O čem si budeme povídat?

- Jednotlivé příkazy.
- Použití Pythonu jako kalkulačky.
- Používání závorek k dosažení správného výsledku.
- Použití formátovacího řetězce pro tisk složitějšího výstupu.
- Nakonec si ukážeme, jak můžeme Python ukončit zevnitř programu.

Nejjednodušší program, který můžete napsat, se skládá z jednoduché posloupnosti příkazů. Nejjednodušší posloupnost je taková, která obsahuje jediný příkaz. Některé příkazy si teď vyzkoušíme. Nadpis bude říkat, co máme napsat za vyzývacími znaky překladače '>>> ' a následující odstavec nám objasní, co se stane.

```
>>> print 'Ahoj, vy tam!'
```

Příkaz print zajistí, aby nám Python zobrazil výsledek na orazovku. V tomto případě se tiskne posloupnost znaků A, h, o, j, ,, , v, y, , t, a, m, !. Takové posloupnosti znaků se v programátorských kruzích říká *řetězec znaků*, *znakový řetězec* nebo prostě jen *řetězec*.

Řetězec zapisujeme mezi uvozovky. V Pythonu můžeme použít buď jednoduché uvozovky (apostrofy, tak jako ve výše uvedeném případě), nebo dvojité uvozovky: "toto je řetězec". Díky tomu můžeme do řetězce vložit jeden ze zmíněných znaků, pokud pro ohraničení řetězce použijeme druhý z nich. Může se námto hodit například v případech, kdy věta obsahuje apostrofy (v anglických větách — viz ukázka — je tento jev poměrně častý):

```
>>> print "Monty Python's Flying Circus has a ' within it..."
```

Ale tisknout nemusíme jen znaky:

```
>>> print 6 + 5
```

Zde jsme vytiskli výsledek aritmetické operace — sečetli jsme šest a pět. Python rozpoznal zapsaná čísla a znak plus a zajistil sečtení čísel. Zobrazí se výsledek součtu (tedy 11).

Python tedy můžeme využít jako kapesní kalkulačku. Vyzkoušejte si pár dalších součtů. Vyzkoušejte si také další aritmetické operátory:

- odčítání (-)
- násobení (*)
- dělení (/)

Vyzkoušejte si i kombinované výrazy, jako například:

```
>>> print ((8 * 4) + (7 - 3)) / (2 + 4)
```

Povšimněte si, jak byly pro vymezení podvýrazů použity závorky. Co by se stalo, kdybychom napsali stejný výraz bez závorek? Python preferuje násobení a dělení před sčítáním a odčítáním a proto by tedy uskutečnil tyto operace jako první. V matematice je to běžné, ale u programovacích jazyků to nemusí platit vždy. Všechny programovací jazyky definují pravidla, která určují posloupnost vyhodnocování operací. Tato pravidla jsou známa pod pojmem *priorita operátorů*. Pokud chcete přesně zjistit, jak bude výraz v konkrétním jazyce vyhodnocován, musíte se podívat do jeho referenční příručky. U jazyka Python je tomu většinou tak, jak bychom očekávali na základě logiky a intuice. Ale občas je to trochu jinak… Pokud pracujeme s dlouhými výrazy a chceme si být jistí, že dostaneme výsledek podle našeho očekávání, pak bychom obecně měli použít závorek.

Další poznámka:

```
>>> print 5/2
```

výsledkem je celé číslo (integer) — v našem případě 2. Je to dáno tím, že Python zjistil, že jste ve výrazu uvedli pouze celá čísla a předpokládá, že chcete získat výsledek jako číslo stejného typu. (Jinými slovy, operátor dělení má celočíselné operandy, a proto Python předpokládá, že máte na mysli celočíselné dělení.) Pokud budeme chtít výsledek v podobě desetinného čísla, zapište alespoň jeden z operandů v podobě desetinného čísla:

```
>>> print 5/2.0
2.5
```

Python našel zápis čísla 2.0 a zjistil, že nám práce s desetinnými čísly nečiní problémy, takže nám vrátí výsledek v podobě desetinného čísla. (Desetinná čísla se v souvislosti s programováním označují také jako*reálná* čísla nebo čísla v *plovoucí řádové čárce*.) V posledních verzích Pythonu můžete popsané chování změnit tak, aby výsledkem dělení bylo vždy reálné číslo. Dosáhnete toho přidáním následujícího řádku na začátek vašeho programu (povšimněte si, že kolem slova future píšeme z obou stran dvojici znaků podtržení):

```
>>> from __future__ import division
```

Dá se očekávat, že tento způsob dělení se v některé z budoucích verzí systému Python stane standardem. Prozatím musíme Pythonu říci, že toto chování chceme zapnout.

Pokud byste chtěli zůstat u počítání v celých číslech, pak můžete získat hodnotu zbytku po celočíselném dělení pomocí operátoru % (znak procenta). Python pak zobrazí tento zbytek.

```
>>> print 7/2
3
>>> print 7%2
1
>>> print 7%4
3
```

Operátor % je znám pod názvem *modulo* nebo mod a v jiných jazycích se často zapisuje jako MOD nebo podobně. Experimentujte a brzy vám bude jasné, o co jde.

```
>>> print 'Soucet je: ', 23+45
```

Již jsme si předvedli, že dokážeme tisknout řetězce a čísla. Nyní jsme oba případy zkombinovali do jednoho příkazu pro tisk s tím, že jsme je oddělili čáskou. Tento rys můžeme dále rozšířit o další trik jazyka Python pro výstup dat, který se nazývá *formátovací řetězec*:

```
>>> print "Soucet cisla %d a cisla %d je: %d" % (7,18,7+18)
```

V tomto příkazu obsahuje formátovací řetězec uvnitř značky uvozené znakem '%' (procento). Písmeno 'd' za % překladači jazyka Python říká, že zde má být umístěno dekadické číslo (tj. celé číslo vyjádřené v desítkové řádové soustavě). Hodnoty, které mají být dosazeny místo značek uvnitř řetězce, jsou získány z hodnot uvnitř výrazu v závorkách, který následuje samostatně umístěný znak % za formátovacím řetězcem.

