Studijní opora k předmětu Skriptovací jazyky

Jan Gaura

4. února 2021

Obsah

Kapitola 1

Úvod

Tento text vznikl pro potřeby výuky předmětu Skriptovací jazyky. Studentům by měl sloužit pro získání nutného minima znalostí ke zvládnutí programování v jazyce kurzu – tedy jazyka Python[1]. V žádném případě však není text plnohodnotnou náhradou za poslechy přednášek a návštěvy cvičení. Studentům je tudíž velmi doporučeno, aby přednášky a cvičení navštěvovali.

Tento text si též neklade za cíl vytvořit kompletního průvodce jazykem Python. Pro takovýto účel lze doporučit některý knižní titul, např.: [1].

1.1 Úvod do skriptovacích jazyků

Skriptovací programovací jazyky tvoří protiváhu tzv. kompilovaným programovacím jazykům. Jednotlivé programy jsou tvořeny zdrojovými kódy (skripty), které jsou tzv. interpretovány, tj. čteny a spouštěny za běhu speciálním programem, tzv. interpretem.

Typický skript těží z výhody, že se nemusí překládat, a často tvoří rozšiřitelnou (parametrickou) část nějakého softwarového projektu, která se může měnit, aniž by bylo potřeba pokaždé rekompilovat hlavní spustitelný soubor. Skripty tak najdeme u her (Quake), grafických uživatelských rozhraní (XULRunner v produktech Mozilla), složitějších softwarových řešení nebo jako hlavní součást dynamických webových stránek (YouTube, Instagram) a podobně.

[1] http://www.python.org

Výhody:

- Není nutné provádět po každé změně kódu kompilaci,
- snadnější údržba, vývoj a správa kódu,
- některé skripty umožňují interpretaci kódu z řetězce (jako například funkce eval() v Pythonu nebo PHP). Něco takového překládané programy bez použití speciálních technik nedokáží.

Nevýhody:

- Rychlost. Interpretace stojí určitý strojový čas a většinou nebývá tak rychlá jako běh přeloženého (a optimalizovaného) programu,
- vyšší paměťová náročnost. Interpret musí být spuštěn a tedy zabírá určitou část operační paměti,
- skriptovací jazyky mají většinou větší omezení než překládané programovací jazyky (např. co do přístupu do paměti, ovládání tzv. handlerů procesů, kontextových zařízení, apod.).

1.2 Programovací jazyk Python

Python je interpretovaný procedurální, objektově orientovaný a funkcionální programovací jazyk, který v roce 1990 navrhl Guido van Rossum. Python je vyvíjen jako open source projekt, který zdarma nabízí instalační balíky pro většinu běžných platforem (Unix, Windows, Mac OS); ve většině distribucí systému GNU/Linux je Python součástí základní instalace.

Python je dynamický interpretovaný jazyk a také jej zařazujeme mezi skriptovací jazyky. Python byl navržen tak, aby umožňoval tvorbu rozsáhlých, plnohodnotných aplikací (včetně grafického uživatelského rozhraní).

Python je hybridní jazyk (nebo také víceparadigmatický), to znamená, že umožňuje při psaní programů používat nejen objektově orientované paradigma, ale i procedurální a v omezené míře i funkcionální, podle toho komu co vyhovuje nebo co se pro danou úlohu nejlépe hodí. Python má díky tomu vynikající vyjadřovací schopnosti. Kód programu je ve srovnání s jinými jazyky krátký a dobře čitelný.

K význačným vlastnostem jazyka Python patří jeho jednoduchost z hlediska učení a proto se jím budeme primárně zabývat v rámci našeho kurzu. Bývá dokonce považován za jeden z nejvhodnějších programovacích jazyků pro začátečníky. Python ale současně bourá zažitou představu, že jazyk vhodný pro výuku není vhodný pro praxi a naopak. Podstatnou měrou k tomu přispívá čistota a jednoduchost syntaxe, na kterou se při vývoji jazyka hodně dbá.

Význačnou vlastností jazyka Python je produktivita z hlediska rychlosti psaní programů. Týká se to jak nejjednodušších programů, tak aplikací velmi rozsáhlých. U jednoduchých programů se tato vlastnost projevuje především stručností zápisu. U velkých aplikací je produktivita podpořena rysy, které se používají při programování ve velkém, jako jsou například přirozená podpora prostorů jmen, používání výjimek, standardně dodávané prostředky pro psaní testů (unit testing) a dalšími. S vysokou produktivitou souvisí dostupnost a snadná použitelnost široké škály knihovních modulů, umožňujících snadné řešení úloh z řady oblastí. V poslední době je Python též oblíben u komunity zabývající se výpočty na superpočítačích (HPC). K tomuto účelu vznikla speciální sada knihoven, která je protiváhou dobře známemu MATLABu – SciPy[2].

[2] http://www.scipy.org/

1.3 Vývojové prostředí jazyka Python

Pro práci v s jazykem Python je doporučováno mít nainstalován Python ve verzi 2.5, 2.6 nebo 2.7. Uživatelé Linuxových distribucí již mají Python nainstalován ve standardní instalaci. Uživatelé Windows si mohou stáhnout instalátor Pythonu z domovské stránky projektu [3], sekce Download.

Skriptovací jazyky mají většinou tu vlastnost, že dovolují spouštění skriptů dvojím způsobem a to v **interaktivním režimu** a nebo spuštění skriptu ze souboru se zdrojovým kódem (**dávkový režim**).

Interaktivní režim lze spustit jednoduše spuštěním interpretu Pythonu z příkazové řádky nebo z programové nabídky. Tento režim slouží většinou k testování funkcionality kódu. Po spuštění interaktivního režimu se na obrazovce objeví následující výpis:

```
Python 3.4.2 (default, Oct 8 2014, 10:45:20)
GCC 4.9.1 on linux
Type "help", "copyright", "credits"or "license"
for more information.
>>>
```

Řádek se znaky >>> je tzv. **prompt** (výzva). Tam již můžeme zadávat příkazy jazyka a pomocí klávesy Enter je spustit. Nyní tedy stačí do promptu napsat `print("Ahoj svete!")` a objeví se nám:

```
>>> print("Ahoj svete")
Ahoj svete
>>>
```

Funkce `print` slouží pro tisk zadaných informací na výstup. Po volání této funkce vidíme výsledek na dalším řádku.

Dávkový režim slouží ke spouštění složitějších skriptů. Takovýto skript spustíme jednoduše na příkazové řádce pomocí příkazu ($ je promptem shellu operačního systému):

```
$ python muj_skript.py
```

Cvičení: Vytvořte si jednoduchý skript, který vypíše na výstup řetězec "Ahoj svete" a ověřte jeho funkcionalitu.

Cvičení: V interaktivním režimu si vyzkoušejte jednoduché počítání. Provádějte operace s čísly jako na kalkulátoru. Pokuste se použít proměnné a závorky.

Cvičení: V interaktivním režimu si vyzkoušejte práci s řetězci. Použijte znaky pro uvození řetězců " a ' a sledujte jak se od sebe liší popřípadě, zda je možné je do sebe zanořit. Přistupujte k jednotlivým prvkům a podřetězcům řetězců pomocí indexů mezi znaky [a]. Řetězce spojujte.

Cvičení: V interaktivním režimu si vyzkoušejte práci s seznamy a n-ticemi (jsou popsány v 6. kapitole knihy). Opět přistupujte k jednotlivým prvkům seznamů a n-tic. Přidávejte prvky do již existujících seznamů.

1.4 Syntaxe

Jazyk Python má relativně jednoduchou syntaxi nez množství různých „kudrlinek" (Anglicky je toto nazýváno pojmem „boilerplate"). Hlavní rysy, se kterými mají někdy začátečníci problémy, by se daly shrnout do následujících bodů:

- nepoužívají se složené závorky `{ }` pro označení bloku, používá se **odsazení**, který je uvozen dvojtečkou `:`,
- u podmínek příkazů **`if`** a **`while`** se nepoužívají kulaté závorky `( )`,
- pro ukončení příkazu nepoužíváme středník `;`, ale jednoduše přejdeme na nový řádek a vložíme další příkaz.

Uveďme si nyní jednoduchý příklad, který shrnuje obě vlastnosti.

```
if a > 0:
    print("Positive")
elif a < 0:
    print("Negative")
else:
    print("Zero")
```

Na příkladu můžeme vidět, že bloky jednotlivých větví příkazu **`if`** jsou uvozeny dvojtečkou a jejich tělo (blok) je odsazen o 4 mezery doprava oproti korespondující větvi. Samozřejmě můžeme do bloku umístit více příkazů tak, že je postupně řadíme pod sebe se stejným odsazením. Též si povšimněme, že není použit znak středníku. Speciálně odsazení bloků je někdy problematické. V jazycích jako je C nebo Java však také běžně bloky odsazujeme proto, abychom je jednoduše vizuálně odlišili. Proto je v jazyce Python této dobře známé praktiky využito s tím, že není nutné psát označení bloků složenými závorkami. Také si povšimněme, že blok je uvozen dvojtečkou. Pro srovnání je ještě zobrazena stejná situace v jazyce C.

```
if (a > 0) {
    printf( "Positive\n" );
}
else if (a < 0) {
    printf( "Negative\n" );
}
else {
    printf( "Zero\n" );
}
```

1.5 Proměnné

V dynamicky typovaných jazycích nedeklarujeme typy použitých proměnných jako třeba v jazycích C/C++, Java, apod. Nebuďme však uvedeni v omyl, že by proměnné neměly svůj typ! Typy jednotlivých proměnných jsou rozlišeny za běhu. V jazyce Python není možné, aby proměnná neměla přiřazen nějaký typ.

