál, zatímco nespojitá cesta využívá protokolu UDP). Spojitý typ soketu určíme konstantou `SOCK_STREAM`, nespojitý `SOCK_DGRAM`.

Proběhla-li funkce `socket()` v pořádku, vrátí soketový objekt, který se používá pro další komunikaci se soketem. V případě serveru tedy nejprve soket musíme pojmenovat. S tím nám pomůže metoda `bind()` soketového objektu. Metoda `bind` přejímá jako jeden argument adresu, jejíž datový typ se liší podle zvolené rodiny protokolu. Pro rodinu `AF_INET` jde o tuple o dvou prvcích, první je síťové jméno počítače (případně IP adresa) a druhý pak číslo portu, oba prvky určuje jméno, pod nímž bude náš server naslouchat.

Poněvadž vytváříme server, musíme vytvořit frontu, do níž se budou postupně řadit neobsloužená spojení. Proto zavoláme metodu `listen()` soketu, jíž předáme celé číslo reprezentující délku fronty.

Pak již můžeme čekat na příchozí připojení od klientů. Toto čekání zajistí metoda `accept()`, která převezme prvního klienta ve frontě a vrátí tuple o dvou prvcích – první prvek je soket určený pro komunikaci *pouze* s tímto klientem, druhý pak informace o připojeném klientovi (ve stejném tvaru, jako jsme ji předali metodě `bind()`). Jestliže se žádný klient ještě nepřipojil, metoda `accept()` se zablokuje až do příchodu první žádosti o připojení. Je třeba upozornit, že serverový soket vzniklý funkcí `socket()` reprezentuje pouze jakousi ústřednu, ze které jsou potom získány sokety pro jednotlivé klienty, do serverového soketu tudíž nemůžeme zapisovat a rovněž z něj nelze číst.

Jestliže jsme již získali soket pro komunikaci s klientem, můžeme od něj číst data pomocí metody `recv()`, které předáme počet bytů, jež se má přečíst. Jestliže tato data ještě nedorazila, volání se zablokuje do doby, dokud data nedorazí nebo dokud nebude soket uzavřen, načež jsou data vrácena jako řetězec. Stejně tak můžeme data odesílat pomocí metody `send()`, které naopak předáme řetězec reprezentující data, která se mají do soketu zapsat a

která tudíž bude moci přečíst proces na druhém konci spojení. Uložme tedy kód serveru do souboru dejme tomu 'server.py' a spusťme ho pomocí příkazu `python server.py`.

Tento server čeká na klienta, když se k němu připojí, převezme od něj data, prostrká je mezerami a nakonec je vrátí zpět klientovi. Nyní pro tento server napíšeme klienta:

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

HOST = ''
PORT = 50007
s~= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
print 'posilam data serveru'
s.send('Python umi vsechno')
data = s.recv(1024)
s.close()
print 'prijal jsem data od serveru:', `data`
```

Na straně klienta je práce viditelně jednodušší. Klient opět vytvoří soket a následně ho připojí k serveru (resp. k jeho pojmenovanému soketu). Toto připojení se uloží do fronty, odkud si ho server může vyzvednout a obsloužit. Připojení vykoná metoda `connect()`, která přejímá stejné parametry jako metoda `bind()`. Poté, co je soket připojen ke svému protějšku, můžeme posílat a přijímat data standardními metodami `recv()` a `send()` tak, jak jsme si je popsali dříve.

Náš klient se připojí k serveru, přičemž použije adresu a číslo portu stejné jako server (prázdný řetězec znamená totéž co 'localhost', tedy rozhraní loopback lokálního počítače). Po připojení mu pošle nějaká data (v našem případě řetězec 'Python umi vsechno') a přečte si zpět odpověď. Nakonec se od serveru odpojí a vytiskne odpověď serveru.

Jak jste si jistě všimli, rozhraní soketového objektu je jiné než rozhraní standardního souborového objektu. Proto každý soketový objekt nabízí metodu `makefile()`, která přebírá dva argumenty – mód a velikost bufferu – a vytvoří z nich souborový objekt asociovaný se soketem.

V případě, že se rozhodnete používat Unixové souborové sokety, použijte rodinu `AF_UNIX` a metodám `bind()` a `connect()` předejte ne internetovou adresu, ale cestu v souborovém systému k souboru, který bude reprezentovat tento soket.

17.3 Vlastnosti deskriptorů

V dnešním povídání o rourách a soketech několikrát padlo slovo *blokování*. Blokování můžeme vypnout u soketových objektů pomocí metody `setblocking()`, která přejímá jeden argument, a to příznak, zda při volání metod typu `recv()`, případně `accept()`, a i dalších, má dojít k blokování, nebo ne. U rour si musíme vypomoci modulem `fnctl`, který obsahuje funkce pro nastavování vlastností deskriptorů. Pro bližší seznámení s tímto modulem nahlédněte do dokumentace jazyka Python.

Kapitola 18

Práce s internetovými protokoly

18.1 Protokol SMTP

Jak jistě mnozí z vás ví, je protokol SMTP (Simple Mail Transfer Protocol, viz. [`http://www.faqs.org/rfcs/rfc821.html`] RFC 821) navržen pro přenos elektronické pošty. Tato pošta se sestává z pouhého prostého textu. Pro přenos strukturovaných e-mailů (tj. e-mailů, které se skládají z více částí, např. text a přiložené obrázky apod.) se musíme držet standardu [`http://www.faqs.org/rfcs/rfc2822.html`] RFC 2822. Práci s takto formátovanými zprávami se zabývá modul `email`. S jeho pomocí můžeme zkonstruovat strukturovaný e-mail z jeho jednotlivých částí. Následně si od něj vyžádáme tento e-mail jako prostý text, který odešleme pomocí protokolu SMTP. Klientská část protokolu SMTP je v Pythonu implementována v modulu `smtplib`, jenž si dnes popíšeme. Dokumentaci modulu email si můžete nastudovat jako malý domácí úkol.

Samotný modul `smtplib` toho příliš neobsahuje. Kromě několika výjimek nabízí pouze třídu `SMTP`, která ale sdružuje veškerou funkcionalitu. Pomocí jejích metod se můžeme připojit k SMTP serveru, odeslat e-mail a odpojit se. Zde je třeba podotknout, že pomocí modulu smtplib je možné implementovat pouze klientskou část protokolu SMTP. Následně si popíšeme třídu `SMTP` a její metody.

Veškerá komunikace s SMTP serverem začne vytvořením nové instance třídy `SMTP`, přičemž konstruktoru předáme jako argument jméno hostitele a číslo portu, kam se má tato instance připojit. Číslo portu můžeme vynechat (pak se použije 25) nebo může obsahovat řetězec hostitele (např. "smtp.nekde.com:8989"). Dokonce je možné vynechat i jméno hostitele, pak se vytvoří nepřipojená instance třídy `SMTP` (před jejím používáním je třeba se nejprve připojit k serveru pomocí metody `connect()`, která akceptuje stejné argumenty jako konstruktor). Nepodaří-li se navázat spojení s tímto serverem, dojde k výjimce `smtplib.SMTPConnectError`.

Následuje odeslání e-mailu pomocí metody `sendmail()`. Ta má tři povinné argumenty: první je adresa odesílatele e-mailu, druhý pak adresa (případně adresy) adresáta e-mailu a poslední je vlastní tělo e-mailu včetně hlaviček. Adresy předané metodě `sendmail()` slouží pouze pro vytvoření „obálky“ e-mailu, modul `smtplib` nijak neovlivňuje obsah samotného těla ani hlaviček. Pokud při volání metody `sendmail()` nedojde k výjimce, máme jistotu, že alespoň jednomu adresátovi se e-mail podařilo odeslat. Adresy těch, jimž se zprávu nepodařilo doručit, vrátí metoda `sendmail()` (jako asociativní pole, klíče jsou adresy adresátů, hodnoty pak tuple (chybový_kód, popis_chyby)). Při vzniku výjimky `SMTPRecipientsRefused` se e-mail nepodařilo doručit nikomu z adresátů (přičemž atribut `recipients` objektu výjimky

obsahuje asociativní pole ve tvaru popsaném výše). Pro popis dalších výjimek vás odkáži do dokumentace modulu `smtplib`.

Pro ukončení komunikace s SMTP serverem je zapotřebí zavolat metodu `quit()`. Ta ukončí sezení a uzavře komunikační kanál k SMTP serveru. Po jejím zavolání již není možné používat metody této instance.

Nakonec si ukážeme, jak by mohl vypadat jednoduchý program pro odesílání e-mailů:

```
import smtplib
import sys

od = raw_input('Od: ')
pro = raw_input('Pro: ')
predmet = raw_input('Předmet: ')

msg = ("From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n"
           % (od, pro, predmet))

print "Zadejte text zprávy, ukončete stiskem ^D na samotném řádku:"

msg += sys.stdin.read()

server = smtplib.SMTP('localhost')
server.sendmail(od, pro, msg)
server.quit()
```

18.2 Protokol FTP

Pro potřebu přenosu souborů po síti Internet byl navržen protokol FTP. Již v základní distribuci Pythonu je obsažen modul implementující klientskou část tohoto protokolu – `ftplib`. Tento modul má obdobnou strukturu jako výše uvedený modul `smtplib`, čili základní modul obsahuje třídu `FTP`, která slouží pro práci s protokolem FTP, a několik výjimek.

Třídě `FTP` se budeme věnovat pouze stručně, protože případný zájemce si další najde v dokumentaci modulu `ftplib`. Třídě `FTP` můžeme při jejím vzniku (stejně jako u třídy SMTP) předat adresu počítače, na který se má připojit. Předáme-li jí tuto adresu, je zavolána metoda `connect()` této instance s adresou předanou jako jediný parametr (v případě, že adresu nepředáme, musíme metodu `connect()` volat sami). Dále je možné konstruktoru předat další dva parametry – uživatelské jméno a heslo na vzdáleném počítači, přičemž konstruktor zavolá metodu `login()`, jíž tyto dva argumenty předá (obdobně – pokud tyto argumenty nepředáme konstruktoru, musíme sami zavolat metodu `login()`).

Specifickou vlastností protokolu FTP je používání dvou různých módů – binárního a textového. Modul `ftplib` samozřejmě tyto dva módy podporuje (příkladem může být dvojice metod `retrbinary()` a `retrlines()`).

Po připojení na vzdálený počítač můžeme používat metody pro práci s jednotlivými službami protokolu FTP. Následuje stručný výčet těchto metod (pro kompletní dokumentaci nahlédněte prosím do dokumentace modulu `ftplib` případně přímo do jeho zdrojového kódu):

- `sendcmd(příkaz)` – zašle serveru příkaz a vrátí odpověď serveru jako jediný řetězec (obdobně existuje i `voidcmd()`, který v případě chyby vyvolá výjimku)

- `retrlines(příkaz, funkce)` – zašle serveru příkaz (téměř vždy jím je řetězec 'RETR jméno_souboru', který slouží pro stáhnutí souboru) a začne stahovat odpověď na příkaz v textovém módu, přičemž pro každý řádek zavolá funkci, které předá jako jediný argument tuto řádku. Funkce je nepovinný argument, je-li vynechána, vytiskne se obsah souboru na standardní výstup.

- `retrbinary(příkaz, funkce)` – obdobně jako metoda `retrlines()` zašle serveru příkaz (opět řetězec 'RETR jméno_souboru') a odpověď začne stahovat v binárním módu, přičemž pro každý blok binárních dat zavolá funkci.

- `storlines(příkaz, soubor)` – zašle serveru příkaz (většinou jím je řetězec ve tvaru 'STOR jméno_souboru'), přičemž čte řádky ze souboru (pomocí metody `readlines()`) a zapisuje je do souboru na vzdáleném počítači, pro komunikaci používá textový mód.

- `storbinary(příkaz, soubor)` – pracuje obdobně jako metoda `storlines()`, přenos ale pracuje v binárním módu. Pro čtení souboru se používá metoda `read()`, zároveň je možné použít třetí nepovinný argument, který určuje velikost bloku.

- `dir(adresář, funkce)` – zašle FTP serveru příkaz 'LIST adresář' a výsledek přečte v textovém módu, význam funkce je stejný jako u metody `retrlines()`. Je-li funkce vynechána, je výsledek vytisknut na standardní výstup.

- `quit()` – zašle FTP server příkaz 'QUIT', čímž ukončí spojení s tímto serverem. Po zavolání `quit()` již není možné další metody této instance používat.

Existují i další metody pro práci se soubory jako `rename()`, `delete()`, `cwd()`, `pwd()`, `mkd()` a další. Pro nedostatek prostoru se na ně nedostalo, jsou ale dobře popsány v dokumentaci jazyka Python.

18.3 Protokol HTTP

Pro práci s klientskou částí protokolu HTTP nabízí Python modul `httplib`. Ten, narozdíl od výše uvedených modulů, nabízí hned dvě třídy. První z nich, `HTTPConnection`, odpovídá spojení s HTTP serverem, druhá, `HTTPResponse`, reprezentuje odpověď HTTP serveru.

Nejprve tedy vytvoříme spojení, instanci `HTTPConnection`. Té musíme předat adresu počítače, na který se má připojit. Můžeme uvést i port, na němž běží HTTP server. Není-li číslo portu uvedeno, použije se standardní 80. Číslo portu předáme buď jako samostatný argument nebo může být součástí adresy počítače (podobně jako je tomu u třídy SMTP).

Po vytvoření instance již můžeme serveru zasílat požadavky pomocí metody `request()`. Její první argument je název metody (např. 'GET', 'PUT'), druhý je URL, jehož se tato metoda týká (např. '/index.html'). Dále mohou následovat nepovinné argumenty, první jsou data, která budou poslána jako součást požadavku, přičemž hodnota hlavičky Content-Lenght je automaticky nastavena na velikost dat. Druhý nepovinný argument může být associativní pole, které reprezentuje hlavičky předané spolu s požadavkem (např. 'User-Agent': 'X-browser/3.0').

Po odeslání požadavku zavoláme metodu `getresponse()`, která vrátí instanci třídy `HTTPResponse`, pomocí níž můžeme přečíst odpověd serveru.

Po získání odpovědi můžeme buď zaslat další požadavek (pokud server podporuje HTTP/1.1) nebo ukončit spojení pomocí metody `close()`, která uzavře spojení se serverem. Další volání metody `request()` pak již nebude možné.

Instance třídy `HTTPResponse` vrácená metodou `getresponse()` obsahuje několik metod a atributů, sloužících k získání informací o odpovědi serveru. Nejdůležitější je metoda `read()`, která vrátí tělo odpovědi jako jediný řetězec. Další důležitá metoda je `getheader()` sloužící pro dotazování hlaviček. První argument je jméno hlavičky, jejíž hodnotu chceme získat. Metodě můžeme předat i druhý, nepovinný argument, který bude použit jako návratová hodnota v případě, že požadovaná hlavička není nalezena.

Ve zkratce ještě zmiňme další atributy. Především `version`, který udává používanou verzi protokolu HTTP (pro HTTP/1.0 je to číslo 10, pro HTTP/1.1 pak 11). Další dvojice atributů – `status` a `reason` – úzce souvisí, jedná se totiž o návratový kód odpovědi. `status` odpovídá jeho číslu, `reason` pak obsahuje řetězec vypovídající více o samotném návratovém kódu.

Následuje příklad (po jeho vykonání obsahuje proměnná data úvodní stránku serveru www.root.cz):

```
>>> from httplib import HTTPConnection
>>> conn = HTTPConnection('www.root.cz')

>>> conn.request('GET', '/')
>>> r = conn.getresponse()
>>> print r.status, r.reason
200 OK
>>> data = r.read()
>>> conn.close()
```

18.4 Modul `urllib`

Modul `urllib` je vysokoúrovňová nadstavba nad moduly `httplib` a `ftplib`. Dokáže rovněž pracovat s lokálními soubory. Dokáže otevřít soubory pomocí protokolů HTTP a FTP a nakonec vrátí souborový objekt, který má stejné rozhraní jako standardní souborový objekt vrácený funkcí `open()`. (Má však omezenou funkcionalitu, podporovány jsou pouze metody `read()`, `readline()`, `readlines()`, `fileno()` a `close()`).

Pro základní práci s modulem `urllib` stačí znát jedinou funkci – `urlopen()`. Té předáme URL, která chceme otevřít a ona vrátí souborový objekt:

```
>>> from urllib import urlopen
>>> f = urlopen('http://www.root.cz')
>>> r = f.read()
```

Tento příklad provádí přesně totéž, co výše uvedený příklad použití modul `httplib`. Pokud URL neobsahuje schema (čili http:, ftp: nebo file:), je automaticky předpokládán lokální souborový systém, čili schema file:.

Modul `urllib` nabízí i třídu `URLOpener`, pomocí níž můžeme dokonaleji řídit přístup ke vzdáleným souborů a hlavně přistupovat k serverům FTP, aniž by byl na obrazovku vypsán

prompt pro zadání hesla. Tato třída totiž obsahuje metodu `prompt_user_passwd()`, jejíž předefinováním získáme požadované chování. Více o modulu `urllib` a třídě `URLOpener` naleznete ve standardní dokumentaci jazyka Python.

Kapitola 19

Zabezpečené prostředí

19.1 Dynamická interpretace kódu

Jako téměř každý interpretovaný jazyk, i Python umožňuje svému kódu přístup k jádru interpretru a proto i program napsaný v Pythonu může spouštět své vlastní bloky kódu. K tomu slouží příkaz `exec`, kterému můžeme předat tři typy objektů. Jednak to může být řetězec. Ten je pak považován za zdrojový kód, zkompilován a následně spuštěn. Dále se může jednat i o otevřený soubor, který je přečten a interpretován jako kód Pythonu. A konečně jím může být i kódový objekt získaný voláním interní funkce `compile()`. Pokud za tímto objektem uvedeme ještě klíčové slovo `in` následované asociativním polem, je toto pole bráno jako prostor jmen. Dokonce můžeme uvést i dvě asociativní pole, pak jedno má význam globálního a druhé lokálního prostoru jmen. Neuvedeme-li ani jedno pole, je kód spuštěn přímo v kontextu kódu obsahujícího příkaz `exec`. Následující ukázka naznačuje používání tohoto příkazu:

```
>>> v~= 1
>>> g = {'v': 'jedna'}
>>> l = {'v': 'jedenact'}
>>> exec 'print v'
1
>>> exec 'print v' in g
jedna
>>> exec 'print v' in g, l
jedenact
```

V případě víceřádkového složeného příkazu (`for` apod.) musí být řetězec k interpretaci ukončen znakem nový řádek. Více o tomto příkazu se dozvíte z dokumentace jazyka.

Zatímco příkaz `exec` slouží pro spouštění jednotlivých příkazů, pro vyhodnocování výrazů slouží interní funkce `eval()`. Té předáme jako řetězec požadovaný výraz, funkce `eval()` provede jeho vyhodnocení a výsledek vrátí jako svou návratovou hodnotu. Podobně jako u příkazu `exec` jí můžeme předat dvě asociativní pole ve významu globálního a lokálního prostoru jmen.