Pro značky ve formátovacím řetězci lze použít i kombinace % s jinými znaky. Některé z nich jsou uvedeny níže:

- %s - pro řetězec
- %x - pro zápis čísla v hexadecimálním tvaru
- %0.2f - pro zápis reálného čísla na dvě desetinná místa
- %04d - pro zápis celého čísla doplněného zleva nulami na 4 pozice

V příručce jazyka Python jich naleznete mnohem více…

Ve skutečnosti můžete v jazyce Python příkazem print vytisknout libovolný *objekt*. Někdy výsledek neodpovídá tomu, co byste očekávali (někdy se zobrazí jen popis, který říká, o jaký druh objektu jde), ale objekt můžete vytisknout vždy.

```
>>> import sys
```

Tento příkaz je nějaký podivný. Pokud jste jej vyzkoušíte, uvidíte, že zjevně nedělá vůbec nic. Ale ve skutečnosti to není pravda. Abychom rozuměli tomu, co se děje, potřebujeme nahlédnout do architektury jazyka Python. (Pokud programujete v jiném jazyce, mějte chvíli strpení. Určitě používáte nějaký podobný mechanismus.).

Když Python spustíte, máte k dispozici skupinu příkazů, které jsou označovány jako *zabudované* (built-in), protože jsou zabudovány přímo do jádra překladače jazyka Python. Ale Python může rozšiřovat seznam dostupných příkazů tím, že k jádru přičleňuje rozšiřující moduly. Trochu to připomíná situaci, kdy si ve vašem oblíbeném obchodě *Udělej si sám* koupíte nový nástroj a přidáte jej do své bedny s nářadím. Nástroj zde představuj část příkazu sys a operace import jej přidá k dosud užívaným nástrojům.

Ve skutečnosti tento příkaz zpřístupní celou skupinu nových 'nástrojů' v podobě dalších příkazů jazyka Python, které jsou ve výše uvedeném případě definovány v souboru 'sys.py'. Toto je tedy způsob, jakým lze Python rozšiřovat, aby uměl provádět všechny možné důmyslné věci, které nejsou zabudovány do základního systému. Můžete dokonce vytvořit své vlastní *moduly*, naimportovat je (příkazem import) a potom je používat naprosto stejným způsobem, jako se používají moduly, které jsou dodávané s instalací jazyka Python.

Takže jak vlastně máme tyto nové nástroje používat?

```
>>> sys.exit()
```

Ale! Co se to stalo? Prostě jsme spustili příkaz exit, který je definován v modulu sys. Tento příkaz způsobí ukončení práce s interpretem jazyka Python. (**Poznámka:** Obvykle Python ukončujeme tím, že za vyzývací posloupnost >>> zadáme znak EOF (End Of File — konec souboru) — v systému DOS je to znak Ctrl-Z, v systému Unix je to znak Ctrl-D.)

Všimněte si, že za příkazem exit jsou uvedeny dvě závorky. To proto, že se jedná o *funkci* definovanou v sys. A pokud v jazyce Python voláme funkci, musíme závorky psát, i když v nich není nic uvedeno.

Zkuste napsat sys.exit bez závorek. Místo toho, aby Python funkci provedl, zareaguje pouze výpisem sdělení, že exit je funkce.

A nakonec bychom si měli povšimnout, že poslední dva příkazy jsou ve skutečnosti užitečné pouze v kombinaci. To znamená, že pokud chceme Python ukončit jinak, než napsáním EOF, musíme napsat:

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

A to je přece posloupnost dvou příkazů! Takže jsme se zase dostali o krůček blíže k opravdovému programování…

A teď v jazyce JavaScript

Narozdíl od Pythonu nemůžeme při použití JavaScript zkoušet příkazy tak, že bychom je jednoduše psali a pozorovali okamžité důsledky jejich provedení. Příkazy ale můžeme vepsat do html souboru a ten zobrazit v prohlížeči. Tím zjistíme, jak fungují:

```
<html><body>
<script type="text/javascript">
document.write('Hello there!<br>');
document.write("Monty Python's Flying Circus has a ' within it<br>");
document.write(6+5);
document.write("<br>");
document.write(  ((8 * 4) + (7 - 3)) / (2 + 4)  );
document.write("<br>");
document.write( 5/2 );
document.write("<br>");
document.write( 5 % 2 );
</script>
</body></html>
```

Výsledek by měl vypadat takto:

```
Hello there!
Monty Python's Flying Circus has a ' within it
11
6
2.5
1
```

Povšimněte si, že přechody na nové řádky si musíme vynutit zápisem
. Je to dáno tím, že JavaScript produkuje svůj výstup v podobě HTML a formát HTML definuje, že se zobrazované řádky mají zalamovat na takovou šířku, jakou dovolí okno prohlížeče. Abychom vynutili přechod na nový řádek, musíme použít příslušný HTML symbol, kterým je právě
 (br z anglického *break* [brejk] — zalomit).

A ještě v jazyce VBScript…

Podobně jako v případě JavaScript musíme i v případě VBScript vepsat příkazy do souboru a otevřít jej v prohlížeči. V jazyce VBScript budou mít příkazy z předchozího příkladu následující podobu:

```
<html><body>
<script type="text/vbscript">
MsgBox "Hello There!"
MsgBox "Monty Python's Flying Circus has a ' in it"
MsgBox 6 + 5
MsgBox ((8 * 4) + (7 - 3)) / (2 + 4)
MsgBox 5/2
MsgBox 5 MOD 2
```

```
</script>
</body></html>
```

Výstupem by mělo být zobrazení dialogových oken, z nichž každé zobrazuje výstup z jednoho řádku programu.
Zmiňme se ještě o jedné věci. Jazyk VBScript neumožňuje zápis řetězců v apostrofech (v dalších tématech si ozřejmíme důvod). Uvnitř řetězce, který musí být obklopen uvozovkami, můžeme apostrof normálně použít. Pokud ovšem chceme do řetězce vložit uvozovky, musíme použít funkci Chr(), která pro zadaný ASCII kód znaku vrací příslušný znak. Výsledný kód není zrovna přehledný, ale následující příklad by měl ukázat, jak na to:

```
<script type="text/vbscript">
Dim uv
uv = Chr(34)
MsgBox uv & "Vypadni!" & uv & " zařval"
</script>
```

Pokud potřebujete zjistit ASCII kód pro libovolný znak, můžete ve Windows použít aplikaci *Mapa znaků*. Můžete se také podívat na úvodní stránku www.asciitable.com, kde vás bude (pro náš účel) zajímat dekadická hodnota kódu. Můžete použít i následující kousek programu zapsaný v JavaScript a případně nahradit znak uvozovek tím znakem, jehož kód vás zajímá:

```
<script type="text/javascript">
var code, chr = '"';
code = chr.charCodeAt(0);
document.write("<br>ASCII kód znaku  " + chr + "  je " + code);
</script>
```

Nemusíte si lámat hlavu s tím, co to znamená. Ještě se k tomu dostaneme. Prozatím skript prostě *použijte* v situaci, kdy budete chtít zjistit ASCII kód hledaného znaku.
Náš první pohled na programování máme za sebou. Ani to moc nebolelo, že ne? Než se pustíme dál, budeme se muset podívat na *suroviny* využívané při programování — zejména na data a na to, co s nimi můžeme dělat.