Proměnnou vytvoříme jednoduchým přiřazením nějaké hodnoty do jejího jména. Na následujícím výpisu kódu je ukázáno, jak vytvořit proměnnou, zjistit její datový typ a dále do ní přiřadit jinou hodnotu jiného datového typu.

```
In [1]: promenna
---------------------------------------------------------------
NameError                 Traceback (most recent call last)
<ipython-input-2-4c4614e58bc8> in <module>()
----> 1 promenna

NameError: name 'promenna' is not defined

In [2]: promenna = 1
```

```
In [3]: promenna
Out[3]: 1

In [4]: type(promenna)
Out[4]: int

In [5]: promenna = 1.0

In [6]: type(promenna)
Out[6]: float

In [7]: promenna = "Toto je text."

In [8]: type(promenna)
Out[8]: str

In [9]: promenna = True

In [10]: type(promenna)
Out[10]: bool

In [11]: promenna = False

In [12]: type(promenna)
Out[12]: bool
```

Na výše uvedeném výpisu je použit interpret **iPython**[3], který poskytuje větší uživatelský komfort než standardní interpret. Řádek `1` demonstruje, že pokud do proměnné **`promenna`** neuložíme nějakou hodnotu, její jméno neexistuje a ani jej nemůžeme referovat. Také můžeme vidět, že tento příkaz vyhodil výjimku **`NameError`**. Pro teď nám bude stačit konstatovat, že všechny chyby v jazyce Python budou interpretem vyhodnoceny jako výjimky. Díky tomu je také můžeme jednoduše ošetřit. Na řádku `2` vidíme přiřazení hodnoty **`1`** do proměnné **`promenna`**. Na řádku `3` vidíme, že pokud použijeme jméno proměnné, dostaneme její hodnotu. Na řádku `4` se ptáme na typ proměnné pomocí funkce **`type`**. Na řádku `5` jsme přiřadili do proměnné hodnotu typu **`float`** a na řádku **`7`** hodnotu typu **`str`**, která je zkratkou pro typ string. Na řádcích `9` a `11` vidíme použití pravdivostní hodnoty **`True`** a **`False`**, které jsou datového typu **`bool`**.

Na výše uvedeném výpisu programu můžeme vidět první základní vlastnost dynamicky typovaných jazyků a to tu, že proměnná, která má přiřazenu nějakou hodnotu a s ní nějaký datový typ může v průběhu programu tento typ změnit. Samozřejmě to není vlastnost jediná, ale je to vlastnost, která je nejčastěji vidět.

V interpretu můžeme provádět klasické operace s proměnnými tak, jak jsme zvyklí z běžných jazyků (tím myslíme operace sčítání, násobení, apod.). Nezapomeňme, že dělení dvou celých čísel je opět celé číslo. Dělení celých čísel se tedy chová stejně jako v jazyce C.

To, že proměnné nemají předem určený typ však neznamená, že bychom mohli provádět nedefinované operace. Uveďme si jednoduchý příklad.

```
In [13]: 123 + "Ahoj"
---------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-13-4d72da21761a> in <module>()
----> 1 123 + "Ahoj"

TypeError: unsupported operand type(s) for +: 'int' and 'str'

In [14]: a = 123
```

[3] http://ipython.org/

```
In [15]: b = "Ahoj"

In [16]: a + b
---------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-16-f96fb8f649b6> in <module>()
----> 1 a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Na příkladu můžeme vidět, že interpret poznal, že se snažíme o sečtení celého čísla a řetězce. Při takovéto operaci nevíme, jaký má být výsledný datový typ, a proto interpret vyhodil výjimku **`TypeError`**. Ve výpisu je provedena ukázka s použitím proměnných i bez nich, aby bylo vidět, že i jednotlivé literály jsou v jazyce Python objekty. K tématu objektů se vrátíme později.

1.6 Jednoduché datové typy

Nyní si ve stručnosti ukážeme základní datové typy.

Čísla: **`int, long, float, complex`**

- **`23 2345837934346901268 0x17 3.4 5.6+7.8j`**
- operátory: **`+ - * ** / // % ~ & |`** << >>
- build-in funkce: **`abs min max pow round sum`**

Cvičení: V interaktivním režimu si vyzkoušejte práci s čísly. Vyzkoušejte si operátory, speciálně operátor **`**`**. Také si zkuste použití build-in funkcí, které jsou často užitečné.

Řetězce: jednoduché a Unicode

- **`'apostrofy' "uvozovky" r'aw' u"nicode" \n`**
- operátory: **`+`** (spojení, konkatenace), **`*`** (opakovaní), **`% (format)`**
- formátování podobné jako v jazyce C
- jsou neměnitelné (immutable) sekvence: **`len`**, **`[ ]`** (index/slice), **`for`**
- objekt řetězce má mnoho metod

Cvičení: Vyzkoušejte práci s řetězci. Použijte znaky pro uvození řetězců " a ' a sledujte jak se od sebe liší popřípadě, zda je možné je do sebe zanořit. Přistupujte k jednotlivým prvkům a podřetězcům řetězců pomocí indexů mezi znaky [a]. Řetězce spojujte.

1.7 Kolekce

Jedním z důležitých prvků programovacích jazyků je práce s kolekcemi dat. Pod pojmem kolekce si budeme představovat nějakou datovou strukturu, která nám umožní vkládat jednotlivé (třeba i různé) prvky do jedné proměnné. Takováto proměnná se pak bude nazývat kolekcí (shromažďuje v sobě jiné prvky). S kolekcí jsme se již setkali v podobě řetězců, kde je řetězec znaků tvořen jednotlivými znaky, které jsou uloženy v jednotlivých prvcích seznamu. K jednotlivým znakům řetězce též můžeme přistoupit pomocí hranatých závorek uvedených za seznamem. Ukažme si to na příkladu:

```
>>> ret = "Ahoj"
>>> print(ret[0], ret[1], ret[2], ret[3])
A h o j
```

Z uvedeného příkladu vyplývá, že seznamy se do určité míry chovají jako pole v jazyce C.

1.7.1 Seznam (`list`)

Seznamy v jazyce Python jsou však univerzálnější. Mohou ve svých prvcích obsahovat jakékoli další prvky bez ohledu na datový typ (toto je důsledek dynamického typování jazyka). Uveďme si tedy další příklad:

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> print(sez[0], sez[1], sez[2], sez[3])
"Ahoj" 1 2 [1, 2, 3]
```

Indexování seznamu

S poli v jazyce C bychom byli hotovi, v Pythonu však nikoli. Indexovat seznam můžeme i od konce:

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> print(sez[-1], sez[-2], sez[-3], sez[-4])
[ 1, 2, 3] 2 1 "Ahoj"
```

Poslední prvek seznamu má index -1, předposlední -2, atd.

Z nějakého seznamu můžeme vytvořit podseznam (subsekvenci). Tu vytvoříme tak, že do hranatých závorek vložíme index prvního a posledního prvku (bez tohoto prvku), které chceme ve výsledném podseznamu mít. Počátek a konec oddělíme dvojtečkou. Často může nastat případ, kdy chceme podseznam od začátku do nějakého indexu, nebo od nějakého indexu do konce seznamu. V takovémto případě se začátek nebo konec v rozsahu vynechá. Ukažme si příklady:

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> sez[2:4] # od 2. do 3. prvku
[2, [1, 2, 3]]
>>> sez[:2] # od zacatku do 2. prvku
['Ahoj', 1]
>>> sez[2:] # od 3. prvku do konce
[2, [1, 2, 3]]
```

Modifikace seznamu

Seznam můžeme též modifikovat. Nejjednodušeji lze modifikovat určitý prvek seznamu tak, že jej indexujeme a do takto indexovaného prvku přiřadíme požadovanou hodnotu. Stávající hodnota se v seznamu přepíše hodnotou novou.

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> print(sez)
['Ahoj', 1, 2, [1, 2, 3]]
>>> sez[1] = 5 # 2. prvek bude nahrazen cislem 5
>>> print(sez)
['Ahoj', 5, 2, [1, 2, 3]]
```

Přidání prvku na konec seznamu se provede jednoduše zavoláním funkce **`append`** na objektu seznamu. Parametr této funkce je objekt, který se vloží na konec seznamu.

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> print(sez)
['Ahoj', 1, 2, [1, 2, 3]]
>>> sez.append(5)
>>> print(sez)
['Ahoj', 1, 2, [1, 2, 3], 5]
```

Samozřejmě přidávání prvků na konec seznamu není jedinou možností jak přidat do seznamu prvek. Pokud chceme na partikulární pozici v seznamu umístit nějaký objekt, použijeme funkci **insert**, která přijímá dva parametry. Prvním parametrem je pozice, na kterou se prvek vloží, druhým je prvek samotný. Opět si ukažme příklad:

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> print(sez)
['Ahoj', 1, 2, [1, 2, 3]]
>>> sez.insert(0, 5)
>>> print(sez)
[5, 'Ahoj', 1, 2, [1, 2, 3]]
```

Metoda **insert** tedy vloží prvek na zadanou pozici a ostatní prvky za ní o jednu posune.

Odstranění prvku ze seznamu můžeme provézt dvěma různými způsoby. Pokud víme, na které pozici se prvek který chceme odstranit nachází, můžeme použít příkazu **del** a zadat seznam s indexem prvku k odstranění. Druhá možnost je ta, kdy neznáme pozici prvku v seznamu, ale známe jeho hodnotu. V takovémto případě zavoláme metodu **remove** s argumentem prvku k odstranění na našem seznamu (pokud je v seznamu více prvků, které vyhovují zadanému argumentu, odstraní se první prvek zleva). Příklad ukazuje použití obou metod:

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> print(sez)
['Ahoj', 1, 2, [1, 2, 3]]
>>> del sez[1]
>>> print(sez)
['Ahoj', 2, [1, 2, 3]]
>>> sez.remove("Ahoj")
>>> print(sez)
[2, [1, 2, 3]]
```

Práce se seznamem

Čisté přidávání nebo odebírání prvků ze seznamu není zcela užitečnou operací. Nyní si ukážeme několik jednoduchých, avšak užitečných operací nad seznamem.