```
>>> eval('1 + 1')
2
>>> jedna = 2
```

```
>>> eval('jedna + jedna')
4
>>> eval('eval("\'Ahoj \'") * 3')
'Ahoj Ahoj Ahoj '
```

Kromě funkce `eval()` zná Python ještě interní funkci `execfile()`, která provede spuštění všech příkazů uložených v souboru. Argumentem je soubor otevřený pro čtení.

19.2 Bezpečné prostředí

V tuto chvíli si pravděpodobně každý programátor-začátečník položí otázku: Proč se vůbec bezpečným prostředím zabývat? Pravidelným čtenářům ROOTa je odpověď na tuto otázku jistě více než jasná, bezpečnost aplikací by měla být na prvním místě. Ne vždy tomu tak ale je, spousta programátorů této problematice nevěnuje dostatečnou pozornost!

Každý program v Pythonu má díky mnoha modulům přístup téměř ke všem službám operačního systému. Bez omezení může otevírat soubory a pracovat s adresáři, odesílat signály, případně může používat další „citlivá“ systémová volání. Pro většinu typů úloh to plně postačuje.

V některých případech však tato přílišná otevřenost aplikace může být na závadu. Typickým příkladem je internetový browser Grail, napsaný v Pythonu, jenž umožňuje interpretaci apletů napsaných v tomto jazyku a vložených do webových stránek. Představte si, že by tyto aplety mohly využívat veškeré možnosti vašeho počítače, nekontrolovaně by používaly systémová volání, upravovaly by vaše soubory nebo komunikovoly s vnějším světem. A protože browser potřebuje úplnou sadu služeb, kdežto aplet samotný musí mít tuto sadu (vcelku razantně) omezenu, nelze použít postup, jenž by napadl pravděpodobně každého – odstranění nechtěných modulů (nehledě na to, že některé moduly ani odstranit nelze, jsou zakompilovány přímo v jádře interpretru).

Většina interpretovaných jazyků (ne-li všechny) mají z těchto důvodů implementovány prostředky, umožňující některým blokům programu omezit možnosti, co se týče využívání služeb operačního systému. V těchto blocích pak lze spouštět i nedůvěryhodný kód (za nedůvěryhodný kód by měl být požadován každý kód, který nenapsal programátor aplikace ani její uživatel). Tento kód pak nemá přístup ke službám operačního systému. Jejich rozsah je závislý na implementaci tohoto bezpečného prostředí. Některé implementace (včetně té pythonovské) dokáží možnosti bezpečného prostředí nastavit přesně na míru požadavkům aplikace.

Kód spouštěný uvnitř tohoto programového pískoviště nemá žádnou šanci dostat se mimo toto prostředí (jde o pouhý teoretický předpoklad, v praxi závisející na kvalitě implementace bezpečného prostředí). Nedůvěryhodný kód v jazyce Python může vytvořit opět své bezpečné prostředí, v němž lze spustit další kód, přičemž množinu služeb, nabízených tomuto kódu, lze pouze zúžit, nikdy ne rozšířit! Takto lze bezpečná prostředí zanořovat do sebe a pečlivě tak odladit bezpečnostní politiku naší aplikace.

Bezpečného prostředí nemusíme používat pouze k omezení pravomocí kódu, jde pomocí něho nahradit služby operačního systému uživatelskými. Například takto lze všechny požadavky na souborový systém přesměrovat na vzdálený server. Pro kód běžící v bezpečném prostředí pak budou všechna volání týkající se souborů vypadat jako klasický lokální přístup, bezpečné prostředí se ale postará o předání požadavků souborovému serveru.

19.3 Bezpečné prostředí a Python

V jazyce Python bezpečné prostředí reprezentují instance třídy `RExec`, kterou exportuje modul `rexec`. Tyto instance nabízejí všechny potřebné funkce pro interpretování kódu uvnitř bezpečného prostředí. Při vytváření instance třídy `RExec` jí můžeme předat instanci třídy `RHooks`, jejíž metody jsou volány při importování modulů. Je třeba ještě podotknout, že ke spouštění příkazů můžeme použít metody `r_exec()`, `r_eval()` a `r_execfile()`. Ty mají obdobnou funkci jako příkaz exec a interní metody `eval()` a `execfile()`, pracují však nad bezpečným prostředím. Jednoduchá ukázka použití třídy `RExec` může vypadat třeba takto:

```
>>> import rexec
>>> sandbox = rexec.RExec()
>>> sandbox.r_exec('import os')
>>> sandbox.r_exec('os.remove("/bin/zsh")')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "/usr/local/lib/python2.2/rexec.py", line 254, in r_exec
exec code in m.__dict__
File "<string>", line 1, in ?
AttributeError: 'module' object has no attribute 'remove'
>>> sandbox.r_exec('print dir(os)')
[ ... ]
```

Jak vidíme z příkladu, nejprve jsme si vytvořili bezpečné prostředí reprezentované instancí třídy `RExec`. Dále jsme v tomto prostředí pomocí jeho metod spouštěli příkazy. První importoval bezpečnou verzi modulu `os`, další se pak pokusil smazat soubor /bin/zsh. Bezpečná verze modulu `os` však nepodporuje funkci `remove()`, proto toto volání selhalo. Nakonec jsme si nechali vytisknout seznam všech funkcí, exportovaných modulem `os`.

Nejjednodušeji se bezpečné prostředí nechá nastavovat pomocí třídních atributů třídy `RExec`. Pro jejich změnu odvoďte od třídy `RExec` třídu novou a tyto atributy předefinujte (všechny tyto atributy jsou tuple řetězců):

- `nok_builtin_names` – interní jména, která nebudou z bezpečného prostředí přístupná.
- `ok_builtin_modules` – interní moduly, které lze bez problémů importovat. Jde o moduly typu `math`, `time`, `struct` apod.
- `ok_path` – tuple cest, ve kterých budou hledány moduly, které se mají importovat.
- `ok_posix_names`, `ok_sys_names` – jména funkcí modulu `os` resp. `sys`, která lze bez problémů používat.

Obsah těchto proměnných je nastaven tak, aby nebylo možné nekotrolovaně používat například sokety, pracovat se soubory, importovat nebezpečné nebo nedůvěryhodné moduly a další činnosti, které by mohly ovlivnit bezpečnost počítače, na němž interpretr běží.

Zároveň můžeme přepsat metody třídy `RExec`, které jsou používány pro další účely. Jsou to především metody:

- `r_import` – metoda volaná při importování modulu, jde-li o nebezpečný modul, vyvolá výjimku `ImportError`. Standardní implementace volá metody instance třídy `RHooks` předané při incializaci prostředí.
- `r_open` – metoda volaná při otevírání souboru z bezpečného prostředí. Vrátí souborový objekt reprezentující otevřený soubor. Standardní implementace povoluje otevření pouze pro čtení.
- `r_reload`, `r_unload` – metody volané při znovunačtení/odstranění modulu. Standardně volá metody instance třídy `RHooks`.

Pokud chcete nějaký standardní modul nahradit svojí vlastní verzí, je možné použít atribut `modules` instance `RExec`. Tento modul má stejný význam jako proměnná `sys.modules`. Jde o asociativní pole, jehož klíče tvoří názvy modulů a hodnoty jsou tyto moduly. Pokud se příkaz import pokusí importovat modul, nejprve se podívá právě do této proměnné. Pokud se zde nachází, nečte se modul znovu z disku, ale použije se právě tato hodnota. Tuto náhradu modulů můžeme provést také odvozením své vlastní třídy od třídy `RHooks`. To je ale již nad rámec našeho seriálu. Proto si prosím projděte zdrojové kódy modulu `rexec` a případně i modulu `ihooks`.

Všechny výjimky rozšířené z bezpečného prostředí se šíří dále standardním způsobem. Speciálním případem je ale výjimka `SystemExit`, kterou kód může vyvolat jednak sám od sebe, jednak ji způsobí i spuštění funkce `sys.exit()`. Pokud nechcete nebezpečnému kódu dovolit ukončení interpretru, musíte každé spuštění bezpečného prostředí obklopit příkazem `try`, který tuto výjimku odchytí a zabrání tak ukončení Pythonu.

Chcete-li modifikovat proměnné uvnitř bezpečného prostředí, máte k nim přístup pomocí modulu `__main__`, který nalezneme uložený v atributu `modules`. Ve výše uvedeném příkladu proto můžeme vytvořit uvnitř bezpečného prostředí novou proměnnou pomocí následujícího příkazu:

```
>>> sandbox.modules['__main__'].foo = 1
>>> sandbox.r_eval('foo')
1
```

Někdy takto potřebujeme do tohoto prostředí dostat určitý objekt, nechceme však umožnit přístup ke všem atributům objektu. To nám zajistí modul Bastion.

19.4 Bastion

Modul `Bastion` nabízí jednu velice užitečnou funkci jménem `Bastion()`. Té lze předat určitý objekt a funkci, která kontroluje přístup k tomuto objektu. Funkce `Bastion()` poté vytvoří zástupce, který při každém pokusu o zpřístupnění libovolného atributu původního objektu nejprve zavolá filtrovací funkci a až podle jejího rozhodnutí případně zpřístupní tento atribut. Filtrovací funkce je funkce, která přebírá jediný argument – jméno atributu – a vrací buď logickou nulu (přístup zamítnut), nebo logickou jedničku (přístup povolen). Filtrovací funkci lze vynechat, pak se použije implicitní filtr, zamezující přístup ke každé metodě, jejíž název začíná znakem podtržítko '_'. Pro zajímavý příklad použití nahlédněte přímo do zdrojového kódu modulu `Bastion`.

Kapitola 20

Ladění programu

20.1 Debugger

Každý program má chyby. To je jedno ze základních pravidel programování. Proto nikoho nepřekvapí, že existuje mnoho postupů, jak se chyb vyvarovat. Může jít o používání průhledných postupů a konstrukcí, používání jednoduchých funkcí apod. Pokud jsme však již zjistili, že náš kód obsahuje chybu, můžeme použít několik metod, jak ji odhalit.

Tou nejzákladnější a velmi oblíbenou metodou je používání příkazu `assert`. Ten má jeden povinný a jeden nepovinný argument. První je podmínka, pokud její logická hodnota nabývá log. 0 (tj. tato podmínka není splněna), dojde k výjimce `AssertionError`, pokud byl příkazu předán ještě druhý argument, bude tento použit jako parametr výjimky. Připomeňme, že příkaz `assert` pracuje pouze, pokud je hodnota interní proměnné `__debug__` rovna 1 (což platí vždy kromě případu, kdy je interpretr spuštěn s volbou `-O`).

Další postup je používání kontrolních výpisů. Kontrolní výpisy jsou jednoduchou možností, jak se přesvědčit, že program pracuje, jak má. Další v Pythonu oblíbený postup spočívá v používání interaktivního interpretru, zde jednak můžeme provést jednotlivé části kódu a okamžitě sledovat, co se děje, jednak můžeme provést i tzv. suchý běh, kdy si sami spouštíme řádek za řádkem a sledujeme stav proměnných.

Pokud výše uvedené možnosti selžou, přijde na řadu *debugger*, ladící program, v němž můžeme kontrolovat provádění jednotlivých částí našeho kódu. Přestože debugger je velice silný nástroj, jeho přílišné používání v místech, kde nám mohou pomoci kontrolní výpisy, příliš nedoporučuji.

Velkou výhodou Pythonu je, že debugger je vlastně program kompletně napsaný v Pythonu samotném, nejde o žádnou službu jádra interpretru apod. Debugger podporuje nastavování zarážek (breakpointů), které mohou být dokonce podmíněné, krokování kódu na úrovni zdrojového kódu, výpis zdrojového kódu, prohlížení zásobníku volaných funkcí a vyhodnocování libovolných příkazů jazyka Python v kontextu vybraného rámce ze zásobníku volaných funkcí.

Veškerý kód debuggeru je součástí modulu `pdb`. Debugger se (což nikoho asi nepřekvapí) ovládá pomocí řádkového rozhraní, jež je implementováno za použití modulu `cmd`. Tento modul nabízí několik funkcí, kterými se debugger spouští. Jde především o tyto:

- `run()` – této funkci předáme jeden argument – řetězec – reprezentující příkaz, který se má ladit, je možné jí předat i další dva argumenty – asociativní pole, které jsou použity jako globální (resp. lokální) prostor jmen.

- `set_trace()` – funkce bez parametrů, která vyvolá debugger v místě svého zavolání.
- `pm()` – další funkce bez parametrů, spustí post-mortem debugger, který umožní zjištění příčin, za nichž vznikla výjimka.

Další možností, kterak spustit debugger nad určitým programem, je přímé spuštění modulu `pdb`, kterému jako argument předáme jméno laděného programu:

```
% python /usr/local/lib/python2.2/pdb.py muj_program.py
```

Po spuštění debuggeru můžeme provádět kontrolovaný běh programu. Zde si uvedeme pouze některé příkazy, jejich kompletní popis najdete v dokumentaci jazyka. První množinou příkazů jsou příkazy, které spouští kód laděného příkazu. Jde o příkaz `step`, který vykonává jedinou činnost – v případě, že program volá nějakou funkci, debugger začne tuto funkci trasovat. Podobným příkazem je next. Ten však funkce spouští jako jediný příkaz a neprovádí jejich trasování. Příkaz `continue` spustí program od aktuálního místa, program poběží, dokud nebude zastaven nějakou zarážkou. Obdobně příkaz `return`, ten vykoná zbytek aktuální funkce, zastaví se buď na konci funkce, nebo na příkazu `return`, který způsobil ukončení funkce.

Další sada příkazů se týká nastavování zarážek. Nejdůležitějším je příkaz `break`, jemuž můžeme předat buď jméno souboru a číslo řádku, na němž má zarážku nastavit, nebo jméno funkce: pak bude zarážka aktivována na prvním příkazu této funkce. Dále je možné uvést ještě podmínku, při které bude zarážka aktivní. Při nastavení zarážky příkaz vytiskne číslo breakpointu, které bude možné použít v dalších příkazech. Jedním z nich je příkaz `clear`, jenž odstraní zarážku danou číslem.

Velice užitečné jsou příkazy pro práci se zásobníkem volaných funkcí. Jedná se o trojici příkazů: `where`, `up` a `down`. Jejich význam je zřejmý z názvu, takže jen ve zkratce: `where` vytiskne zásobník volaných funkcí, přičemž funkce na jeho vrcholu je vytištěna jako poslední, aktuální rámec pak reprezentuje šipka. `Up` skočí na rámec funkce, která zavolala funkci v aktuálním rámci (pohybujeme se zásobníkem směrem vzhůru), `down` naopak posune ukazatel o jednu funkci hlouběji (pohyb dolů).

Následují příkazy bez rozdělení do skupin: příkaz `list` vytiskne zdrojový kód v okolí aktuálního řádku. Je možné mu předat i rozsah řádek, odkud kam má kód vytisknout. Příkaz `args` vytiskne seznam argumentů aktuální funkce. Velice užitečný je i příkaz `help`, jenž vytiskne nápovědu ke každému příkazu debuggeru. Pro ukončení debuggeru slouží příkaz `quit`.

Všechny příkazy můžeme zkrátit (např. step = s, next = n ...), pokud příkaz předaný debuggeru není jeho vlastním příkazem, je považován za příkaz jazyka Python a debugger ho takto spustí. Pokud bychom se chtěli vyhnout nejednoznačnostem, můžeme použít příkaz `!`. Všechny znaky zapsané za ním bere debugger jako pythonovské příkazy.

20.2 Profiler

I když jste již svůj kód zbavili většiny chyb, může na vás číhat další nepříjemnost. Tou je pomalý běh programu. Program v jazyce Python je interpretovaný, což pro něj zcela jednoznačně znamená pomalejší vykonávání než u jazyků kompilovaných. Přesto má programátor možnost, jak běh zrychlit. Může totiž kritické části kódu přeorganizovat nebo v případě nejhorším přepsat do C, čímž získá výhody (i nevýhody) kompilovaného kódu.

A právě pro zjištění těchto kritických míst nabízí Python *profiler*, čili program, který přesně určí, jakou funkci program vykonával a jak dlouho, kolikrát ji během své činnosti zavolal, a další, pro programátora velice užitečné informace. Mnohdy se programátor nestačí divit, předpokládal, že program tráví nejvíce času uvnitř nějaké dlouhé smyčky, náhle ale zjistí, že jeho kód nejvíce brzdí špatně napsaná funkce volaná z tohoto cyklu.

Profiler jazyka Python je deterministický, což znamená, že po každém příkazu spočítá dobu, jak dlouho byl vykonáván, a zaznamená i další užitečné informace. Naproti tomu existují i profilery statistické. Ty v pravidelných intervalech vzorkují ukazatel aktuální instrukce a podle těchto vzorků určují, kde program strávil nejvíce času.

Profiler jazyka Python umožňuje profilování pouze kódu Pythonu, jakékoli externí moduly v C jsou pro něj skryty, čas v nich strávený počítá jako čas funkce, která funkce těchto externích modulů volala. Pokud však kód v C volá funkci Pythonu, je již tato klasicky profilována.[1]

Všechna funkcionalita profileru je založena na modulu `profile`. Spolu s tímto modulem zmíníme i modul `pstats`, který slouží k interpretaci výsledků profileru. Veškeré možnosti profileru nám zpřístupňuje funkce `profile.run()`. Její první (povinný) argument je řetězec – příkaz, který má být profilován. Dále je možné uvést i další řetězec. Ten bude použit jako jméno souboru, do nějž budou uloženy statistiky, které vyprodukoval profiler. K interpretaci těchto statistik je určena třída `pstats.Stats`. Pokud funkci `run()` nepředáme jméno souboru, vytiskne profilovací zprávu, obdobně můžeme modul profile spustit i jako hlavní modul, pak také vytiskne zprávu, která může vypadat podobně:

```
main()
2706 function calls (2004 primitive calls) in 4.504 CPU seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)
  2    0.006   0.003   0.953   0.477   pobject.py:75(save_objects)
43/3   0.533   0.012   0.749   0.250   pobject.py:99(evaluate)
 ...
```

Jak vidíte, zpráva obsahuje několik sloupců. Jejich význam si nyní postupně objasníme.

- `ncalls` – počet volání funkce, je-li ve tvaru 43/3, pak druhé číslo udává počet primitivních volání funkce (to jsou všechna volání kromě rekurzívních, tj. případů, kdy funkce volá sama sebe)
- `tottime` – čistý čas strávený ve všech voláních funkce, neobsahuje čas strávený ve funkcích jí volaných
- `precall` – v obou případech jde o přepočet na jedno volání funkce
- `cumtime` – čas strávený ve všech voláních funkce a všech její „podfunkcích“
- `filename` – identifikuje funkci, jde o jméno souboru a číslo řádku, jméno funkce následuje v závorce

[1] Obdobná pravidla platí pro debugger `pdb`.

Jestliže jste funkci `run()` předali jméno souboru, je profilovací zpráva uložena do tohoto souboru. Jelikož „doba života“ profilovacích protokolů je velice krátká, není zaručena kompatibilita formátu tohoto souboru mezi jednotlivými verzemi jazyka.

Pro interpretaci profilovacích protokolů je zde modul `pstats`. Jeho třída `Stats`, jejímuž konstruktoru předáte jméno souboru s protokolem, podporuje mnoho metod, díky nimž můžete manipulovat s výsledky profilování a získat tím přesně to, co chcete. Zde si uvedeme pouze nejzákladnější metody:

- `sort_stats()` – seřadí zprávu podle zadaného klíče ('calls', 'cumulative', 'file', 'module', 'pcalls', 'line', 'name', 'nfl', 'stdname', 'time')
- `reverse_order()` – zrevertuje pořadí prvků, první bude poslední atd.
- `print_stats()` – vytiskne profilovací zprávu, je možné jí předat argumenty, pomocí nichž lze vyfiltrovat pouze ty funkce, jež nás zajímají
- `print_callers()` – vytiskne seznam funkcí, které volaly každou funkci v profilovacím protokolu
- `print_callees()` – vytiskne seznam funkcí, které byly volány z každé funkce v protokolu

20.3 IPython

Pokud jste někdy používali interaktivní mód Pythonu, jistě víte, že jde o velice silný nástroj. Když si k tomu ještě přičtete přístup k debuggeru, dokumentačním řetězcům, shellovým příkazům a mnoho dalšího, dostanete IPython [`http://www-hep.colorado.edu/~fperez/ipython`].

- Přidávání vlastních příkazů, které lze používat z prostředí IPythonu.
- Barevné a číselné rozlišení vstupu/výstupu Pythonového shellu.
- Vytváření logů, do nichž se ukládají spouštěné příkazy, posléze je možné tyto logy „přehrát“, čímž získáme (nebo nezískáme :) stav předchozího sezení.
- Je možné ho volat z libovolného uživatelského kódu, čímž získáme další výborný ladicí nástroj.
- Automatické doplňování závorek. Např. funkci "sin(3)" lze volat i jako "sin 3".

Kapitola 21

Práce s grafikou

21.1 Co je PIL?

Python Imaging Library je balíček modulů pro práci s grafickými daty z prostředí jazyka Python. Zdrojový tarball si můžete stáhnout z [`http://www.pythonware.com/products/pil/`] domovské stránky. Instrukce pro nainstalování jsou zahrnuty v balíčku. Je rovněž možné použít předkompilované balíčky pro vaši distribuci Linuxu (v Debianu např. python2.2-imaging). Po nainstalování máte přístupný pythonovský balíček `PIL`, někdy (např. v Debianu) může být instalace Pythonu nakonfigurována tak, že cesta `sys.path` již obsahuje adresář tohoto balíčku, takže všechny jeho moduly jsou přístupné přímo bez uvedení prefixu (jméno balíčku).

21.2 Jak a k čemu lze PIL použít?

Především je PIL výhodná pro manipulaci s grafickými daty v různých formátech. Lze pomocí ní vytvářet náhledy souborů, generovat nové grafické soubory, provádět různé úpravy parametrů obrázku (změna kontrastu, jasu). Lze dokonce vytvořit u uživatelské filtry, které modifikují původní obrázek. Budete-li Python Imaging Library používat delší dobu, zjistíte, jaká síla se v ní ukrývá.

Jestliže budete chtít vytvořit nový obrázek, máte několik možností. Především lze obrázek načíst z nějakého souboru. Paleta podporovaných souborových formátů je velice široká, proto je téměř vyloučena možnost, že PIL nedokáže nějaký formát přečíst. Další možností, kterou máte, je vytvoření obrázku „na zelené louce“. Konečně, obrázek lze získat i operacemi nad jiným obrázkem (výřez, získání barevného kanálu apod.).

Grafická data lze přečíst ze souboru pomocí funkce `open()` modulu `Image`:

```
>>> import Image
>>> obrazek = Image.open('galeon.png')
```

Tím jsme získali instanci třídy `Image.Image`, která nám poskytuje přístup k různým informacím ohledně obrázku:

```
>>> obrazek.size
(48, 48)
>>> obrazek.format
'PNG'
```

```
>>> obrazek.mode
'RGBA'
```

Je důležité vědět, že PIL se nepokouší načíst celý obrázek hned při volání funkce `open()`, ale až při požadavku na nějaká grafická data. Proto lze při otevření souboru načíst relativně malou hlavičku informující o velikosti obrázku, případně jeho módu, a vyčkávat, zda uživatel nebude chtít obrázek zpracovávat. Proto můžeme velice rychle přečíst informace o mnoha souborech najednou.

Instance třídy `Image.Image` podporují také množství metod, díky kterým můžeme s obrázkem pracovat. Například již zmíněný náhled získáme voláním metody `thumbnail()`:

```
>>> obrazek.thumbnail((96, 96))
```

Metoda `thumbnail()` modifikuje původní obrázek, abychom si mohli výsledky našeho snažení prohlédnout, musíme obrázek uložit pomocí volání metody `save()`:

```
>>> obrazek.save('galeon.thumb.png')
```

V tomto případě je formát výstupního souboru určen na základě přípony souboru, pokud nebude knihovna moci formát souboru rozpoznat, musíte jí předat formát, který požadujete, např: `obrazek.save('galeon.thumbnail', 'PNG')`

Pro účely ladění se může hodit metoda `show()`, která obrázek uloží do dočasného PPM souboru a zavolá program `xv` pro jeho zobrazení. Pokud se budete PIL zabývat více, zjistíte, že existuje dokonce rozhraní, které umožňuje z obrázku vytvořit Tk widget, tudíž takto lze spolupracovat s nejrozšířenějším toolkitem pro Python.

Pokročilejší metody nabízejí manipulaci s výřezy obrázku. Z obrázku můžete získat podobrázek, ten třeba upravit a navrátit ho zpět do původního:

```
>>> pulka = obrazek.crop((0, 0, 24, 47))

>>> pulka.load()
>>> pulka = pulka.transpose(Image.FLIP_LEFT_RIGHT)
>>> obrazek.paste(pulka, (0, 0))
```

Metoda `crop()` z původního obrázku získá obdélníkový podobrázek, přičemž levý horní roh má souřadnice [0, 0] a pravý dolní [24, 47]. Následné volání metody `load()` je sichr, metoda `crop()` totiž může, ale *nemusí* vytvořit nezávislou kopii data, tudíž změny v původním obrázku se nám mohou promítat do naší poloviny, se kterou chceme dále pracovat. Metoda `load()` zajistí, že se data zkopírují vždy, tudíž se přeruší jakékoli vazby mezi původním a novým obrázkem.

Následné volání metody `transpose()` zrcadlově překlopí obrázek. Existují i další možné konstanty, kterými lze řídit chování této funkce (např. `FLIP_TOP_BOTTOM`, `ROTATE_90`, `ROTATE_180`, `ROTATE_270`). Tato funkce vrací nový obrázek, který vznikl aplikováním dané transformace na původní obrázek, v našem případě jsme tento obrázek uložili zpět do proměnné `pulka`. Nakonec jsme polovinu obrázku překlopenou kolem horizontální osy vložili zpět do původního obrázku pomocí metody `paste()`, té musíme předat obrázek a souřadnice, kam se má vložit.

Pro získání obrázku o jiných rozměrech, než má původní obrázek, použijte metodu `resize()` a jako první argument jí předejte velikost nového obrázku. Druhý argument může být volitelně jedna z konstant: `Image.NEAREST`, `Image.BILINEAR`, `Image.BICUBIC`, `Image.ANTIALIAS`. Ty specifikují filtr použitý pro přepočítání do jiného rozlišení. Implicitně je použit `Image.NEAREST`:

```
>>> obraz = obrazek.resize((128, 128), Image.ANTIALIAS)
```

21.3 Práce s jednotlivými kanály

Jak nám již prozradila hodnota atributu `mode`, náš obrázek pracuje v módu RGBA, což znamená, že hodnota jednoho pixelu je složena ze čtyř složek – červené, zelené, modré a míry průhlednosti. PIL umožňuje práci s těmito barevnými módy: 1 (černá a bílá, 1b/px), L (256 odstínů šedi, 8b/px), P (mapování do jiného módu za použití palety, 8b/px), RGB (3x8b/px), RGBA (RGB a míra průhlednosti 4x8b/px), CMYK (4x8b/px), YCbCr (3x8b/px), I (32bitové celočíselné hodnoty pixelů), F (32bitové reálné hodnoty pixelů).

Knihovna bez problémů podporuje převody mezi jednotlivými módy, umožňuje to metoda `convert()`, které předáte řetězec reprezentující nový mód:

```
>>> cmyk_obrazek = obrazek.convert('CMYK')
```

Co víc, PIL dokáže pracovat s jednotlivými kanály odděleně. Metodou `split()` můžeme obrázek rozdělit na tuple dílčích obrázků (s módem `L`), které reprezentují každý jeden kanál (tj. jeden reprezentuje červenou složku obrázku, druhý zelenou, třetí modrou a čtvrtý míru průhlednosti).

S těmito obrázky potom můžeme pracovat jako s klasickými instancemi `Image.Image`. Pokud budeme chtít obrázek získat zpět z daných kanálů, použijeme funkci `Image.merge()`, které předáme jako první argument mód nového obrázku a jako druhý tuple, jehož prvky jsou kanály nového obrázku. Následující fragment kódu vzájemně prohodí všechny tři barevné kanály obrázku:

```
>>> R, G, B, A~= obrazek.split()
>>> obrazek2 = Image.merge('RGBA', (G, B, R, A))
```

21.4 Kreslení do obrázku

Jak jsme si již řekli, obrázek můžeme vytvořit celý od základů. Nyní si povíme, jak na to. Ti, kteří znají nějaký grafický toolkit (např. Qt nebo GTK+), jistě vědí, že pro kreslení do nějakého primitiva se používá tzv. „plátno“, čili objekt, pomocí jehož metod se dají kreslit čáry, obdélníky a další. Obdobou tohoto plátna je instance třídy `ImageDraw.Draw`. Jejímu konstruktoru jednoduše předáte obrázek a již můžete kreslit:

```
>>> import ImageDraw
>>> obrazek3 = Image.new('RGB', (120, 90))
>>> platno = ImageDraw.Draw(obrazek3)
>>> MODRA = 0xff0000
>>> BILA = 0xffffff
>>> CERNA = 0x000000
>>> platno.rectangle((0, 0, 120, 90), fill = BILA, \
...   outline = CERNA)
>>> platno.ellipse((40, 20, 50, 30), fill = MODRA, \
...   outline = CERNA)
>>> platno.ellipse((70, 20, 80, 30), fill = MODRA, \
```

```
...   outline = CERNA)

>>> platno.arc((20, 0, 100, 80), 25, 155, fill = MODRA)
>>> del platno
```

Nyní máme v proměnné `obrazek3` obrázek, který jsme si vlastnoručně nakreslili. Můžeme s ním manipulovat jako s jakýmkoli jiným obrázkem. (Zkuste si ho uložit a uvidíte, co těch několik řádek kódu kreslí.) Toto je jen ukázka, jak lze velice jednoduše kreslit do obrázku, budete-li chtít více informací, nahlédněte do dokumentace k této knihovně.

21.5 Změna parametrů obrázku

Pokud budete chtít změnit parametry obrázku (např. jas, kontrast, barevný tón apod.), bude se vám hodit modul `ImageEnhance`. Ten nabízí několik filtrů, které jsou všechny odvozeny od třídy `ImageEnhance._Enhance`. Ta definuje jednotné rozhraní – konstruktor `__init__()`, kterému předáme obrázek, jehož parametry chceme měnit, a metodu `enhance()`, jež přebírá argument, pomocí kterého se řídí parametry filtru, přičemž po aplikování filtru vrátí nový obrázek. Například pro změnu jasu použijete třídu `ImageEnhance.Brightness`:

```
>>> obrazek4 = Image.open('galeon.png')
>>> import ImageEnhance
>>> filtr = ImageEnhance.Brightness(obrazek4)
>>> jas = filtr.enhance(0.5)

>>> del filtr
```

Další možné filtry, které můžete použít, jsou: `Color` (vyvážení barev), `Brightness` (jas), `Contrast` (kontrast) a `Sharpness` (vyostření).

21.6 Vytváření postscriptových souborů

Poslední zajímavou vlastností knihovny PIL, kterou se budeme v této kapitole zabývat, je možnost vytváření postscriptových souborů. (Každý UNIXový mág jistě ví, k čemu se nám hodí, lze pomocí nich například vyřešit tisk z programu apod.) Modul `PSDraw` nám poskytuje třídu `PSDraw`, pomocí jejíchž metod můžeme PS soubor vytvářet. Jejímu konstruktoru předáme pouze soubor otevřený pro zápis. Ten můžeme dokonce vynechat, pak PIL použije standardní výstup. Následně zavoláme metodu `begin_document()`, vesele si kreslíme, a až bude po všem, zavoláme `end_document()` a `PSDraw` nám postscriptový dokument uloží do souboru:

```
>>> import PSDraw
>>> soubor = open('output.ps', 'w')
>>> ps = PSDraw.PSDraw(soubor)
>>> INCH = 72
>>> ps.begin_document()

>>> ps.line((INCH, INCH), (5*INCH, INCH))
```

```
>>> ps.line((INCH, INCH), (3*INCH, 4*INCH))
>>> ps.line((3*INCH, 4*INCH), (5*INCH, INCH))
>>> ps.end_document()
>>> soubor.close()
```

Pozor, všechny kreslící metody PSDraw používají souřadnicový systém PostScriptu, kde souřadnice [0, 0] odpovídá levému dolnímu rohu. Dále jeden palec odpovídá 72 bodům, proto každou souřadnici násobíme konstantou 72 (INCH). Více se dozvíte ze zdrojových souborů modulu `PSDraw` nebo z dokumentace knihovny PIL.

Kapitola 22

Numeric Python

Numeric Python (zkráceně NumPy) umožňuje programátorovi pracovat s poli čísel, která mohou být svým objemem v řádech megabytů i více, přičemž vyniká velkou rychlostí i při opravdu velkých objemech dat. Všechna data, která NumPy dokáže obhospodařovat, musí být stejného typu, jmenovitě jde o různé typy celých a reálných čísel, komplexní čísla a dokonce odkazy na pythonovské objekty. Tato pole musí být homogenní, tj. žádnou hodnotu nelze vynechat.

Nad těmito poli lze provádět různé matematické funkce, ať se jedná o ty základnější, jako je sčítání nebo násobení, přes goniometrické funkce až třeba po násobení matic atd. Všechny tyto funkce jsou v terminologii Numericu nazývány *univerzální funkce*.

Výkonný kód balíčku je napsán v jazyce C, čímž je docíleno maximální rychlosti provádění. Samotný balíček obsahuje několik modulů. Jde především o samotné jádro umožňující práci s poli (`multiarrays`) a uživatelskými funkcemi (`ufuncs`), dále Numeric nabízí modul, obsahující funkce pro práci s maticemi obdobným způsobem jako v lineární algebře, modul pro vytváření polí, jež mají některé prvky nedefinovány, nebo modul pro aplikování rychlé Fourierovy transformace (`FFT`).

22.1 Numeric a pole

Jak jsme si již řekli, základním stavebním kamenem celého Numericu je nový datový typ implementovaný v C – `multiarray`. Tato pole se chovají obdobným způsobem jako klasické sekvence v Pythonu (seznam, tuple), ale přesto nabízejí několik nových vlastností:

- mohou mít neomezeně dimenzí a přes každou dimenzi mohou být indexována
- jsou homogenní a všechny prvky mají stejný typ
- jejich velikost je neměnná, ale jejich obsah je možné libovolně měnit

Pole je v Pythonu reprezentováno jako klasický objekt, který má několik atributů a metod. Lze k němu přistupovat jako ke kterékoli sekvenci pomocí indexů apod. Každé pole reprezentuje několik vlastností:

- počet prvků (size) – jde o celkový počet prvků v celém poli, tento počet je dán již při vytvoření pole a nelze ho později změnit. Lze zjistit pomocí funkce `size()`.

- tvar (shape) – určuje, kterak jsou prvky uspořádány do jednotlivých rozměrů. Tuto vlastnost reprezentuje atribut shape, jedná se o tuple hodnot, které pro každou dimenzi specifikuje rozměr pole. Tvar pole lze měnit v průběhu jeho života. Je však třeba ale vždy zachovat počet prvků. Proto lze například matici typu 3/5 přetransformovat na 5/3, ale nikdy ne na 2/7.

- počet dimenzí (rank) – pro skalár (tj. bezrozměrné pole) je roven 0, vektor 1 a matice má rank roven hodnotě 2. Počet dimenzí se rovná např. délce tuple specifikující shape, případně ho lze zjistit pomocí funkce `rank()`

- typ prvků (typecode) – určuje typ jednotlivých prvků pole, vrací ho metoda `typecode()`. Jedná se o jediné písmeno (např. 'I' pro pole skládající se z celých čísel, 'D' pro komplexní čísla nebo 'O' pro odkazy na Pythonové objekty).

Nové pole lze vytvořit například pomocí funkce `array()`. Tuto funkci použijeme, pokud budeme chtít vytvořit nové pole z nějakého již existujícího, případně z libovolné pythonovské sekvence:

```
>>> print array(1)                      # skalár
1
>>> print array([1, 2, 3])              # vektor
[1 2 3]
>>> print array([[1, 0], [1, 0]])       # matice typu 2/2
[[1 0]
 [1 0]]
>>> print array([[1, 0], [1, 0]], 'f') # matice reálných čísel
[[ 1.  0.]
 [ 1.  0.]]
```

Funkci `array()` můžeme předat i argument, který bude specifikovat typecode (viz poslední řádek příkladu). Tento postup se hodí, pokud známe strukturu a data, kterými chceme pole naplnit. Numeric nám ale umožňuje ještě jednu, rychlejší cestu, kterak vytvořit nové pole naplněné daty. Předpokládejme, že máme seznam čísel, ze kterých chceme vytvořit čtvercovou matici. Pak jednoduše vytvoříme z tohoto seznamu pomocí funkce `array()` nové pole a následně změníme jeho tvar tak, abychom získali matici požadovaných rozměrů:

```
>>> l = [1, 0, 0, 0, 1, 0, 0, 0, 1]

>>> matice = array(l)
>>> matice.shape = (3, 3)
>>> print matice
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Tento zápis lze zkrátit pomocí funkce `reshape()`, které lze předat pole, ale můžeme jí předat dokonce i obyčejnou sekvenci a ona z předaného shape vytvoří nové pole:

```
>>> matice2 = reshape(1, (3, 3))
>>> print matice2
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Kromě těchto funkcí existují i další funkce, které dokáží vytvořit pole. Například funkce `zeros()`, která vrátí pole iniciované samými nulami, přičemž toto pole má shape nastaveno na hodnotu prvního argumentu funkce `zeros()`:

```
>>> print zeros((2, 3))
[[0 0 0]
 [0 0 0]]
```

Obdobně pracuje i funkce `identity()`, která přebírá jediný argument `n` a vytvoří jednotkovou matici řádu `n`. Numeric definuje i svou vlastní obdobu interní funkce `range()` s názvem `arange()`, která má stejné rozhraní jako `range()` a vrací vektor naplněný hodnotami aritmetické posloupnosti. Při volání `arange()` lze specifikovat i typecode výsledného vektoru. `arange()` dokáže navíc pracovat s reálnými čísly, což obyčejná `range()` neumí!

22.2 Zpřístupnění prvků

Každý datový typ by byl bezcenný, kdyby nad ním nebyly definovány určité operace. Numeric a jeho pole podporují širokou škálu různých funkcí a operátorů, které dokáží s poli pracovat.

Velice důležitou operací, kterou pole umožňují, je zpřístupnění prvku. Jedná se o dokonalé využití slice operací. Rozšířené slice zápisy tak, jak je známe z dnešního Pythonu, mají svůj původ právě v Numericu. Jeho autoři totiž potřebovali silný mechanismus pro indexování, a proto vývojáři v čele s Guido van Rossumem rozšířili specifikaci jazyka přesně podle jejich představ.

Představme si tedy následující pole (jde o třírozměrné pole o rozměrech 2, 3 a 4):

```
>>> m = reshape(arange(24), (2, 3, 4))
>>> print m
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Prvek na pozici 1, 2 a 2 získáme klasicky jako `m[1, 2, 2]`. Zde se pole od svých protějšků v jiných jazycích příliš neliší. Jiná situace ale nastává, pokud chceme získat určité části tohoto pole jako v následujícím příkladě:

```
>>> print m[1, :, :]
[[12 13 14 15]
 [16 17 18 19]]
```

```
>>> print m[:, :, ::2]
[[[ 0  2]
  [ 4  6]]
 [[12 14]
  [16 18]]]
```

Jak vidíte, je možné použití slice i bez udání mezí, pak platí stejná pravidla, jaká jsme si řekli u sekvencí. Všimněte si ale druhého zápisu a obzvláště dvou dvojteček před dvojkou. V tomto případě se jedná o zápis, který z celého třetího rozměru vybere pouze sudé prvky. Číslo za druhou dvojtečkou tudíž specifikuje krok. To je oblast, kde Numeric předstihl Python. Dochází zde k paradoxu, kdy přestože to specifikace jazyka obsahuje, žádný interní datový typ není možné indexovat s udaným krokem. To vše umožní až příští verze Pythonu, lze jen předeslat, že již je vypsán PEP, který toto řeší.

Podobně jako indexování s udaným krokem je další specialitou (alespoň zatím) Numericu rozšiřování symbolu tří teček (`...`). Pokud Numeric narazí na tento symbol uvnitř slice indexů, považuje ho za „všechny dostupné dimenze“, což se těžko popisujem ale je to snadno pochopitelné z následujícího příkladu:

```
>>> print m[..., 1]
[[ 1  5  9]
 [13 17 21]]
```

Z tohoto zápisu je vidět, že to, co vytiskl příkaz `print`, je pole, jehož prvky tvoří všechny prvky pole `m`, které mají poslední dimenzi rovnu 1. Pokud se však v jedné slice vyskytnou dva symboly `...`, pak se expanduje pouze první! Případné další výskyty převede na obyčejnou dvojtečku.

Další možnost, kterou je možné ve slice používat, je slovo `NewAxis`. Pokud se uvede, Numeric na jeho místě vytvoří novou dimenzi. Opět si vše ukážeme na příkladu. První zápis zpřístupní prvky, které mají první dimenzi rovnu 0 a třetí rovnu 1. Jde o tři prvky, které jsou logicky vráceny jako pole mající jednu dimenzi. Pokud bychom však chtěli vytvořit matici, jejíž první sloupec tvoří tyto prvky, použijeme právě `NewAxis`:

```
>>> print m[0, :, 1]
[ 2  6 10]
>>> print m[0, :, 1, NewAxis]
[[ 2]
 [ 6]
 [10]]
```

Všech možných kombinací slice indexů existuje mnoho a je pouze na fantazii programátora, jak je použije. Numeric nabízí všechny možnosti klasických sekvencí a jejich množinu podstatně rozšiřuje. Pro omezený prostor se již budeme věnovat dalším operacím na poli.

22.3 Operace nad poli čísel

Začneme tím nejpodstatnějším faktem – všechny operace pracují prvek po prvku, čili pokud napíšeme mezi dvě pole `*`, *nejde* o násobení matic, ale o pouhé násobení prvků mezi sebou. Pokud bychom si přáli násobit matice, musíme použít funkci `matrixmultiply()`:

```
>>> a = reshape(arange(4), (2, 2))
>>> b = reshape(arange(8,4,-1), (2, 2))
>>> print a + b
[[8 8]
 [8 8]]
>>> print a * b
[[ 0  7]
 [12 15]]
>>> print matrixmultiply(a, b)
[[ 6  5]
 [34 29]]
```

Podotkněme, že binární operátory jako `+` nebo `*` a další musí mít zarovnaný tvar, tudíž, ve zkratce řečeno, musí mít stejné rozměry. Numeric ale podporuje i určitý mechanismus opakování dané operace, tudíž je možné provést i cosi jako:

```
>>> a = identity(2)
>>> b = arange(2)
>>> print a + b
[[1 1]
 [0 2]]
>>> print (a * 3) + 1
[[4 1]
 [1 4]]
```

Pro popis tohoto mechanismu se ale musíte začíst do dokumentace Numericu, kde je vše detailně popsáno.

Všechny operátory jsou implementovány jako univerzální funkce (`ufuncs`), čili mají podobné rozhraní. Tudíž funkce `sin()` pracuje obdobně prvek po prvku jako výše uvedené sčítání apod. Navíc, každá univerzální funkce umožňuje předání ještě jednoho dalšího argumentu, do něhož se uloží výsledek. Takto jsou implementovány in-place operátory jako `+=`, `*=` atd. Ukázka praktického použití s funkcí `log()`:

```
>>> m = reshape(arange(1, 10, 1, Float), (3, 3))
>>> print log(m, m)
[[ 0.          0.69314718  1.09861229]
 [ 1.38629436  1.60943791  1.79175947]
 [ 1.94591015  2.07944154  2.19722458]]
>>> print m[2, 2]
2.19722458
```

Protože se Numeric snaží maximálně šetřit místem, nedojde při zpřístupnění určitých prvků pomocí slice k jejich zkopírování, pouze se vytvoří odkaz na danou oblast v *původním* poli. Proto byste neměli být překvapeni při spuštění následujícího příkladu:

```
> a = zeros((3, 3))

> b = a[:2, :2]
```

```
> b += 1
> print a
[[1 1 0]
 [1 1 0]
 [0 0 0]]
```

Více se nám již do této kapitoly nevešlo, protože Numeric Python je natolik propracovaný a bohatý na funkce, že by jeho popis vydal na samostatný seriál. Nicméně jako motivaci pro vaše další studium lze dodat, že obrázky z PIL lze převést na pole Numericu a zpět, pole Numericu lze ukládat a číst pomocí modulu `pickle`, na úrovni jazyka C lze psát nové univerzální funkce apod. Čili balíček Numeric by měl být součástí každé instalace Pythonu!

Kapitola 23

Distutils

Kdysi v dobách pythonového pravěku se distribuce hotových programů, modulů a balíčků řešila různě, někdo dodával programy stylem „urob si sám“, kdy šlo vlastně o tarball různých modulů, balíčků a skriptů spolu s instrukcemi, jak a kam ten který soubor umístit. Jak je jistě každému jasno, nešlo o cestu úplně ideální, a proto se našlo několik způsobů, jak z této situace ven. Některé programy se daly cestou používající autoconf, kdy byla lokální instalace systému osahána skriptem configure a pak pomocí svaté trojice nainstalována. Poněvadž se ale jedná o Python, přišel ve verzi 1.6 Guido van Rossum s novým balíčkem nazvaným `distutils`.

Dnešní díl poněkud „ošidíme“. Jelikož jsme se ve všech minulých dvaceti dílech vůbec nezmínili o psaní modulů v jazyce C, vynecháme i distribuci takovýchto modulů pomocí `distutils`. Myslím si, že ten, kdo se dostal ke psaní modulů v C, tento seriál již nepotřebuje a ti, kteří o programování rozšiřujících modulů nic neví, by si z dnešního dílu taktéž nic neodnesli.