Zapamatujte si

- Program se může skládat dokonce i z jediného příkazu.
- Python provádí matematické operace *téměř* tím způsobem, který byste očekávali.
- Pokud chcete obdržet výsledek v podobě desetinného čísla, musíte použít desetinné číslo i na vstupu.
- Text a čísla můžete kombinovat při použití formátovacího řetězce a formátovacího operátoru %.
- Činnost lze ukončit napsáním import sys; sys.exit().

Data, datové typy a proměnné

O čem si budeme povídat?

- Co jsou data.
- Co jsou proměnné.
- Datové typy a co se s nimi dá dělat.
- Definování našich vlastních datových typů.

Úvod

Při jakékoliv kreativní tvorbě potřebujeme tři základní položky: nástroje, suroviny a techniky (postupy). Když si například budeme chtít něco namalovat, našimi nástroji budou štětce, tužky a paleta. Technikami budeme rozumět malování vlhkým do vlhkého, rozmývání, sprejování a podobně. A konečně surovinami budou barvy, papír a voda. Při programování je to podobné. Našimi nástroji jsou programovací jazyky, operační systémy a hardware. Za techniky budeme považovat používání programových konstrukcí, o kterých jsme se zmínili v předchozí části (Koncepty — Co je to *programování*? — Společné vlastnosti programů). Surovinami budou data, se kterými budeme manipulovat. V této kapitole se zaměříme právě na tyto suroviny.
Tato kapitola je docela dlouhá a ze své podstaty poněkud nezáživná. Dobrá zpráva je, že si ji nemusíte přečíst celou najednou. Kapitola začíná pohledem na nezákladnější datové typy, které máme k dispozici. Dále si řekneme něco o tom, jak zacházet s kolekcemi položek. A nakonec se podíváme na některá pokročilejší témata. Čtení kapitoly můžete přerušit po přečtení části o kolekcích. Poté si můžete přečíst několik dalších kapitol a dočíst ji můžete v okamžiku, kdy začneme používat složitější věci.

Data

Pojem *data* je jedním z těch, který lidé často používají, ale málokdo rozumí jeho přesnému významu. V mém slovníku je pojem data definován takto:
"fakta nebo údaje, ze kterých je možné odvodit závěry; informace"
Příliš nám to sice nepomůže, ale tuto definici můžeme použít alespoň jako odrazový můstek. Uvidíme, zda se věci neozřejmí, když se podíváme na to, jak je pojem data používán v oblasti programování. Data jsou oním *materiálem*, surovými informacemi, se kterými manipuluje váš program. Bez dat nemůže program provádět žádnou užitečnou činnost. Programy mohou s daty manipulovat různými způsoby. Často to závisí na *typu* dat. S každým datovým typem je spojena skupina *operací* — tj. činností, které lze s daty daného typu provést. U čísel jsme si například ukázali, že je můžeme sčítat. Sčítání je tedy operace, kterou můžeme použít pro data typu číslo. Datových typů může být velké množství. Postupně se podíváme na nejběžnější z nich a na operace, které lze pro ně použít.

Proměnné

Data jsou uložena v paměti vašeho počítače. Můžeme ji přirovnat k velké stěně plné skříněk, která se na poštách používá k třídění poštovních zásilek. Dopis můžete strčit do libovolné skříňky, ale pokud by na těchto skříňkách nebyly štítky s konkrétní cílovou adresou, nedávalo by jejich použití smysl. Proměnné přestavují ty popisné štítky[1] na skříňkách a ony skříňky jsou vytvořeny v paměti vašeho počítače.
Zatím tedy víme, jak data vypadají. To je sice pěkné, ale k tomu, abychom s nimi mohli manipulovat, musíme být schopni se k nim dostat. Právě k tomuto účelu se používají proměnné. V programátorské terminologii říkáme, že vytváříme *instance* datových typů a přiřazujeme je do proměnných[2]. Proměnná představuje *odkaz*(nebo jinými slovy *referenci*) na určitou oblast, která leží někde v paměti počítače. V těchto paměťových oblastech jsou data uložena. V některých počítačových jazycích musí proměnná odpovídat typu dat, na která odkazuje. Jakýkoliv pokus o přiřazení chybného typu dat do takové proměnné způsobí ohlášení chyby. Někteří programátoři dávají takovým jazykům přednost — říká se jim jazyky se *statickou typovou kontrolou*[3] —, protože umožňují předcházet určitým záludným chybám, které se obtížně hledají.

Pravidla pro tvorbu jmen proměnných jsou závislá na konkrétním programovacím jazyce. Každý jazyk předepisuje, které znaky mohou být součástí jména proměnné, a které ne. V některých jazycích — a patří k nim i Python a JavaScript — záleží na *velikosti* písmen a v angličtině jsou označovány jako *case sensitive* [keis sensitiv], čili *citlivé na velikost*. Jiné jazyky, jako například VBScript, velká a malá písmena nerozlišují. Jazyky citlivé na velikost písmen po programátorovi vyžadují trošku pečlivější přístup. Velmi nám pomůže, když budeme při volbě jmen proměnných používat určitý systém. Jeden z běžných stylů, který budeme často používat, zahajuje jméno proměnné malými písmeny a u každého dalšího slova uvádí první písmeno velké:

```
velmiDlouheJmenoPromenneSOddelovanimSlovVelkymiPismeny
```

Poznámka překladatele: Pro jména proměnných musíme v jazyce Python volit písmena bez diakritických znamének.

Nebudeme zde podrobněji probírat pravidla popisující, které znaky můžeme v našich jazycích používat. Při dodržování stylu používaného v ukázkách byste se neměli setkat s nějakými problémy.