Velice často se setkáváme s požadavkem, zda seznam obsahuje nějaký prvek či nikoli. Python obsahuje klíčové slovo **in**, které ve spojení se seznamem lze k takovémuto účelu využít. Ukažme si tedy jednoduchý příklad, kdy se budeme snažit zjistit, zda je nějaký prvek v našem seznamu:

```
>>> sez = ["Ahoj", 1, 2, [1, 2, 3]]
>>> print(sez)
['Ahoj', 1, 2, [1, 2, 3]]
>>> "Ahoj" in sez
True
>>> "Cau" in sez
False
```

Tento jednoduchý způsob zjišťování obsahu seznamu se často používá ve webových aplikacích, kde se snažíme zjistit, jestli je v URL předáván nějaký parametr. Ve spojení s podmínkou **if** se pak můžeme rozhodnout, jak se k přijatému HTTP požadavku zachovat.

Pro zjištění maxima nebo minima v seznamu můžeme využít vestavěnou funkci **`min`** a **`max`**, které budou brát jako argument seznam.

```
>>> sez = [8, 1, 2, 5]
>>> max(sez)
8
>>> min(sez)
1
```

1.7.2 N-tice (`tuple`)

Seznam má v Pythonu také nemodifikovatelný ekvivalent v podobě n-tice. Význam slova nemodifikovatelný je třeba chápat tak, že jakmile je n-tice vytvořena, nelze její prvky přidávat ani mazat. Další práce s n-ticí je však stejná jako se seznamem. Můžeme jednoduše vytvářet podn-tice pomocí stejné notace jako u seznamu. Zjišťování obsahu n-tice probíhá obdobným způsobem.

Jak tedy odlišit n-tici od seznamu? Odpověď je velice jednoduchá. Seznam je vytvořen pomocí hranatých závorek, n-tice pomocí závorek kulatých. Pro názornost si opět uveďme jednoduchý příklad:

```
>>> tup = (8, 1, 2, 5)
>>> tup[2:3]
(2, 5)
>>> max(tup)
8
>>> min(tup)
1
```

N-tice používáme hlavně tam, kde víme, že nepotřebujeme měnit jejich obsah. Slouží též jako základ pro konstrukci slovníků.

1.7.3 Slovník (`dict`)

Pokud chceme v seznamu nebo n-tici, které jsou obecně nesetřízené, nalézt nějakou položku, její vyhledání je v čase $O(n)$, kde n je počet prvků uložených v seznamu nebo n-tici. Pro velké kolekce dat je toto samozřejmě značně problematické. Tento problém řeší datový typ slovník. Slovník mapuje klíč (key) na hodnotu (value) pomocí hašovací tabulky. Vyhledání hodnoty pro zadaný klíč je pomocí slovníku v konstantním čase $O(1)$.

Příklady, jako mohou vypadat slovníky:

- **`{}`**
- **`{'key': 12}`**
- **`{1: 'value'}`**
- **`{'key': 'value'}`**
- **`{'Galactica': 75, 'Pegasus': 62}`**
- **`dict(one=1, two=2)`**

Příklad práce se slovníkem:

```
In [1]: slovnik = {}

In [2]: slovnik
```

```
Out[2]: {}

#ulozeni hodnoty pod klic 'SPJA'
In [3]: slovnik['SPJA'] = "Skriptovaci jazyky"

#ziskani hodnoty pro klic 'SPJA'
In [4]: slovnik['SPJA']
Out[4]: 'Skriptovaci jazyky'
```

Je jasné, že pro neexistující klíč není ve slovníku hodnota. Pokud bychom se tedy zeptali na neexistující klíč, obdržíme výjimku **`KeyError`** tak, jak už jsme z jazyka zvyklí.

```
# pokus o ziskani hodnoty pro klic 460
In [6]: slovnik[460]
---------------------------------------------------------------
KeyError                  Traceback (most recent call last)
<ipython-input-6-ef73e3cb1ba9> in <module>()
----> 1 slovnik[460]

KeyError: 460

#ulozeni hodnoty pod klic 460
In [7]: slovnik[460] = 'Katedra informatiky'

#ziskani hodnoty pro klic 460
In [8]: slovnik[460]
Out[8]: 'Katedra informatiky'
```

Pro zjištění, zda-li slovní obsahuje daný klíč můžeme použít klíčového slova **`in`**. Díky tomu můžeme předejít vyhození výjimky neexistujícího klíče. Použití je ukázáno na následujícím příkladu.

```
In [29]: slovnik
Out[29]:
{460: 'Katedra informatiky',
 'SPJA': 'Skriptovaci jazyky'}

In [30]: 470 in slovnik
Out[30]: False

In [31]: slovnik[470] = "Katedra aplikovane matematiky"

In [32]: slovnik
Out[32]:
{460: 'Katedra informatiky',
 470: 'Katedra aplikovane matematiky',
 'SPJA': 'Skriptovaci jazyky'}

In [33]: 470 in slovnik
Out[33]: True
```

1.8 Sekvence

Z výše uvedených datových typů jsou sekvencemi: řetězce (string), seznamy a n-tice. Sekvence můžeme sčítat (**`c1 + c2`**), opakovat (**`c*N`**). Všechny sekvence je také možno indexovat pomocí

jednotného rozhraní: **`c[i]`**, **`'ciao'[2] == 'a'`**. Ze sekvencí též můžeme vybírat různé podsekvence pomocí tzv. slicingu: **`c[i:j]`**, **`c[i:j:k]`**, **`'ciao'[3:1:-1] == 'oa'`**. Více příkladů na toto téma jsme si ukázali v kapitole 1.7.1.

Sekvence také můžeme jednoduše procházet pomocí cyklu **`for`**, kdy v jednotlivých průchodech dostáváme hodnotu na dané pozici ze sekvence v řídící proměnné.

Slovník má tu vlastnost, že není sekvencí, ale je možné jej procházet cyklem **`for`**. Více ukáže následující příklad.

```
In [1]: slovnik = {}

In [2]: slovnik['SPJA'] = "Skriptovaci jazyky"

In [3]: slovnik[460] = "Katedra informatiky"

In [4]: slovnik
Out[4]: {460: 'Katedra informatiky',
         'SPJA': 'Skriptovaci jazyky'}

In [5]: for klic in slovnik:
   ...:     print("Klic:", klic, "Hodnota:", slovnik[klic])
   ...:

Klic: 460 Hodnota: Katedra informatiky
Klic: SPJA Hodnota: Skriptovaci jazyky
```

1.9 Porovnání, testy, pravdivost

Pro porovnání hodnot v jazyce Python používáme operátory **`==`** a **`!=`**. Pro porovnání identity pak operátory **`is`** a **`is not`**. Operátor identity funguje jinak než např. metoda **`equals`** v jazyce Java. V principu operátor identity neporovnává obsah nějakých proměnných, ale porovnává, zda jednotlivé proměnné ukazují na stejný objekt v paměti. Více nám osvětlí následující příklad.

```
In [36]: 123435 is 123435
Out[36]: True

In [37]: 123435 == 123435
Out[37]: True

In [38]: a = 123456

In [39]: a == 123456
Out[39]: True

In [40]: a is 123456
Out[40]: False
```

Na posledním řádku je zřejmé, že proměnná **`a`** ukazuje na nějaké místo v paměti, které je odlišné od místa v paměti, které reprezentuje objekt čísla **`12345`**.

K porovnání pořadí nám slouží operátory **`<`**, **`>`**, **`<=`** a **`>=`**. Členství nějaké hodnoty v sekvenci obstarají operátory **`in`** a **`not in`**. Boolovské hodnoty nám repzentuji singletony **`True`** a **`False`**. Speciálním singletonem je též typ **`None`**, který je do jisté míry ekvivalentní s hodnotou **`null`** v jazyce C.

1.10 Řízení toku

Nejznámější příkaz pro řízeni toku **`if`**, jsme si již uvedli v kapitole 1.4. Schématicky si jej proto pouze připomeneme:

- if <výraz>: <vnořený blok>
  - 0+ elif <výraz>: <vnořený blok>
  - volitelně: else: <vnořený blok>

Pro vytvoření cyklů nám v jazyce Python slouží dvě konstrukce: **`while`** a **`for`**.

1.10.1 Cyklus `while`

Cyklus **`while`** funguje podobně jako jsme zvyklí z jazyka C. Na začátku máme dánu podmínku iterace. Je-li podmínka splněna, je vykonáno tělo cyklu, v opačném případě je cyklus přeskočen a pokračuje se dále v programu. V těle cyklu můžeme použít klíčová slova **`break`** a **`continue`**. Slovo **`break`** zapříčiní ukončení cyklu. Slovo **`continue`** zapříčiní přeskočení zbytku těla cyklu a tím pádem vykonání dalšího cyklu za předpokladu, že je splněna vstupní podmínka.

```
In [1]: i = 0

In [2]: while i < 5:
   ....:     print(i)
   ....:     i += 1
   ....:

0
1
2
3
4
```

1.10.2 Cyklus `for`

Cyklus **`for`** je v jazyce Python poněkud odlišný od svého ekvivalentu v jazyce C. Jeho sémantika je podobná jako u příkazu **`foreach`** v jazycích Java nebo C#. Cyklus **`for`** je schopen procházet pouze sekvence, o kterých jsme si něco řekli v kapitole 1.8. Podstatným rozdílem oproti cyklu **`for`** z jazyka C je to, že řídící proměnná cyklu postupně nabývá jednotlivých hodnot v zadané sekvenci. Odpadá tak nutnost přistupovat k prvkům sekvence pomocí nějaké jiné proměnné. Zápis je také podstatně jednodušší. Ukažme si proto jednoduchý příklad na součet prvků v seznamu.

```
seznam = [1, 2, 5, 10, 100]

sum = 0

for prvek in seznam:
    sum += prvek

sum #118
```

I v cyklu **`for`** můžeme použít klíčová slova **`break`** a **`continue`**. Ukažme si tedy příklad, kdy jsou v seznamu uloženy i hodnoty jiného typu než **`int`** nebo **`float`**, a které samozřejmě spolu nemůžeme sčítat.

```
seznam = [1.5, 2, 'Ahoj', 10, 3+5j]

sum = 0

for prvek in seznam:
    if type(prvek) == int or type(prvek) == float:
        sum += prvek

sum # 13.5
```

Kapitola 2

Funkce

V této kapitole si probereme funkce, které na rozdíl od jazyků C, Java a C# disponují v jazyce Python většími možnostmi použití a definic.