23.1 Tvorba distribuce

Nejprve začneme trochou teorie. Každý programátor, který napíše nějaký program či modul, by neměl zapomínat na to své dílko důstojným způsobem šířit po světě. Každý sebelepší program je k ničemu, pokud ho uživatel nedokáže nainstalovat a spustit! Vaše práce, tedy pokud píšete svoje programy v Pythonu, je poměrně jednoduchá a nedá se měřit s námahou, kterou musíte vyvinout při šíření třeba programu v C++. Stručně řečeno, musíte udělat pouze následující:

- Napsat instalační skript pojmenovaný podle konvencí setup.py
- Sestavit konfigurační soubory pro vaši instalaci
- Vytvořit distribuci zdrojových kódů
- V případě, že váš program používá rozšiřující moduly napsané v jiném jazyce, vytvořit případné binární distribuce

23.2 Instalační skript

Nejprve samozřejmě musíme mít napsaný program, pro nějž dále vytvoříme instalační skript. V našem případě se omezíme na následující fiktivní software a naším úkolem bude připravit

ho k distribuci:

```
.:
data/ gnomovision/ LICENCE README setup.py

./data:
config.xml images/

./data/images:
icon1.xpm icon2.xpm

./gnomovision:
compile/ gnmv* __init__.py main.py

./gnomovision/compile:
foo.py __init__.py
```

Jak vidíte, jde o překladač s nakladačem gnomovision, máme ho uložen ve stejnojmenném adresáři. Ten má i dva podadresáře, jednak jde o adresář data, obsahující sdílené soubory, a dále o adresář gnomovision, v němž jsou uloženy zdrojové soubory. Program je rozdělen do hlavního skriptu `gnmv`, modulu `main.py` a balíčku `compile`. V hlavním adresáři programu najdeme ještě soubory LICENCE a README. Podle seznamu výše musíme nejprve napsat konfigurační skript `setup.py`. Jeho obsah bude jednoduchý, importuje modul `distutils` a zavolá funkci `setup()`, které předá potřebné parametry, jež budou specifikovat seznamy souborů a další užitečné věci:

```
from distutils.core import setup

setup(name = 'gnomovision',
          description = 'Překladač s nakladačem',
          author = 'Jan Svec',
          author_email = 'honza@py.cz',
          url = 'http://www.root.cz',
          version = '0.1',
          packages = ['gnomovision', 'gnomovision.compile'],
          scripts = ['gnomovision/gnmv'],
          data_files = [('/etc', ['data/config.xml']),
            ('share/gnomovision/images', ['data/images/icon1.xpm',
                          'data/images/icon2.xpm'])
                             ]
)
```

Funkci setup() můžeme předat různé meta-informace o našem balíčku. Jak vidíte z příkladu výše, jedná se o informace typu jméno balíčku, popis, jméno autora, jeho adresa apod. Tyto informace předáváme jako řetězce pomocí pojmenovaných argumentů.

Druhá část argumentů specifikuje balíčky a soubory, z nichž se náš balíček bude skládat. Pomocí pojmenovaného argumentu `packages` jsme funkci `setup()` řekli, že do distribuce má

zahrnout balíčky 'gnomovision' a 'gnomovision.compile'. Kromě těchto balíčků má zahrnout ještě skript uložený do souboru 'gnomovision/gnmv'. Nakonec zbývají datové soubory. Zde se jedná o seznam dvojic (jmeno_adresáře, [seznam_souborů_které_tam_přijdou]). Tedy, do adresáře '/etc' přijde soubor 'data/config.xml' atd.

23.3 Zdrojová distribuce

Jak je na dobrých systémech zvykem, základní tvar, v němž je software šířen, je balíček zdrojových kódů. Knihovna `distutils` proto má funkce, které umožňují takový balíček vytvořit.

Zdrojový balíček se nevytváří jen ze souboru setup.py, ale i z dalšího souboru pojmenovaného MANIFEST.in, v něm jsou popsána pravidla, kterými se specifikují soubory, jež budou do balíčku zdrojových souborů zahrnuty. V našem případě vypadá takto:

```
include LICENCE
include README
include setup.py
graft gnomovision
graft data
global-include README
```

Jak vidíte, jde o obyčejný textový soubor, jenž má na každém řádku jeden příkaz. Pomocí těchto příkazů se specifikují soubory, které budou vloženy do výsledného tarballu. Těchto příkazů je několik:

- *include* – tento příkaz zahrne všechny soubory specifikované předlohami za ním uvedenými (např. 'include *.txt' apod.)
- *exclude* – opak include, soubory, které odpovídají předlohám za ním, nebudou do tarballu zahrnuty (např. 'exclude .*.swp')
- *recursive-include*, *recursive-exclude* – za těmito příkazy je uvedeno jméno adresáře a seznam předloh, pak se na celý tento adresář provede příkaz include/exclude (např. 'recursive-include src *.py')
- *global-include*, *global-exclude* – obdoba rekurzivních příkazů, ale neurčuje se jméno adresáře, neboť tyto příkazy pracují na celém adresářovém stromu zdrojových souborů (např. 'global-exclude *.pyc *.pyo')
- *prune* – za příkazem následuje jméno adresáře, který má být kompletně vynechán, (např. 'prune build')
- *draft* – za příkazem opět následuje jméno adresáře, který má být naopak kompletně vložen (např. 'prune data')

Pokud jsme si již připravili soubor MANIFEST.in, ve kterém jsme popsali celý zdrojový kód našeho balíčku, můžeme se pustit do vytvoření tarballu příkazem shellu:

```
% python setup.py sdist
```

Tento příkaz spustí soubor `setup.py` a vykoná příkaz knihovny `distutils` pojmenovaný 'sdist'. Ten podle souboru MANIFEST.in vytvoří soubor `gnomovision-0.1.tar.gz` umístěný v adresáři `dist`. Pokud by si někdo přál jiný souborový formát než .tar.gz, pak ho může zadat pomocí volby `--formats` (např. 'python setup.py sdist –formats=bztar,zip', seznam všech dostupných formátů viz 'python setup.py –help-formats').

Volbami skriptu `setup.py` předanými na příkazovém řádku je možné velice důkladně řídit chování tohoto skriptu. Můžeme určit, jak se bude jmenovat vstupní šablona (typicky jde o MANIFEST.in), kam budou umístěny výstupní soubory atd.

23.4 Binární distribuce

Mnoho uživatelů balíčků dává přednost binární distribuci před distribucí zdrojových souborů. V našem případě je to poměrně jedno, protože doposud jsme vytvářeli moduly, které byli napsány v čistém Pythonu, tj. neobsahovaly žádné části napsané v C/C++.

Pro vytvoření binární distribuce slouží příkaz obdobný výše uvedenému:

```
% python setup.py bdist
```

Ten opět vykoná soubor `setup.py`, ale tentokráte vytvoří binární distribuci pro náš systém. Opět můžeme specifikovat mnoho voleb, mezi nejdůležitější opět patří přepínač `--formats` určující formát binárního balíčku (v současnosti *gztar*, *bztar*, *rpm*, *zip* a *wininst*, bohužel chybí *deb*!). Pokud jde o tarbally (*gztar* a *bztar*), pak jsou v něm uloženy soubory takovým způsobem, aby se ocitly na správných místech, pokud bude tento TAR rozbalen v kořenovém adresáři. Za zmínku také stojí, že samorozbalovací .exe soubor (formát *wininst*) se vám vytvoří i na Linuxu!

23.5 Instalace balíčku

Balíček můžeme nainstalovat jiným příkazem knihovny `distutils`:

```
% python setup.py install
```

Jak je tomu zvykem u všech příkazů, i 'install' můžeme konfigurovat pomocí voleb na příkazovém řádku (pro úplnost dodejme, že kompletní seznam všech příkazů získate příkazem 'python setup.py –help-commands'). My si zde uvedeme pouze část z nich. První je dvojice voleb `--compile` a `--no-compile`, která rozhoduje, zda kompilovat jednotlivé zdrojové .py soubory do .pyc souborů, či ne. Další volba, která souvisí s kompilací do bytecodu, je volba `--optimize`, která `distutils` říká, že soubory se mají kompilovat i do optimalizovaného bytecodu (soubory .pyo).

Další důležitá volba je `--force` nařizující instalačnímu skriptu, aby přepsal všechny existující soubory. Pro správce systémů může být užitečná volba `--root` určující adresář, který bude považován za kořenový a vzhledem ke kterému se budou všechny soubory instalovat. A ti, kdo nejsou správci systémů, pravděpodobně budou používat volbu `--home` nařizující instalaci do domovského adresáře uživatele.

Pro tvůrce balíčků ještě jedna důležitá věc, příkazy `install` a `bdist`, pokud narazí na skript (viz argument 'scripts' funkce setup()), který začíná magickou sekvencí #!/nějaká/cesta/k/Pythonu, automaticky tento řádek upraví na cestu ukazující na interpretr, kterým byl vykonán skript setup.py (a který se tím pádem považuje za funkční :)

Kapitola 24

Závěr

Dnešní díl o `distutils` byl vyvrcholením našeho seriálu. Jak možná někdo zkušenější zjistil, tento díl byl opravdu stručný výtah, pokud budete chtít, podívejte se i do dokumentace jazyka Python, kde je těmto modulům věnován daleko větší prostor.

Celý seriál uzavřeme výrokem: „Je jedno, jak daleko jste došli, vždy se můžete vrátit a zkusit to jinudy.“ Toto moudro platí všude a v programování dvojnásob.

Rejstřík