V jazyce Python proměnná získává typ podle do ní přiřazených dat. Datový typ si tato proměnná zapamatuje a pokud se pokusíte kombinovat data různého typu nedovoleným způsobem — jako je například sčítání řetězce a čísla —, budete varováni. (Vzpomínáte si na dříve uvedený příklad s chybovým hlášením? Ten byl příkladem právě takového typu chyby.) Pokud později proměnné přiřadíte data jiného typu, pak se typ proměnné změní. (To není například u výše zmíněných jazyků se statickou typovou kontrolou možné.)

```
>>> q = 7       # v q je nyní číslo
>>> print q
7
>>> q = "Sedm"  # do q jsme přiřadili řetězec
>>> print q
Sedm
```

Povšimněte si, že na začátku byl proměnné q přiřazen odkaz na *číslo* 7. V proměnné se hodnota odkazu udržovala až do té doby, než jsme ji přinutili, aby ukazovala na řetězec "Sedm". Takže ještě jednou: proměnné jazyka Python si uchovávají typ dat, na která ukazují. Ale to, na co ukazují, můžeme změnit jednoduše dalším přiřazením do této proměnné. V tomto okamžiku jsou původní data prohlášena za 'ztracená' a Python je odstraní z paměti (pokud na ně současně neodkazuje jiná proměnná). Tento proces je znám jako *garbage collection*. (Poznámka překladatele: Čti gábidž kolekšn. Tento pojem je natolik specifický, že se obvykle nepřekládá; doslova by se dal přeložit jako *sbírání smetí*.)

Garbage collection můžeme přirovnat k činnosti poštovního úředníka, který jednou a čas odstraní všechny dopisy a balíčky ze skříněk, které nemají žádný nápis. Pokud tyto dopisy na sobě nemají cílovou adresu nebo adresu odesílatele, hodí je do smetí.

Proměnné v jazycích VBScript a JavaScript

Při používání proměnných v jazycích VBScript a JavaScript se navíc setkáváme s drobnou odlišností. Oba uvedené jazyky vyžadují, abychom proměnnou před jejím použitím *deklarovali*. Jde o obecný rys, se kterým se setkáme u kompilovaných jazyků a u jazyků se *silnou typovou kontrolou*. Jednou z velkých výhod tohoto přístupu je to, že překladač může odhalit pokus o použití neznámé proměnné. K této situaci může dojít tím, že se při zápisu použití proměnné dopustíme překlepu v jejím jméně. Nevýhodou je samozřejmě to, že programátor toho musí napsat o něco víc.

Poznámka překladatele: Uvedená nevýhoda je zanedbatelná tím více, čím větší projekt se realizuje. V takových případech programátor věnuje mnohem více úsilí ostatním věcem, než je jednoduché klapání do klávesnice. Za skutečnou nevýhodu můžeme považovat nutnost deklarace proměnných pouze u jazyků, kde musíme deklaraci uvést na výrazně jiném místě zdrojového textu, než se objevuje její první použití. (Při pohledu do zdrojového textu — typicky přes okno editoru s omezeným počtem řádků na obrazovce — může programátor ztratit přehled o deklaraci proměnné. Musí ji dohledávat a ztrácí tím část svého času a mentální kapacity.) Příkladem takového jazyka je například jazyk C. Naproti tomu například modernější jazyk C++dovoluje uvést deklaraci přímo v místě prvního použití proměnné.

Zajímavější diskusi lze vést na téma *silná* versus *slabá typová kontrola*. Donedávna se tradovalo, že silná typová kontrola je vždy a jednoznačně lepší, protože umožňuje odhalit určitou kategorii chyb již při překladu. V poslední době již nejsou názory tak vyhraněné. Silná typová může mít i nevýhody (komplikuje například *generické programování*) a navíc nezaručí odhalení jiné kategorie chyb — špatnou interpretaci hodnoty formálně správného typu. Odpovědi na tyto problémy se nyní nehledají pouze v technikách fungování překladačů, ale i v technologiích a technikách vývoje aplikací, v postupech programátorů. Jedním z přístupů, který prokázal svou užitečnost, je psaní *testů funkčnosti* (*unit testing* — jednotkové testy a *acceptance testing*— uživatelské testy). Poučení i zajímavé náměty k zamyšlení naleznete v článku Strong versus Weak Typing.

VBScript

Deklaraci proměnné zajistíme v jazyce VBScript použitím příkazu Dim, což je zkratka slova *Dimension*([dajmenžn] = rozměr). Jde o projev dávných kořenů jazyka VBScript v jazyce BASIC a přes něj ještě dále v jazycích typu assembler. V těchto jazycích jste museli uvést, jak velký paměťový prostor bude proměnná vyžadovat — tedy její rozměr, velikost. Zkratka Dim se dochovala z těchto dob.

Deklarace proměnné v jazyce VBScript vypadá takto:

```
Dim promenna
```

Jakmile je proměnná jednou deklarována, můžeme ji používat stejně, jako jsme to ukázali u Pythonu. Jedním příkazem Dim můžeme deklarovat více proměnných. Jejich seznam oddělujeme čárkami:

```
Dim promenna, druha, treti
```

Přiřazení provedeme takto:

```
promenna = 42
druha = "Toto je krátká větička."
treti = 3.14159
```

Dalším klíčovým slovem, se kterým se můžete občas setkat, je Let ([let] = nechť). Jde opět o pozůstatek z jazyka BASIC a ve skutečnosti jej nemusíte vůbec používat. Pokud přesto někdy uvidíte jeho použití, bude vypadat nějak takto:

```
Let promenna = 22
```

V této učebnici nebudeme Let používat.

JavaScript

V jazyce JavaScript je deklarace proměnných uvedena klíčovým slovem var a v jedné deklaraci můžeme uvést více proměnných, jako v případě VBScript:

```
var promenna, druha, treti;
```

Součástí příkazu var může být v jazyce JavaScript i inicializace (nebo *definice*, tj. určení počáteční hodnoty) proměnných:

```
var promenna = 42;
```

```
var druha = "Krátká věta.", treti = 3.14159;
```

Ušetříme tím trochu psaní, ale jinak se tato forma funkčně od dvoukrokové definice proměnných v jazyce VBScript nijak neliší.

Doufám, že tento stručný pohled proměnné v jazycích VBScript a JavaScript objasnil rozdíl mezi jejich *deklarací* a *definicí*. V jazyce Python dojde k *deklaraci* proměnné v okamžiku, kdy je uvedena její prvnídefinice.