Asi bychom byli schopni naše programy psát tak, abychom nepotřebovali použít funkce. Toto by nám však vystačilo na velmi krátké programy, poněvadž bychom jinak museli všechen znovupoužitelný kód psát znovu a znovu. Základní vlastností funkcí je jejich znovupoužitelnost. Pokud tedy máme úlohu, kterou víme, že budeme požívat více než jednou, je vhodné ji umístit do funkce. Takovýto blok kódu by též měl fungovat pokud možno co nejvíce samostatně.

2.1 Funkce v jazyce Python

V jazyce Python je definice funkce do jisté míry podobná definici funkce v jiných jazycích s tím rozdílem, že neuvádíme návratový datový typ a ani neuvádíme datové typy argumentů funkce. Toto je dáno dynamickou typovostí jazyka. Uveďme si jednoduchý příklad pro výpočet mocniny čísla.

```
def sqr(number):
    return number**2

sqr(3) #9
```

Pokud pomineme to, že v jazyce Python máme pro mocninu operátr **`**`**, měl by nám být zápis funkce celkem jasný. Definice funkce je uvozena klíčovým slovem **`def`** následovaným jménem funkce a v závorce uvedeným seznamem argumentů oddělených čárkou, což si za chvíli ukážeme. Volání funkce je pak provedeno jménem funkce s parametry uvedenými v kulatých závorkách tak, jak je to uvedeno na příkladu.

Funkce nám také dovolují provádět generalizaci, což je zobecnění zadaného problému. Vezměme si výpočet mocniny čísla jako jednoduchý příklad. První ukázka funkce **`sqr`** umí vypočítat pouze druhou mocninu zadaného čísla **`number`**. My bychom však chtěli vytvořit obecnou funkci na výpočet jakékoli mocniny čísla. Taková funkce je v ukázce níže.

```
def pow(number, exponent):
    return number**exponent

pow(2, 3) #8
```

Funkce **`pow`** má nyní 2 parametry s mocněncem **`number`** a mocnitelem **`exponent`**. Místo natvrdo nastavené hodnoty **`2`** pro výpočet druhé mocniny z předcházejícího příkladu je mocnitel zadán parametrem funkce. Máme tak zobecněný (generalizovaný) kód pro výpočet mocniny.

2.2 Parametry funkcí

Z jazyka C znáte využití defaultního parametru funkce. I v jazyce Python je tato možnost zachována. Syntaxe je opět podobná jazyku C, kdy parametr funkce, který může nabývat defaultní hodnoty, má tuto hodnotu v definici funkce nastavenou pomocí rovnítka. **Poznamenejme**, že takovýchto parametrů může být více, ale musí být vždy uvedeny na konci seznamu argumentů funkce, jinak by interpret nedokázal rozlišit, které parametry má nastavit jako defaultní, a kterým přiřadit volanou hodnotu.

Uveďme si tedy jednoduchý příklad, který nám opět generalizuje předchozí příklad tak, že pro výpočet druhé mocniny funkcí **pow** nebudeme muset zadávat hodnotu mocnitele.

```
def pow(number, exponent=2):
    return number**exponent

pow(3) #9

pow(3, 3) #27
```

2.3 Proměnlivý počet argumentů funkce

V jazyce C jste používali funkci **printf**, která slouží pro tisk na výstup. Její zajímavou vlastností je to, že do ní může vstupovat proměnlivý počet argumentů. Pojďme se podívat, jak takovou funkci vytvořit v jazyce Python.

Proměnlivý počet argumentů funkce je možno vytvořit tak, že poslední argument funkce bude mít před svým jménem znak *****. Jakmile pak zavoláme takovou funkci, bude v tomto parametru vždy uložen tuple, který bude obsahovat tolik hodnot, kolik se jich již nevešlo do standardních argumentů funkce. Z toho tedy plyne, že můžeme kombinovat různé typu argumentů funkce. Ukažme si jednoduchý příklad funkce s proměnlivým počtem argumentů.

```
def va_args_function(a, *b):
    print(a, b)

va_args_function(1)
#1 ()

va_args_function(1, 2)
#1 (2,)

va_args_function(1, 3, 4)
#1 (3, 4)

va_args_function(1, 3, 4, "Ahoj")
#1 (3, 4, 'Ahoj')

va_args_function(1, 3, 4, "Ahoj", 5.3)
#1 (3, 4, 'Ahoj', 5.3)
```

Na této ukázce vidíme, jak se tuple v argumentu **b** postupně zaplňuje více hodnotami podle toho, jak jich přibývá při volání funkce. Také si všimněme, že je argument použit bez znaku *****.

Pokusme se nyní vytvořit funkci **super_print**, která bude akceptovat proměnlivý počet argumentů (předpokládejme typ **str**). Tyto argumenty budou vytištěny a jejich výpis bude ohraničen nahoře a dole znakem **=** v délce celé sekvence. V kódu jsme využili metodu **join** na objektu **str**[1].

[1] http://docs.python.org/library/stdtypes.html#str.join

```
def super_print(*p):
  s = " ".join(p)
  print "-" * len(s)
  print s
  print "-" * len(s)

super_print("Jedna", "dve", "3", "4")

#=============
#jedna dve 3 4
#=============
```

2.4 Argumenty funkce předané klíčovými slovy

Posledním typem argumentu funkce je ten, který je předáván jako slovník. Takovýto argument je v definici funkce nejčastěji podle konvence označován jako ****kw**. Argumenty jsou pak funkci předávány tak, že je jako argument funkce použito jméno argument, kterému je přiřazena hodnota pomocí rovnítka. Tyto argumenty jsou odděleny čárkou. Ukažme si jednoduchý příklad.

```
def kw_function(a, **kw):
    print a, kw

kw_function("jedna", SPJA="Skriptovani", ALG="Algorimizace")
#jedna {'ALG': 'Algorimizace', 'SPJA': 'Skriptovani'}
```

Na ukázce také vidíme, že se jednotlivé typy argumentů můžou kombinovat. Podstatnou částí však je, že argument **kw** se uvnitř funkce jeví jako slovník, jenž jsme si již probrali v kapitole 1.7.3. Pozoruhodnou vlastností funkcí, které používají předávaní argumentů klíčovými slovy je jejich rozšiřitelnost.

Představme si, že pracujeme na větším softwarovém díle a potřebujeme změnit určitou komponentu. V takovém případě většinou musíme provést refactoring kódu s tím, že budeme takovou komponentu doplňovat o další parametry její funkce. V případě, že však takováto funkce má parametr předávaný klíčovými slovy, je její rozšíření jednoduché. Stačí pouze upravit operace práce se slovníkem **kw**. Ukažme si tedy typičtější práci s takovou funkcí. Mějme funkci, která filtruje města podle počtu obyvatel. Funkce **filtruj_mesta** s argumentem **gt** vrátí taková města, jejichž počet obyvatel je vyšší než zadaná hodnota.

```
def filtruj_mesta(mesta, **kw):
    """Funkce vrati seznam mest, ktere odpovidani svym
    poctem obyvatel zadanym omezenim.
    V **kw bude mozno predat argument:
     'gt': jen mesta s poctem obyvatel vetsim nez hodnota argumentu.
    Je mozne zadat zadny nebo jeden parametr.
    Pokud nezadame zadny parametr, vrati se prazdny seznam.
    """
    flt = []
    if 'gt' in kw:
        lim = kw['gt']
        flt = [ key for key in mesta if mesta[key] > lim ]
    return flt

mesta = {"Ostrava": 336000, "Praha": 1249000,
         "Brno": 405000, "Olomouc": 101000,
         "Karvina": 63000, "Havirov": 82000}
```

```
filtruj_mesta(mesta, gt=100000)
#['Ostrava', 'Praha', 'Brno', 'Olomouc']
```

Pokud bychom chtěli rozšířit funkcionalitu výše zmíněné funkce o to, aby také uměla vrátit města s počtem obyvatel nižším než bude zadaná hodnota, provedeme to pomocí následujícího kódu.

```
def filtruj_mesta(mesta, **kw):
    """Funkce vrati seznam mest, ktere odpovidani svym
    poctem obyvatel zadanym omezenim.
    V **kw bude mozno predat argument:
     'gt': jen mesta s poctem obyvatel vetsim nez hodnota argumentu,
     'lt': jen mesta s poctem obyvatel mensim nez hodnota argumentu.
    Je mozne zadat zadny, jeden nebo oba parametry nejednou.
    Pokud nezadame zadny parametr, vrati se prazdny seznam.
    """
    flt = []
    if 'gt' in kw:
        lim = kw['gt']
        flt = [ key for key in mesta if mesta[key] > lim ]
    if 'lt' in kw:
        lim = kw['lt']
        flt += [ key for key in mesta if mesta[key] < lim ]
    return flt

mesta = {"Ostrava": 336000, "Praha": 1249000,
         "Brno": 405000, "Olomouc": 101000,
         "Karvina": 63000, "Havirov": 82000}

filtruj_mesta(mesta, gt=1000000, lt=70000)
#['Praha', 'Karvina']
```

Při porovnání obou kódů vidíme, že API dané funkce se nezměnilo a ani její funkce není nijak narušena, pokud je zavolána jen s původním argumentem **gt**.

Poznámka: V kódu jsme použili list comprehension[2], který je popsán v kapitole 2.9.

Cvičení: Promyslete si sami, jak by vypadala varianta rozšíření funkce bez použití předávaní argumentů klíčovými slovy.