Nyní se podívejme na příklady datových typů a uvidíme, jak to všechno pasuje dohromady.

Jednoduché datové typy

Jednoduché datové typy se nazývají jednoduchými proto, že patří k nejzákladnějším typům dat, se kterými se dá manipulovat. Složitější datové typy jsou ve skutečnosti jen kombinacemi jednoduchých datových typů. Jednoduché datové typy jsou stavebními kameny, ze kterých se ostatní typy budují. Jednoduché datové typy představují opravdový základ veškerých výpočtů. Patří k nim písmena, čísla a něco, čemu se říká boolovský typ (boolean).

Znakové řetězce

Už jsme se s nimi setkali. Patří k nim doslova libovolné řetězce, neboli posloupnosti znaků, které mohou být zobrazeny na vaší obrazovce. (Ve skutečnosti mohou obsahovat i netisknutelné *řídicí znaky*.)

V jazyce Python mohou být řetězce zapsány několika způsoby:

S jednoduchými apostrofy

```
'Toto je retezec'
```

S uvozovkami

```
"Toto je velmi podobny retezec"
```

S trojitými uvozovkami

```
"""Tady je velmi dlouhy retezec, ktery -- pokud
si to budeme prat -- muze byt zapsan na vice
radcich. Python zachova viceradkovy retezec
ve tvaru, v jakem jej zapiseme..."""
```

Naposledy zmíněná forma má jedno speciální použití. Používá se k dokumentování funkcí, které jsme v jazyce Python sami napsali — k tomu se ještě dostaneme.

K jednotlivým znakům řetězce můžeme přistupovat jako ke složkám znakového pole (viz *pole* dále v textu). Každý programovací jazyk obvykle poskytuje operace pro manipulaci s řetězci — nalezni podřetězec, spoj dva řetězce, okopíruj jeden řetězec jinam a podobné.

Za zmínku stojí skutečnost, že některé jazyky používají zvláštní typ pro ukládání znaků — tedy typ pro jeden znak. V takovém případě můžeme o řetězcích uvažovat jako o kolekcích hodnot typu znak. Ve srovnání s tímto přístupem Python používá pro uložení jednoho znaku jednoduše řetězec o délce jedna. To znamená, že nepotřebuje jiný syntaktický předpis pro zápis řetězce a jiný pro zápis jednoho znaku.

Řetězcové operátory

Nad řetězci lze provádět celou řadu operací. Některé z nich jsou přímo zabudovány do jazyka Pyhon, ale řada dalších je poskytována moduly, které musíme zapojit do činnosti příkazem import (tak, jak jsme to udělali s modulem sys v kapitole Jednoduché posloupnosti).

Řetězcové operátory

S1 + S2	Spojení řetězce S1 a S2
S1 * N	N-násobné opakování řetězce S1

Následující příklady demonstrují jejich funkci:

```
>>> print 'Znovu a ' + 'znovu'        # spojení řetězců
Znovu a znovu
>>> print 'Opakuj ' * 3              # opakování řetězce
Opakuj Opakuj Opakuj
>>> print 'Znova ' + ('a znova ' * 3) # kombinace '+' a '*'
Znova a znova a znova a znova
```

Řetězce znaků můžeme přiřazovat do proměnných:

```
>>> s1 = 'Znova '
>>> s2 = 'a znova '
>>> print s1 + (s2 * 3)
Znova a znova a znova a znova
```

Všimněte si, že poslední dva příklady dávají stejný výstup.

S řetězci se dá dělat spousta dalších věcí, ale k tomu se podrobněji dostaneme až v dalších tématech. Nejdříve si musíme osvojit základnější znalosti.

Řetězcové proměnné v jazyce VBScript

V jazyce VBScript jsou proměnné (v angličtině) označovány jako *variants*, protože mohou obsahovat libovolný typ dat. (Nepřekládal jsem jako *varianty*, protože by se to pletlo s *možnostmi*.) VBScript se je podle potřeby pokouší převádět na požadovaný typ. To znamená, že do proměnné můžete přiřadit číslo, ale v okamžiku, kdy proměnnou použijete v místě, kde se očekává řetězec, VBScript provede potřebný převod. Dá se říci, že se to podobá chování pythonovského příkazu print, ale podobné chování je rozšířeno na všechny příkazy jazyka VBScript. Pokud chceme překladači napovědět, že se na nějakou číselnou hodnotu má dívat jako na řetězec, uzavřeme ji do uvozovek:

```
<script type="text/vbscript">
retezec = "42"
MsgBox retezec
</script>
```

Poznámka překladatele: Zatímco print se snaží převést vše na řetězec, automatické konverze ve VBScript se principiálně snaží převádět cokoliv na cokoliv. Můžeme se na to také podívat také jinak. Snaha o převod do použitelné podoby je v Pythonu vlastností příkazu print, zatímco ve VBScript je vlastností proměnných samotných. Tuto vlastnost jazyka VBScript ocení především začátečníci a dá se dobře využít u jednoduchých skriptů. U složitějších a rozsáhlejších programů může být podobné chování zdrojem obtížně odhalitelných chyb.

Řetězce můžeme spojovat dohromady. Této operaci se říká *zřetězení* (konkatenace) a v jazyce VBScript k ní používáme operátor & takto:

```
<script type="text/vbscript">
```

```
retezec = "Ahoj, " & "vy tam!"
MsgBox retezec
</script>
```

Řetězce v JavaScript

V jazyce JavaScript mohou být řetězce uzavřeny buď v apostrofech nebo v uvozovkách. Proměnné, které použijeme pro jejich zpřístupnění, musíme před použitím *deklarovat*. Používáme k tomu klíčové slovo var. Takže deklaraci a *definici* dvou řetězcových proměnných provedeme v jazyce JavaScript takto:

```
<script type="text/javascript">
var retezec1, retezec2;
retezec1 = "Ahoj, ";
retezec2 = "vy tam!";
document.write(retezec1 + retezec2)
</script>
```

Jako poznámku na závěr uveďme, že v jazyce JavaScript můžeme vytvářet řetězce i jako *objekty* typu String. O objektech se budeme bavit o něco později. Prozatím můžeme o objektech typu String uvažovat jako o obyčejných řetězcích s určitými vlastnostmi navíc. Hlavní rozdíl spočívá v tom, že je vytváříme trochu jinak:

```
<script type="text/javascript">
var retezec1, retezec2;
retezec1 = String("Ahoj, ");
retezec2 = String("vy tam!");
document.write(retezec1 + retezec2)
</script>
```

Celá čísla

Celá čísla — v různých programovacích jazycích známá jako *integer*, nemají desetinnou část — jejich hodnoty se pohybují od velkých záporných hodnot až po velké kladné hodnoty. Tuto důležitou skutečnost bychom si měli zapamatovat. Obvykle neuvažujeme o tom, že by čísla měla být co do velikosti nějak omezena, ale u počítačů existují jejich horní a dolní hranice. Hodnota horní hranice je známa jako MAXINT a závisí na počtu bitů, které váš počítač používá pro reprezentaci čísla. Na většině současných počítačů je to 32 bitů, takže konstanta MAXINT odpovídá hodnotě kolem 2 miliard. (V jazyce VBScript je to jen kolem 32 tisíc. Pro reprezentaci čísla se používá jen 16 bitů, čili dva bajty, takže můžeme vyjádřit čísla v rozmezí přibližně +/-32 tisíc.)

Čísla, která mohou nabývat jak kladných, tak záporných hodnot jsou označována jako *celá čísla se znaménkem*(*signed integer*). Někdy se využívají i čísla, která jsou omezena pouze na kladné hodnoty a nulu. Označujeme je jako *celá čísla bez znaménka* (*unsigned integer*). (Poznámka překladatele: Pokud na chvíli zapomeneme na omezení horní hranicí, pak v matematice takovým číslům říkáme *přirozená čísla*.) V takovém případě se horní hranice zvětší na dvojnásobek hodnoty MAXINT — u 32bitových čísel na hodnotu kolem 4 miliard —, protože prostor, který byl jinak vyhrazen pro reprezentaci záporných čísel, může být využit pro reprezentaci dalších kladných čísel.

Protože jsou celá čísla (budeme říkat také *čísla typu integer*) shora omezena konstantou MAXINT, může nastat situace, kdy součet dvou čísel přesáhne konstantu MAXINT a výsledný součet bude chybný, protože správný výsledek nelze do vyhrazeného prostoru 32 bitů uložit. V některých systémech/jazycích se vrací tato špatná hodnota přímo tak jak vyšla (často je současně nastaven skrytý příznak chyby, který můžete testovat, pokud předpokládáte, že k chybě mohlo dojít). Obvykle se však v této situaci vyvolá chybový stav, který můžete vaším programem zjistit a ošetřit. Pokud tak neučiníte, program se ukončí. Jazyky VBScript a JavaScript se chovají posledně zmíněným způsobem. Poslední verze jazyka Python se v chování mírně liší. Od verze 2.3 výše Python automaticky převádí celé číslo na něco, čemu se říká *velké celé číslo* (*long integer*; nezaměňujte s typem long v jazycích C a C++, kdy se číslo ukládá na 32 bitech). Jde o specifickou vlastnost jazyka, která umožňuje pracovat s celými čísly o prakticky neomezené velikosti. Nic ovšem není zadarmo. Platíme za to cenou mnohem pomalejšího zpracování. Přinejmenším si však můžeme být jisti tím, že naše výpočty nakonec skončí korektně. Navíc rychlost je v počítačovém světě velmi relativní. Pokud takových velmi velkých celých čísel nezpracováváme mnoho, pravděpodobně si rozdílu ani nevšimnete. To, že se skutečně jedná o velké celé číslo můžeme poznat podle toho, že je Python zobrazuje s připojeným 'L' (jako long):

```
>>> 1234567 * 3456789
4267637625363L
>>> _
```

Povšimněte si, že jsme zde nepoužili příkaz print. Pokud bychom tak učinili, zmíněné 'L' by zůstalo skryto. Python umožňuje v interaktivním režimu dva způsoby zobrazování hodnot. Výsledek použití příkazu print je obvykle hezčí (ve smyslu snadnější čitelnosti), ale prosté použití hodnoty, jako ve výše uvedeném příkladu, nám občas odhalí více detailů. Zkuste si příklady z předchozích témat zapsat bez použití příkazu print a zaměřte se na pozorování drobných rozdílů v zobrazení. Příkaz print budu používat také z toho důvodu, že mnohé jazyky jeho použití vyžadují. A také chci, aby vám do krve přešly dobré obecné návyky a ne pouze podlné postupy, které vám umožňuje Python.

Aritmetické operátory

S většinou aritmetických operátorů, které budeme potřebovat, jsme se již setkali v kapitole Jednoduché posloupnosti. Zopakujme si je:

Aritmetické operátory v jazyce Python

Příklad	Popis významu
M + N	Sčítání M a N
M - N	Odčítání N od M
M * N	Násobení M a N
M / N	Dělení, jak čísel typu integer tak reálných čísel. Výsledek záleží na typu čísel M a N. Když je alespoň jedno z čísel M a N reálné, výsledek bude též reálný.
M % N	Modulo: nalezne zbytek po celočíselném dělení M : N
M**N	Umocňování: M na N-tou

O posledním z nich jsme se ještě nezmínili. Podívejme se na příklad v němž vytvoříme několik proměnných typu integer a poté použijeme operátor pro umocňování:

```
>>> i1 = 2    # vytvoř proměnnou i1 a přiřaď jí hodnotu celého čísla
>>> i2 = 4
>>> i3 = 2**4 # přiřaď výsledek dvě na čtvrtou do i3
>>> print i3
16
```

Celá čísla v jazyce VBScript

Jak již bylo uvedeno výše, hodnoty typu integer jsou v jazyce VBScript omezeny menší hodnotou MAXINT, která odpovídá uložení na 16 bitech — konkrétně zhruba +/-32 tisíc. Pokud potřebujete pracovat s větší celočíselnou hodnotou, můžete použít long. Ten co do rozsahu odpovídá typu integer, který je standardně používán v jazyce Python. Ve VBScript lze používat i typ byte, který definuje čísla uložená na 8 bitech, s maximální hodnotou 255 (interval 0 až 255). V praktických případech vám většinou bude vyhovovat použití standardního typu integer.

Podporovány jsou všechny aritmetické operátory.