2.5 Funkce jako argument funkce

Funkce je v jazyce Python objekt jako každý jiný, což je poněkud jiný přístup, než na jaký jsme zvyklí z jazyka Java. Pokud je funkce objektem, můžeme ji uložit do námi zvolené proměnné, tak jak to zobrazuje následující příklad.

```
def pow(number, exponent=2):
    return number**exponent

sqr = pow

pow(3) #9

sqr(3) #9
```

[2] http://docs.python.org/tutorial/datastructures.html#list-comprehensions

Pokud je však funkce objekt, můžeme jej stejně jako jiné objekty poslat funkci jako její argument. Když je toto v jazyce možné, říkáme, že jsou funkce „first class object". Tato vlastnost je též základním prvkem funkcionálních jazyků. Ukažme si tedy příklad.

```
def sqr_lst(fun, lst):
    ret_lst = []
    for item in lst:
        ret_lst.append(fun(item))
    return ret_lst

sqr_lst(sqr, [1, 2, 3])
#[1, 4, 9]
```

V ukázce jsme využili již naprogramované funkce **sqr**. Funkce **sqr_lst** dostane dva parametry. Prvním parametrem je funkce, která se bude aplikovat na každý prvek ze seznamu, který je předán jako druhý argument. Uvnitř funkce je tedy smyčka **for**, která postupně vybírá prvky ze seznamu s čísly a každé takové číslo pošle jako argument funkce **fun**. Výsledek každého volání je pak přidán do výstupního seznamu. V našem případě jsme funkci **sqr_lst** předali funkci **sqr**, která nám z každého čísla vrátí jeho druhou mocninu.

Pokud pomineme název funkce **sqr_lst**, máme poměrně obecnou funkci, která umí na všechny prvky jakékoli sekvence aplikovat funkci, kterou předáme v parametru. Samozřejmě jsme nyní nevymysleli nic nového. Pouze jsme se relativně nenáročnou cestou dostali k dobře známému konceptu funkce **map**.

Poznamenejme, že v jazycích C/C++ a C# je funkci také možno použít jako argument funkce.

2.6 Funkce map

Funkce **map** je dlouhou dobu známa z funkcionálních jazyků a v současné době se začíná objevovat i v klasičtějších jazycích. Funkce přijímá dva parametry. Prvním je funkce a druhým pak sekvence. Na kazdý prvek ze sekvence je aplikována funkce. Výsledek je pak seznam, který má stejnou délku jako vstupní sekvence a obsahuje výsledky funkce pro korespondující prvky ze vstupní sekvence.

2.7 Anonymní funkce

Při používání funkce **map** se často setkáváme s tím, že funkce kterou chceme aplikovat na prvky sekvence je použita pouze jednou, popřípadě si nechceme do jmenného prostoru umisťovat jednoúčelové funkce. V takovém případě se nám bude hodit koncept anonymní funkce **lambda**. Syntaxi lambda funkcí by měla osvětlit následující ukázka.

```
sqr = lambda x: x**2

sqr(2) #4

sqr(5) #25

pow = lambda number, exponent: number**exponent

pow(2, 3) #8
```

Vidíme, že do proměnné **sqr** jsme přiřadili anonymní funkci s jedním parametrem. Z něj se počítá druhá mocnina. Anonymní funkce též mohou mít více parametrů jako v příkladu funkce **pow**. Pokud bychom anonymní funkci nikam nepřiřadili, nemohli bychom ji později použít. Pro úplnost uvedeme příklad, kdy takto vytvořenou funkci, která není nikde uložena okamžitě zavoláme a získáme tak její návratovou hodnotu. Poznamenejme, že toto použití není příliš časté.

```
(lambda x, y: x**y)(2, 3) #8
```

2.8 Anonymní funkce jako parametry funkcí

Řekli jsme si, že anonymní funkce můžeme použít pro velmi malé úseky kódu, pro které nemá často význam vytvářet speciální funkce. Častým využitím anonymních funkcí je jejich předávaní jako argumentů jiných funkcí. Velmi časté je proto použití lambda funkce s funkcí **map**. Uveďme si tedy opět ukázku na výpočet druhé mocniny seznamu čísel.

```
map(lambda x: x**2, [2, 3, 4]) #[4, 9, 16]
```

Vidíme, že se nám původní příklad funkce **sqr_lst** podařilo zkrátit na jeden řádek.

Nezapomeňme také, že v sekvencích, které mohou vstupovat do funkce **map** se mohou objevit složitější objekty. V lambda funkci pak můžeme volat jejich metody. Následující ukázka převede všechny řetězce na velká písmena.

```
map(lambda x: x.upper(), ["ahoj", "svete"])
#['AHOJ', 'SVETE']
```

2.9 List comprehension

List comprehension je způsob jak vytvářet nové listy. Doposud jsme je byli schopni vytvářet pomocí funkce **map**. List comprehension navíc umožňuje aplikovat na prvky vstupní sekvence predikát. Pokud je predikát vyhodnocen jako logická hodnota **True**, potom je prvek podroben funkci, která je předána jako parametr. Jinak je prvek vyřazen z výstupního seznamu. Syntakticky je list comprehension čitelnější než funkce **map**. Ukažme si několik příkladů.

```
lst = [1, 2, 3, 4]
ships = ['Daru Mozu' , 'Cloud 9' , 'Galactica' ]

squares = [ x**2 for x in lst ]
#[1, 4, 9, 16]

squares = [ x**2 for x in lst if x > 2 ]
#[9, 16]

new_ships = [ x.upper() for x in ships ]
#[ 'DARU MOZU', 'CLOUD 9', 'GALACTICA' ]

new_ships = [ x.upper() for x in ships if len(x) > 7 ]
#[ 'DARU MOZU', 'GALACTICA' ]
```

Na první pohled vidíme, že oproti funkci **map** není pro aplikování požadované funkce na prvky sekvence zapotřebí vytvoření anonymní funkce. Predikát, který je nepovinný se nalézá na konci zápisu.

Zápis pomocí list comprehension je v současné době preferovaným způsobem zápisu pro generování nových seznamů. Je také zajímavé, že funkce **filter** [3] je plně nahraditelná právě tímto zápisem.

[3] http://docs.python.org/library/functions.html?highlight=filter#filter

Kapitola 3

Objektově orientované programování (OOP)

Naši cestu za objektově orientovaným programováním započneme citátem Alana Kaye, spoluautora jednoho z prvních objektově orientovaných programovacích jazyků jménem Smalltalk:

"Actually I made up the term 'object-oriented', and I can tell you I did not have C++ in mind."

The Computer Revolution hasn't happend yet – Keynote, OOPSLA 1997

V této kapitole si představíme objektově orientované programovaní (OOP) v jazyce Python. Objektově orientované programování není v programování novým pojmem, stále však je přechod od procedurálního programování k objektovému někdy obtížný. Někdy může být na překážku statická typovost některých jazyků. My si ukážeme, že je OOP velmi jednoduchý a příjemný způsob programování. Nepodlehněme však představě, že je OOP vše řešící technika.

3.1 Co jsou objekty

Z dosavadní práce v jazyce by nám již mělo být dostatečně jasné, že vše v jazyce Python je objektem. Např. listy obsahující prvky jsou objekty a také elementy v nich jsou samy objekty. I přes tuto skutečnost nám není jazykem objektový model nějak extrémně vnucován. Můžeme se de facto setkat pouze s klasickou tečkovou notací pro volání metod objektů, např. **`lst.append(1)`** pro přidání objektu typu **`int`** do objektu typu **`list`** (proměnná **`lst`**). Tak, jak jsme zvyklí z jazyka C++, i v Pythonu můžeme volání metod nebo atributů zřetězit, např.
`instance_tridy.metoda().atribut`. Tečková notace také nerozlišuje mezi voláním metod nebo přístupu k atributům. Toto je rozlišeno pouze kulatými závorkami u volání metod.

3.2 Motivace k zavedení objektů

Pokud jste programovali v jazyce C, většina programů nejprve využívala funkce, které měly parametry. Jak program nabýval na složitosti, parametrů u funkcí mohlo přibývat. Takovýto proces je více, či méně nevyhnutelný. Pokud jste pracovali dále, možná jste se rozhodli, že si vytvoříte jednoduchou strukturu, do které si uložíte hodnoty parametrů. Tímto způsobem jste dosáhli toho, že jste nahradili mnoho parametrů jedním. Díky tomu jste byli schopni centralizovat data. Kolem těchto dat jste vytvořili funkce, které např. přijímaly danou strukturu jako svůj argument a vracely vám hodnotu, popřípadě upravovaly danou strukturu.

Toto je již docela zajímavý způsob, jak si zjednodušit práci s daty v jazyce C. Problém však nastává v situaci, kdy bychom chtěli mít jednu funkci (myšleno z pohledu API), která by se

chovala pouze podle toho, jakou strukturu jí pošleme jako argument. V takovém případě musíme nejprve v takové funkci zjistit o jakou strukturu se jedná a podle toho patřičně reagovat (spustit patřičný kód). Ukázka takového přístupu je se zjednodušeními ukázána níže (poznámka: nejedná se o konkrétní programovací jazyk).

```
struct {
    name = '';
} Dog;

struct {
    name = '';
} Cat;

make_sound( struct ) {
    if type( struct ) == Dog {
        print struct.name + 'Haf!';
    }
    else if type( struct ) == Cat {
        print struct.name + 'Minau!';
    }
}

struct dog = Dog( "Lassie" );
struct cat = Cat( "Tom" );

make_sound( dog ); // 'Lassie: Haf!'
make_sound( cat ); // 'Tom: Minau!'
```

V příkladu jsme vytvořili funkci **`make_sound`**, která přijímá jeden argument v podobě struktury. Danou strukturou může být buď struktura **`Dog`** nebo **`Cat`**. Evidentně po funkci **`make_sound`** chceme, aby vytiskla typický zvuk pro dané zvíře. Toho docílíme tak, že se zeptáme, jakého typu je argument a podle něj již patřičně zareagujeme. Tento přístup není obecně špatný, avšak při větším množství struktur se stává značně nepřehledným. Výše popsaný příklad se také snaží o to, aby jedna funkce byla schopna reagovat na více datových typů. Zde bychom již měli vidět, že to, co po funkci požadujeme, je de facto polymorfismus (samozřejmě ještě stále zjednodušený).

3.3 Objektový přístup k motivačnímu problému

Z předchozích kurzů víte, že OOP se skládá ze tří pilířů, kterými jsou: polymorfismus, dědičnost a zapouzdření. Věnujme se nyní primárně nejpodstatnějšímu pojmu, kterým je polymorfismus. V předchozím příkladu chceme funkčnost, kterou bychom mohli vágně popsat následovně: „Jakémukoli objektu pošleme zprávu a dostaneme adekvátní odpověď'". Toto je hodně obecný koncept, který je třeba si přiblížit. Zasílání zpráv je v moderním konceptu OOP jen voláním metod objektu, který je nám již důvěrně znám pomocí tečkové notace. OOP za nás zajistí vše ostatní. Ukažme si tedy předchozí příklad v jazyce Python.

```
class Dog(object):
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        print("{0}: Haf!".format(self.name))


class Cat(object):
    def __init__(self, name):
```

```
        self.name = name

    def make_sound(self):
        print("{0}: Minau!".format(self.name))

animals = [ Dog("Lassie"), Cat("Tom") ]

for animal in animals:
     animal.make_sound()

# Lassie: Haf!
# Tom: Minau!
```

Na chvíli ještě pomiňme syntaxi pro tvorbu tříd. Vidíme však, že oproti předchozímu příkladu je funkce **make_sound** přítomna v obou třídách a je implementována podle toho, ke které třídě patří.

Dále vidíme, že jsme si vytvořili list **animals**, který obsahuje instance tříd **Dog** a **Cat**. Nakonec projdeme tento list a na každém jeho prvku zavoláme metodu **make_sound**.

Vidíme, že při volání metody **make_sound** již do ní neposíláme žádný parametr (jako v předchozím případě strukturu, **self** teď nepočítejme). Posun od procedurálního programování k OOP je tedy již docela markantní. V cyklu **for** pak voláme na jednotlivých instancích metodu **make_sound** bez toho, aniž bychom primárně věděli, o který typ se jedná. Toto je také nejdůležitějším prvkem OOP – volání metod na objektech a nestarání se o jejich vnitřní implementaci.

Dále si v příkladu všimněme, že list **animals** obsahuje instance tříd **Dog** a **Cat**. Na rozdíl od jazyka C++ nebo Java jsme však nemuseli vytvářet společnou nadtřídu, popřípadě interface. Toto je dáno dynamickou typovostí jazyka, se kterou jsme se setkali již dříve. List mohl obsahovat integery, floaty, stringy, apod. Není tedy problém, aby obsahoval i instance námi vytvořených tříd. V dynamicky typovaných jazycích také můžeme volat jakoukoli metodu na jakémkoli objektu. Pokud však objekt neumí na metodu reagovat (metoda není vůbec implementována), je vyhozena výjimka, kterou musíme zpracovat.

3.4 Třídy a instance

Jazyk Python pracuje s objektovým modelem podobně jako C++ nebo Java. Třídy jsou obecným předpisem pro popis objektů světa. Instance jsou pak konkrétními objekty, které mají svou třídu jako předpis. Toto bychom si mohli představit i tak, že automobilka produkuje auta podle předpisu (výrobního postupu). Automobilka má tedy sadu tříd, podle kterých se ve výrobě vytvářejí automobily, které jsou již instancemi tříd.

3.5 Definice třídy a konstrukce objektu

Rozeberme si na příkladu vytvoření objektu, speciálně syntaxi konstruktoru, která je lehce odlišná od C++ nebo Javy.

```
class Battlestar(object):

    def __init__(self, name, commander): # constructor
        self.name = name                 # instance variable
        self.commander = commander

    def identify(self):                  # method
        return 'This is Battlestar {0}, \
                commanded by {1}.'.format(self.name, \
                self.commander)
```

```
galactica = Battlestar('Galactica', 'Bill Adama')
pegasus = Battlestar('Pegasus', 'Helena Cain')

print(galactica.identify())
# This is Battlestar Galactica, commanded by Bill Adama.

print(pegasus.identify())
# This is Battlestar Pegasus, commanded by Helena Cain.
```

Definice přídy je uvozena klíčovým slovem **class** následována jménem. Pokud chce definovaná třída dědit z jiných tříd, je toto uvedeno v kulatých závorkách. Naše třídy budou vždy dědit z třídy reprezentující objekt (**object**).

Konstruktor třídy je vždy metoda s názvem **__init__**, která přijímá jako svůj první argument proměnnou, která je tradičně nazývána **self**. Pak následuje seznam argumentů, které chceme konstruktoru předat. Proměnná **self** nám reprezentuje instanci podobně, jako je tomu v jazyce C++ nebo Java slovem **this**. Instanční proměnné jsou vytvořeny tak, že je přidáme do instance pomocí tečkové notace. Dostatečný příklad je uveden v konstruktoru. Pokud se nám zdá syntaxe metod nepříjemná v tom smyslu, že musíme uvádět proměnnou **self**, tak se zamyslete nad tím, kde se vezme v C++ nebo Javě proměnná **this**. Tato proměnná je ve skutečnosti do zdrojového kódu „vpašována" překladačem daného jazyka.

Metody třídy obdobně přijímají první argument v podobě proměnné **self**. Přístup k instančním proměnným je možný opět pouze s použitím oné proměnné.

Vytvoření objektu je velmi podobné jako v jiných jazycích, které známe. Jediným rozdílem je to, že nepoužíváme klíčového slova **new**. Jazyk Python (stejně jako téměř ve všech případech Java) totiž na rozdíl od jazyka C++ neumožňuje vytvořit instanci objektu na zásobníku, ale pouze na haldě.

3.6 Třídní proměnné

Instanční proměnné by pro nás již neměly být problémem. Každá instance třídy si udržuje své vlastní hodnoty v instančních proměnný. Existují však i jiné proměnné, které nazýváme třídní proměnné a ty jsou pro všechny instance společné. Tyto proměnné jsou dosti podobné globálním proměnným, na které jsme zvyklí ať už z jazyka C nebo z Pythonu. Otázkou však je, jak takové proměnné v Pythonu deklarovat a využít.

Představme si, že bychom chtěli vytvořit internetový vyhledávač. Pomiňme nyní relativně velkou složitost tohoto problému a soustřeďme se na to, jak bychom reprezentovali jednotlivé webové stránky v našem systému.

```
class Document(object):

    def __init__(self, content):
        self.content = content

    def index(self, db):
        pass
```

Třída **Document** tedy reprezentuje stránku pomocí instanční proměnné **content**. Metoda **index** pak danou stránku zaindexuje do databáze **db**. Její implementaci však nebudeme řešit. Dříve nebo později bychom však chtěli vědět, kolik dokumentů v našem systému je. Jednou z možností je zeptat se databáze. Druhou možností je pak využít třídní proměnné tak, jak je ukázáno na následujícím příkladu.

```
class Document(object):
```

```
    no_of_documents = 0

    def __init__(self, content):
        self.content = content
        Document.no_of_documents += 1

    def index(self, db):
        pass

d1 = Document("Text")
print(d1.no_of_documents) # 1

d2 = Document("Another text")
print(d1.no_of_documents) # 2
print(d2.no_of_documents) # 2

print(Document.no_of_documents) # 2
```

Pokaždé, kdy vytvoříme novou instanci dokumentu, je třídní proměnná **no_of_documents** inkrementována. Na počet dokumentů se tak můžeme zeptat jakékoli instance, protože je třídní proměnná sdílená. K třídní proměnné přistupujeme přes jméno třídy. Nemůžeme tak použít notaci přes **self**, poněvadž **self** referuje na instanci a nikoli na třídu. Nakonec k třídní proměnné můžeme přistoupit i tak, že se zeptáme přímo třídy samotné. V naší ukázce je to poslední řádek kódu.

3.7 Statické metody

Již dříve jsme si řekli, že v OOP reprezentujeme objekty reálného světa. Jejich atributy jsou instanční proměnné a pro jejich manipulaci máme metody, které jsou svázány s objekty, což je zajištěno tím, že jsou definovány u dané třídy. Někdy však potřebujeme to, abychom byli schopni volat metodu aniž bychom měli k dispozici konkrétní instanci. Chceme však využít toho, že metoda má implementovánu funkcionalitu, která se nějakým způsobem váže k dané třídě. Využijme tedy opět našeho příkladu pro vyhledávač. Nyní bychom chtěli vědět průměrnou délku stránek, které indexujeme.

```
class Document(object):

    no_of_documents = 0
    total_length = 0

    def __init__(self, content):
        self.content = content
        Document.no_of_documents += 1
        Document.total_length += len(self.content)

    def index(self, db):
        pass

    @staticmethod
    def get_average_length():
        return Document.total_length / Document.no_of_documents

d1 = Document("Text")
d2 = Document("Another text")

print(Document.get_average_length()) # 8
print(d1.get_average_length()) # 8
```

Na první pohled vidíme oproti běžné metodě dva rozdíly. Prvním je to, že metoda je uvozena dekorátorem **`@staticmethod`**. Tento dekorátor nám nezajistí ani tak to, že bude metoda statická, jako spíše to, že ji jako statickou bude možno volat na intanci, což je v jiných jazycích problematické. Toto je ukázáno na posledním řádku příkladu. **Podstatným rozdílem** je však to, že metoda nemá jako svůj první argument proměnnou **`self`**. Je tedy explicitně řečeno, že metoda nemůže být volána na objektu (již víme, že objekt je předán jako argument metody v podobě proměnné **`self`**). Dále je samozřejmé, že statická metoda nemůže přistupovat k instančním proměnný. Logiku tohoto tvrzení ponechme na čtenáři.

Cvičení: Zkuste odstranit dekorátor a spustit příklad, případně proveďte další úpravy.

3.8 Vlastnosti (Properties)

Další vlastností OOP je zapouzdření. Jazyk Python nepodporuje skrývání instančních proměnný tak, jak to například dovoluje Java pomocí modifikátorů **`private`** nebo **`protected`**. Naproti tomu je v Pythonu sada doporučení, jak se k zapouzdření zachovat. S takovýmto přístupem se můžeme setkat i u jiných jazyků. Vzpomeňme si, že je třeba při ředění kyseliny nalévat kyselinu do vody a nikoli opačně. Toto je také doporučení a nikdo nám nebrání to udělat obráceně. Následky však mohou být dosti nepříjemné.