Čísla v jazyce JavaScript

Jistě není žádným překvapením, že JavaScript také definuje numerický typ. Čísla mají opět podobu objektu, viz dále, a nazývají se Number (tj. číslo). Jak originální, že? :-)

V jazyce JavaScript lze použít i takzvané *Not a Number* neboli *NaN* (*ne-číslo* nebo *toto-není-číslo*). Jde o speciální verzi objektu Number a reprezentuje neplatné číslo. Většinou se používá v roli výsledku nějaké operace, která je z matematického hlediska nepřípustná. Hlavní myšlenka zavedení NaN spočívá v možnosti testovat některé typy chyb aniž by došlo k přerušení programu. JavaScript definuje i další speciální verze typu Number, které reprezentují kladné a záporné nekonečno. V programovacích jazycích se tento rys objevuje zřídka. Číselné objekty v jazyce JavaScript reprezentují buď celá čísla nebo reálná čísla — viz dále.

Reálná čísla

Jde o čísla s desetinnou částí, o zlomky[4]. Mohou reprezentovat čísla velmi velká, mnohem větší než MAXINT, ale s menší přesností. To znamená, že dvě čísla, která by měla být shodná, ve skutečnosti pro počítač stejná nejsou. Je to dáno tím, že počítač ukládá jen přibližnou hodnotu v závislosti na tom, jakou úroveň detailů čísla je schopen zachytit. Například číslo 4.0 by mohlo být uloženo jako 3.9999999… nebo 4.000000…01. Ve většině případů taková přibližnost postačuje, ale občas může mít důležité dopady. Pokud při používání reálných čísel dostanete nějaké legrační výsledky, mějte tuto skutečnost na paměti.

S reálnými čísly, známými také jako čísla *s plovoucí řádovou čárkou* (*floating-point numbers)*, můžeme provádět stejné operace jako s čísly typu integer. Navíc máme k dispozici operace pro převod na celá čísla odseknutím nebo zaokrouhlením desetinné části.

Python, VBScript i JavaScript práci s reálnými čísly podporují. V jazyce Python je vytvoříme jednoduše tím, že uvedeme zápis čísla, ve kterém se vyskytuje desetinná tečka — jak jsme si ukázali v části Jednoduché posloupnosti. V jazycích VBScript a JavaScript nejsou celá čísla a reálná čísla při používání jasně rozlišována. Jednoduše je používáme a překladač jazyka si s tím většinou dobře poradí.

Komplexní nebo imaginární čísla

Pokud máte základy matematického nebo jiného vědeckého vzdělání, pak vás možná napadlo: A co komplexní čísla? Pokud tyto základy nemáte, možná jste o komplexních číslech vůbec neslyšeli. V takovém případě můžete tuto část přeskočit, protože vám zde uvedená fakta k ničemu nebudou. Nicméně, některé programovací jazyky, včetně jazyku Python, mají podporu pro typ komplexních čísel zabudovánu přímo do jazyka, zatímco k jiným jazykům se dodávají knihovny nebo funkce, které práci s komplexními čísly umožňují. A ještě než se zeptáte — totéž platí pro matice.

V jazyce Python jsou komplexní čísla reprezentována jako:

```
(real+imaginaryj)
```

... kde real představuje reálnou složku a imaginary složku imaginární.

Takže sčítání komplexních čísel zapisujeme následovně:

```
>>> M = (2+4j)
>>> N = (7+6j)
>>> print M + N
(9+10j)
```

Všechny operace používané pro čísla typu integer je možno použít i pro komplexní čísla.

VBScript ani JavaScript práci s komplexními čísly nepodporují.

Hodnoty typu Boolean — True a False

Tento podivně vypadající název typu je pojmenován po matematikovi 19. století, George Boolovi, který se zabýval studiem logiky. Jak už nadpis napovídá, tento typ má pouze dvě hodnoty — *true* (pravda) a *false*(nepravda). Některé jazyky podporují boolovské hodnoty přímo, jiné používají konvence, kdy vybraná číselná hodnota (často nula) reprezentuje false a jiná hodnota (často 1 nebo -1) reprezentuje true. Bylo tomu tak i u jazyka Python, a to až do verze 2.2 včetně. Od verze 2.3 podporuje Python boolovské hodnoty přímo — používá hodnoty True a False.

Boolovské hodnoty jsou známy také jako *pravdivostní hodnoty*, protože vyjadřují skutečnost, zda je něco pravdivé nebo nepravdivé. Dejme tomu, že píšete program, který má zálohovat všechny soubory v adresáři. Můžeme v cyklu postupovat tak, že uložíme soubor jednoho jména a potom se operačního systému zeptáme, jak se jmenuje další soubor. Pokud už žádný další soubor neexistuje (tj. byly zpracovány všechny soubory v adresáři), vrátí se prázdný řetězec. Vrácené jméno souboru tedy můžete porovnat s hodnotou prázdného řetězce a výsledek můžete uložit jako boolovskou hodnotu (pokud je vrácený řetězec prázdný, uloží se True, pokud ne, uložíme False). Později si ukážeme, jak můžeme takto uložený výsledek použít.

Boolovské (nebo také logické) operátory

Zápis	Jméno operace	Popis významu
A and B	A současně	True, když A i B jsou True. V jiném případě je výsledkem False.
A or B	NEBO	True když oba nebo jeden z A, B jsou True. False, když oba A i B jsou False.
A == B	ROVNOST	True když A je rovno B.

A != B nebo A <> B	NEROVNOST	True když A není rovno B.
not B	NEGACE	True když B není True.

Poznámka: poslední operace se týká jedné hodnoty, zatímco ostatní porovnávají dvě hodnoty.

Jazyk VBScript, stejně jako Python, používá typ Boolean s hodnotami True a False. Jazyk JavaScript rovněž používá typ Boolean, ale jeho hodnoty jsou tentokrát pojmenovány true a false (první písmeno se píše malé).

Různé jazyky používají pro pojmenování boolovského typu mírně odlišná jména. Python mu říká *bool*, VBScript a JavaScript *Boolean*. Většinou se tím nemusíte vůbec zabývat, protože budete spíše používat boolovské výsledky v testech, než abyste je ukládali do boolovských proměnných.

Kolekce

Pro studium *kolekcí* a jejich chování byla v počítačové vědě vybudována celá disciplína. Někdy bývají kolekce označovány pojmem *kontejnery*. V této sekci se nejdříve podíváme na kolekce, které podporují jazyky Python, VBScript a JavaScript. Nakonec stručně shrneme, s jakými dalšími typy kolekcí se můžeme setkat v jiných jazycích.