Přímá manipulace s atributy třídy je sice možná, ale pokud nejsme přímými autory těchto tříd, je dosti pravděpodobné, že takovouto neopatrnou manipulací způsobíme nekonzistenci dat. Standardní způsob manipulace s atributy tříd je přes metody, které jsou přímo určeny k zápisu nebo čtení hodnot těchto atributů. Způsob práce je podobný jako v jazyce C++ nebo Java. Ukažme si tedy příklad.

```
class Document(object):

    no_of_documents = 0
    total_length = 0

    def __init__(self, content):
        self.content = content
        Document.no_of_documents += 1
        Document.total_length += len(self.content)

    def set_content(self, content):
        Document.total_length -= len(self.content)
        self.content = content
        Document.total_length += len(self.content)

    def get_content(self):
        return self.content

    def index(self, db):
        pass

    @staticmethod
    def get_average_length():
        return Document.total_length / Document.no_of_documents

d1 = Document("Text")
d2 = Document("Another text")

print(Document.get_average_length()) # 8
```

```
d2.set_content("This is a bit longer text")

print(Document.get_average_length()) # 14
```

V metodě **set_content** do instanční proměnné **content** uložíme nový obsah stránky. Je však také nutné přepočítat průměrnou délku dokumentů, které máme v našem systému. Toto nám metoda **set_content** zajistí. Pokud bychom však přímo manipulovali s obsahem proměnné **content**, toto by zajisté zajištěno nebylo. Metoda **get_content** pak jen vrací obsah proměnné **content**. Na tomto příkladu bychom měli jasně vidět proč je vhodné používat tzv. setry a getry.

Cvičení: Jako cvičení si zkuste přímou manipulaci s proměnnými a zjistěte, kdy nastává nekonzistence dat a jak se projevuje.

V Pythonu však můžeme ještě využít tzv. **properties**. Ty se používají u atributů, které jsou pro nás zajímavé a vytvářejí iluzi přímé manipulace. Následující ukázka ilustruje jejich použití.

```
class Document(object):

    no_of_documents = 0
    total_length = 0

    def __init__(self, content):
        self._content = content
        Document.no_of_documents += 1
        Document.total_length += len(self.content)

    def set_content(self, content):
        Document.total_length -= len(self._content)
        self._content = content
        Document.total_length += len(self._content)

    def get_content(self):
        return self._content

    def index(self, db):
        pass

    @staticmethod
    def get_average_length():
        return Document.total_length / Document.no_of_documents

    content = property(get_content, set_content)

d1 = Document("Text")
d2 = Document("Another text")

print(Document.get_average_length()) # 8

d2.content = "This is a bit longer text"

print(d2.content) # "This is a bit longer text"
print(Document.get_average_length()) # 14
```

Naši původní proměnnou **content** jsme přejmenovali na **_content**, což výsledek nijak neovlivňuje (toto jsme museli udělat proto, že jinak by docházelo ke kolizi jmen mezi property a proměnnou, která by vyústila v nekonečnou rekurzi, o čemž se můžete přesvědčit, pokud si příklad

náležitě upravíte). Nakonec jsme pomocí funkce **property** řekli, že chceme zpřístupnit data pod názvem **content**, která se čtou a nastavují pomocí funkcí **get_content** a **set_content**. Ve výsledku můžeme jakoby přímo zapisovat do proměnné **content** a přitom je zajištěna konzistence dat. Další výhodou je to, že pokud později změníme set nebo get metody, kód který používá properties se nemusí měnit. Tím máme pěkně zajištěnu dopřednou kompatibilitu.

3.9 Operátory

V průběhu naší práce jsme se setkali s operátory a nepřišlo nám to nijak složité. Např. jsme mohli sčítat čísla pomocí operátoru **+** nebo spojovat řetězce tím samým operátorem. Problémem těchto operátorů však je to, že je tvůrci naučili pracovat pouze s vestavěnými typy, např. **int**, **str**, apod.

Pokud vytvoříme svou vlastní třídu, ta se stává novým datovým typem (toto obecně nazýváme abstraktní datové typy). Problém však je, že nyní standardní operátory nemohou tušit, jak s našimi vlastními typy pracovat. A právě k tomu nám slouží přetěžování operátorů. Seznam operátorů, které můžeme přetížit je relativně dlouhý, a proto se odkážeme na dokumentaci [4]. Nám aktuálně postačí to, že operátory jsou v třídách realizovány jako metody, které vždy začínají a končí znaky dvou podtržítek __. Mezi nimi je pak jméno operátoru. Pozorný čtenář již tuší, že konstruktor je také jen operátorem.

Jako příklad přetížení operátoru si ukážeme, jak naimplementovat třídu **Vector**, která reprezentuje vektor libovolné délky. Budeme chtít sčítat dva různé vektory a také budeme chtít tisknout jejich obsah pomocí příkazu **print**.

```
class Vector(object):
    def __init__(self, vec):
        #warning: vec will be tuple, immutable
        self.vec = vec

    def __add__(self, other):
        vec = []
        for no, item in enumerate(self.vec):
            vec.append(item + other.vec[no])

        return Vector(tuple(vec))

    def __str__(self):
        #content = ", ".join([str(i) for i in self.vec])
        #return "({})".format(content)
        return str(self.vec)

v1 = Vector((1, 2, 3, 4))
print(v1)

v2 = Vector((5, 6, 7, 8))
print(v2)

v3 = v1 + v2
print(v3)
```

V ukázce vidíme, že operátor **+** je realizován metodou **__add__**. Chceme přeci realizovat operaci **x + y**, proto **x** bude instance **self** a **y** bude argument **other.** V této metodě provedeme příslušnou operaci sečtení dvou vektorů a vrátíme novou instanci třídy **Vector**, která obsahuje výsledek.

Dalším přetíženým operátorem je **__str__**, který nám vrací řetězec. Podrobně jako jsme používali funkci **int** nebo **float** pro získání příslušné hodnoty, i funkce **str** nám vrátí řetězec,

který reprezentuje daný objekt. Pro náš typ vektoru chceme, aby se nám vrátily hodnoty vektoru v kulatých závorkách. Toho jsme docílili relativně jednoduchým trikem, kdy naše interní reprezentace vektoru je tuple, který již odpovídá naší zvolené reprezentaci. Pro jistotu však ještě v komentáři uvádíme plnou implementaci cílového chování metody. Tento operátor je vždy zavolán funkcí **`str`** a tato funkce je vždy volána při volání příkazu **`print`**. Proto není přímé použití na první pohled plně viditelné.

Pomocí operátorů je možno značně ovlivnit chování našich typů. Na rozdíl od jiných jazyků jsme schopni ovlivňovat i průběh volání metod nebo průběh konstrukce objektů. Toto je však již docela pokročilé téma, které si necháme jako domácí cvičení.

3.10 Kompozice

Vytváření samotných abstraktních datových typů (tříd) bez návaznosti na okolí by asi nebylo příliš zajímavé. Podívejme se tedy na to, jakým způsobem provádět kompozici a hlavně, kdy ji nahradit dědičností. Častokrát se totiž stává, že se rozhodneme špatně, což pak nese neblahé následky.

Kompozici používáme pro spojení několika komponent do jednoho celku. Pokud si nejsme zcela jisti, zdali máme použít kompozici, stačí se zeptat: **Je X součástí Y?** Pokud je odpověď kladná, jedná se problém kompozice. Zkusme si tedy opět vylepšit náš vyhledávač reprezentovaný třídou **`SearchEngine`**. Tentokráte bychom chtěli vytvořit systém, ve kterém budou obsaženy dokumenty, které jsou reprezentovány třídou **`Document`**, kterou jsme si již uvedli. V příkladu opět pomiňme extrémní zjednodušení, která musíme zavést.

```
class SearchEngine(object):

    def __init__(self):
        self.documents = []

    def add_document(self, document):
        self.documents.append(document)

    def get_documents(self):
        return self.documents

    def search(self, query):
        for doc in self.documents:
            if doc.content.find(query) >= 0:
                return doc

se = SearchEngine()
se.add_document(Document("Text"))
se.add_document(Document("Another text"))

for doc in se.get_documents():
    print doc.get_content()

print(se.search("text").content)
```

Na začátku jsme si řekli, že chceme modelovat systém, který obsahuje dokumenty. Dokumenty jsou tedy součástí většího celku. Je tedy na místě využít kompozice. V konstruktoru třídy **`SearchEngine`** si vytvoříme instanční proměnnou typu **`list`**, která bude obsahovat všechny dokumenty, které bude náš vyhledávač indexovat. V metodě **`add_document`** pak do interního listu přidáme referenci na předaný dokument. Metoda **`search`** pak vyhledá patřičný dokument podle zadaného obsahu (samozřejmě je to naivní implementace, která navíc neošetřuje chybové stavy).

Na tomto příkladu tedy vidíme, jak provést kompozici dvou komponent. Nic nám samozřejmě nebrání komponovat více systémů do sebe. Nezapomeňme však, že je třeba dodržovat jistá pravidla, o kterých se dozvíte v dalších předmětech jako je Softwarové inženýrství.

3.11 Dědičnost

Poslední základní vlastností OOP je dědičnost, kterou také často nazýváme specializací. Podobně jako u kompozice, pokud si nejsme jisti, zda máme problém implementovat pomocí dědičnosti, můžeme se zeptat: **X je Y?** Je-li odpověď kladná, jedná se o dědičnost. Častokrát také dědičnost používáme pro převzetí kódu z nadtřídy, což je ovšem opět specializací.

Ukažme si tedy příklad, kdy chceme implementovat informační systém obsahující studenty. Entita studenta je člověkem, tedy můžeme vytvořit nejprve obecnou třídu **`Person`**, ze které bude třída **`Student`** dědit.

```
class Person(object):

    def __init__(self, firstname, surname):
        self.firstname = firstname
        self.surname = surname

    def get_firstname(self):
        return self.firstname

    def get_surname(self):
        return self.surname

    def __str__(self):
        return " ".join((self.firstname, self.surname))


class Student(Person):

    next_id = 0

    def __init__(self, firstname, surname):
        Person.__init__(self, firstname, surname)
        Student.next_id += 1
        self.edison_id = Student.next_id

    def get_id(self):
        return self.edison_id


s1 = Student("Sean", "Connery")
print("ID:", s1.get_id(), "Student name:", s1)
# ID: 1 Student name: Sean Connery
print(s1.get_firstname(), s1.get_surname()) # Sean Connery

s2 = Student("James", "Bond")
print("ID:", s2.get_id(), "Student name:", s2)
# ID: 2 Student name: James Bond
```

Ve třídě **`Person`** jsme naimplementovali základní funkcionalitu. Třída **`Student`** dědí z **`Person`**. Navíc přidává to, že je každé instanci přiřazeno jednoznačné **`edison_id`**, kterým se identifikuje v našem univerzitním IS. Také je implementována metoda **`get_id`**, která daný identifikátor vrací. Třída **`Student`** je tedy specializovanou třídou odvozenou z třídy **`Person`**. Důležitým prvkem je konstruktor odvozené třídy. Ten jako první vždy volá konstruktor třídy nadřazené jako

statickou metodu, kde jako instanci vloží sebe sama. Toto je jedno z mála míst, kde přímo voláme operátor, jinak se tohoto volání snažíme vyvarovat.

Cvičení: Do třídy **`Person`** doplňte metody pro práci s věkem člověka pomocí modulu **`datetime`**. Poté přetěžte takový oprátor, abyste mohli jednoduše zjistit, zda je člověk nebo student starší než nějaký jiný. Také zkuste mezi sebou porovnat instance studenta a člověka.

Kapitola 4

Standardní knihovna jazyka Python

Standardní knihovna jazyka Python obsahuje velké množství užitečných funkcí. Vysvětleme si nejprve co je a k čemu takováto knihovna slouží. Programovací jazyky obecně slouží ke konstrukci algoritmů, které poté jejich kompilátor nebo interpret převede do spustitelné podoby, která daný algoritmus vykonává. Programovací jazyk obsahuje jen velice málo funkcí, které programátor může použít. Standardní knihovna je ihned dispozici po nainstalování příslušného programovacího jazyka. Většinou ji stačí k našemu programu jednoduše připojit.

Knihovny slouží jako úložná místa s již vytvořenými funkcemi, které programátorovi usnadňují každodenní práci. Velkým skokem kupředu v oblasti standardní knihovny se stal v minulosti programovací jazyk Java. Ten oproti jazyku C s sebou přinesl velkou sadu funkcí, které jinak museli programátoři jazyka C pracně vyhledávat, popřípadě vytvářet sami.

Programovací jazyk Python jde v ohledu ke standardní knihovně v podobném stylu jako jazyk Java. Python disponuje širokou škálou knihoven od práce s regulárními výrazy, přes práci se sítí, až po webové služby. Příjemné též je to, že i vaše vlastní knihovna se může stát součástí standardní knihovny Pythonu, pokud bude dostatečně užitečná, kvalitně napsaná a zdokumentovaná.

4.1 Použití standardní knihovny

Na začátek je třeba si říci, jak se v jazyce Python používá standardní knihovna. Instalace interpretu Pythonu v sobě obsahuje mnoho modulů (knihoven), které se k našemu skriptu dají připojit pomocí klíčového slova import, následovaného jménem modulu. Tímto příkazem se k našemu skriptu připojí modul a dále můžeme používat funkce, které jsou v něm obsaženy pomocí syntaxe `jméno_modulu.funkce()`. Existují i jiné způsoby práce s funkcemi z modulu, tato je však jedna z nejpřehlednějších. Uveďme si krátký příklad, kdy pomocí modulu **`os`** ze standardní knihovny vytiskneme obsah aktuálního adresáře:

```
>>> import os
>>> os.listdir(".")
["text.tex", "text.pdf", "text.aux", "text.toc"]
```

Modul **`os`** obsahuje mnoho užitečných funkcí pro práci s operačním systémem. V našem případě chceme vytisknout obsah aktuálního adresáře (tedy soubory a adresáře v něm obsažené). Funkce, která slouží pro výpis obsahu adresáře se jmenuje **`listdir()`** a její parametr je cesta k adresáři, ve kterém chceme tuto operaci provést. Aktuální adresář tedy bude řetězec `"."`. K funkci **`listdir()`** pak přistoupíme jednoduše pomocí příkazu **`os.listdir(".")`**.

4.2 Vzdálené volání funkcí pomocí XML-RPC

Vzdálené volání funkcí slouží, jak už název napovídá, k volání funkcí na nějakém vzdáleném místě (v tomto případě počítači připojenému do počítačové sítě). V současné chvíli je nám zcela jasné, jak provádět volání funkcí v rámci našeho skriptu, volání funkcí z modulů nebo volání metod tříd.

V praxi však často dochází k tomu, že náš softwarový produkt je umístěn na více počítačích, které jsou spolu propojeny počítačovou sítí. Každý takovýto počítač je vybaven programy, které složí k jinému účelu. Tyto počítače by však měly ve výsledku tvořit kompaktní celek. Z tohoto důvodu spolu musí komunikovat (předávat si zprávy).

Jedním ze způsobů komunikace je tzv. vzdálené volaní funkcí (nebo procedur). V princip se jedná o komunikaci pomocí architektury klient/server, kde se komunikující programy dělí na klienta a server.

Klient je program, který bude požadovat vykonání funkce na vzdáleném serveru. Při požadavku serveru předá potřebné parametry a po vykonání vzdálené funkce serverem také obdrží výsledek.

Server je program, který očekává požadavky od klientů. Pokud takový požadavek obdrží, vykoná jej prostřednictvím volaní své vnitřní funkce a výsledek pak předá klientovi.

My se konkrétně budeme zabývat technologií XML-RPC, tedy vzdáleným voláním funkcí za pomoci XML. Na první pohled to může vypadat velice složitě, avšak díky standardní knihovně Pythonu bude práce s touto technologií velice jednoduchá. Pro praktické použití nebudeme potřebovat ani dva počítače, pokud budou server i klient komunikovat přes lokální síťové rozhraní počítače.

Nyní se pokusme vytvořit jednoduchou kalkulačku pomocí vzdáleného volání funkcí. Server bude poskytovat funkce pro sčítání a odčítání čísel, klient bude tyto funkce využívat.

```
import SimpleXMLRPCServer

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

# simple server
server = SimpleXMLRPCServer.SimpleXMLRPCServer(\
("localhost", 8000))
print "Listening at the port 8000..."

server.register_function(add, 'plus')
server.register_function(subtract, 'minus')

server.serve_forever()
```

Nyní si ve stručnosti projděme zdrojový kód serveru. Standardní knihovna Pythonu obsahuje modul **`SimpleXMLRPCServer`**, který poskytuje funkcionalitu serveru pro vzdálená volání. Na prvním řádku si tento modul připojíme. Dále definuje dvě funkce pro sčítání a odečítání dvou čísel. Zbylá část kódu jen inicializuje server. Nejprve si vytvoříme objekt serveru, který bude naslouchat na lokálním síťovém rozhraní počítače (konkrétně na portu **`8000`**). Dále je nutné tzv. registrovat funkce, které bude server navenek poskytovat klientům. Jak je vidět z výpisu. Funkce **`add`** a **`subtract`** se budou navenek tvářit jako funkce **`plus`** a **`minus`**. Toto je užitečná metoda jak zachovat vnitřní konvenci pojmenovávání funkcí a přitom je navenek propagovat pod jiným jménem. Nakonec náš server spustíme a budeme čekat na volání od klientů.

Všimněme si důležitého faktu, že funkce které chceme zpřístupnit pro vzdálené volání jsou zcela totožné z hlediska syntaxe s funkcemi, které již dobře známe.

Klientská část vypadá následovně:

```
import xmlrpclib

remote_server = xmlrpclib.ServerProxy("http://localhost:8000/")
print remote_server.plus(7, 3) # 10
print remote_server.minus(7, 3) # 4
```

Pro klienta využijeme modul **xmlrpclib**, který obsahuje funkce pro zpřístupnění vzdáleného volání funkcí na nějakém serveru. Pro aktuální zpřístupnění funkcí na serveru si nejprve vytvoříme objekt serveru zavoláním funkce **ServerProxy** z modulu **xmlrpclib**, kde parametrem bude síťová adresa serveru (v našem případě lokální adresa a port použitý v kódu serveru). Pomocí toho to objektu pak můžeme volat funkce definované na serveru tak, jako by byly definované uvnitř nějakého objektu, který jsme si vytvořili sami. V našem případě tedy příkaz **remote_server.plus(7, 3)** zavolá funkci **add** na serveru.

Jak je z tohoto příkladu vidět, vzdálené volání v Pythonu za použití standardní knihovny je velice jednoduché. Při tom si všimněte, že jsme se nikde nesetkali s XML, které je použito ve vnitřní implementaci celého mechanizmu.

Cvičení: Vyzkoušejte si uvedený příklad a doplňte server o další funkce.

Cvičení: Naimplementujte server, který bude provádět jednoduché výpočetní operace. Při implementaci serveru použijte třídu, jejíž metody budou přístupné klientům. Všimněte si rozdílu v registraci funkcí na serveru. V dokumentaci k jazyku Python je tento případ dobře popsán[1].

[1] http://docs.python.org/2/library/simplexmlrpcserver.html#simplexmlrpcserver-example

Kapitola 5

Literatura

[1] Harms, D., McDonald, K.: Začínáme programovat v jazyce Python. Computer Press, a.s.

[2] Švec, J.: Učebnice jazyka Python (aneb Létající cirkus).
URL: http://www.pythondocs.ic.cz/tut/tut.html

[3] Oficiální webová stránka jazyka Python.
URL: http://www.python.org

[4] Seznam oprátorů a jejich použití.
URL: http://docs.python.org/reference/datamodel.html#special-method-names

Rejstřík