Seznam

Seznamy dobře známe z každodenního života.

Seznam (anglicky *list* [list]) je jednoduše tvořen posloupností položek. Do seznamu můžeme položky přidávat nebo je můžeme naopak odstraňovat. Pokud máme seznam napsaný na papíru, pak můžeme stěží vkládat položky doprostřed. Můžeme je přidávat pouze na konec. Ale pokud seznam udržujeme v elektronické podobě, dejme tomu v textovém editoru, můžeme nové položky vkládat do libovolného místa seznamu.

V seznamu můžeme také vyhledávat — pokud chceme zjistit, zda v něm něco už je, nebo ne. V takovém případě ale musíme seznam procházet postupně, od začátku do konce, a kontrolovat, zda se jedná o položku, kterou hledáme.

Seznamy patří v mnoha moderních programovacích jazycích k základním typům s charakterem kolekce.

V jazyce Python jsou seznamy přímo jeho součástí (jsou zabudovány do jazyka). Můžeme s nimi provádět všechny základní operace, o kterých jsme se zmínili výše. Navíc můžeme položky seznamu zpřístupňovat i prostřednicvím *indexu*. To znamená, že k prvku seznamu můžem přistupovat na základě znalosti jeho pořadového čísla. (Prvnímu prvku je přiděleno pořadové číslo nula.)

V jazyce VBScript neexistují seznamy jako takové, ale jejich vlastnosti můžeme simulovat jinými typy kolekcí, o kterých se zmíníme později.

V jazyce JavaScript rovněž nemáme přímo typ seznam, ale téměř ke všemu, co potřebujete dělat se seznamem, můžete využít jeho typ *pole* (array [erey]). Jde o jiný typ kolekce, o kterém se budeme bavit o něco později.

Operace nad seznamem

Python definuje nad kolekcemi řadu operací. Téměř všechny z nich lze aplikovat na seznamy. Část operací lze aplikovat na další typy kolekcí a také na řetězce, které jsou vlastně jen speciálním případem seznamu — jde o *seznam znaků*. V jazyce Python seznam vytvoříme a zpřístupníme použitím hranatých závorek. (Tímto zápisem seznam *konstruujeme*, proto se takto použité dvojici hranatých závorek říká také *konstruktor seznamu*.)Pokud použijeme pouze hranaté závorky a nic do nich nevepíšeme, vytvoříme prázdný seznam. Seznam s hodnotami vytvoříme tak, že požadované hodnoty zapíšeme dovnitř závorek a oddělíme je čárkami:

```
>>> seznam = []
>>> jinySeznam = [1, 2, 3]
>>> print jinySeznam
[1, 2, 3]
```

K jednotlivým položkám můžeme přistupovat pomocí indexu, který uvedeme v hranatých závorkách. První položka seznamu má přidělen index 0 (nula). Pokud například chceme zpřístupnit třetí prvek, použijeme index 2:

```
>>> print jinySeznam[2]
3
```

Hodnoty položek seznamu můžeme podobným způsobem i měnit:

```
>>> jinySeznam[2] = 7
>>> print jinySeznam
[1, 2, 7]
```

Povšimněte si, že třetí prvek (index 2) změnil svou hodnotu z 3 na 7.

Záporné hodnoty indexového čísla používáme pro zpřístupnění položek indexovaných vůči konci seznamu. Nejčastěji se používá index -1 (mínus jedna), který zajistí zpřístupnění poslední položky seznamu:

```
>>> print jinySeznam[-1]
7
```

Operátorem append() můžeme přidávat nové položky na konec seznamu:

```
>>> seznam.append(42)
>>> print seznam
[42]
```

Položkou seznamu může být dokonce i jiný seznam, takže pokud připojíme na konec prvního seznamu náš druhý seznam, dopadne to takto:

```
>>> seznam.append(jinySeznam)
>>> print seznam
[42, [1, 2, 7]]
```

Všimněte si, že výsledkem je seznam složený ze dvou položek, kde druhou položku tvoří opět seznam (jak je znázorněno párem okolních hranatých závorek). V posledním případě můžeme prvek s hodnotou 7 zpřístupnit pomocí dvojitého indexu:

```
>>> print seznam[1][2]
7
```

Hodnota prvního indexu (tj. 1) zpřístupní druhou položku seznamu, která je vlastně seznamem. Hodnota druhého indexu (tj. 2) zpřístupní třetí položku zmíněného podseznamu.

Možnost vnořování seznamů jednoho do druhého je velmi užitečná. Tato vlastnost nám umožňuje budovat datové tabulky, jako je například následující:

```
>>> radek1 = [1, 2, 3]
```

```
>>> radek2 = ['a', 'b', 'c']
>>> tabulka = [radek1, radek2]
>>> print tabulka
[[1, 2, 3], ['a', 'b', 'c']]
>>> prvek2 = tabulka[0][1]
```

Tímto způsobem si můžeme vybudovat například adresář (pro ukládání adres lidí), kde každá položka představuje seznam se jménem a dalšími detaily adresy. Následující příklad uvádí adresář s dvěmi položkami:

```
>>> adresy = [
... ['Mirek', 'Kolbenova 15', 'Olomouc', '585 456 231'],
... ['Hanka', 'Ypsilantiho 42', 'Brno', '525 698 444']
... ]
```

Povšimněte si, že jsme celý zanořený seznam vytvořili zápisem na jednom řádku (je zalomen jen kvůli nedostatku prostoru — tečky znázorňují pokračování řádku). Python sleduje, zda počet uzavřených závorek odpovídá počtu otevřených. Pokud ne, pokračuje v načítání vstupu, až do doby, kdy se počty srovnají. Uvedený zápis představuje velmi efektivní způsob rychlého budování složitých datových struktur. Celková struktura — v tomto případě seznam seznamů — přitom zůstává pro čtenáře kódu přehledná.

Cvičně zkuste získat Mirkovo telefonní číslo — čtvrtý prvek z prvního řádku. Uvědomte si, že indexy začínají nulou. Zkuste přidat několik svých záznamů použitím operace append(), o které jsme se zmiňovali výše.

Jakmile Python ukončíte, vaše data budou ztracena. Až se budeme bavit o souborech, zjistíte, jak můžete data ze seznamu uložit pro další použití.

Opačnou operací k přidávání položky je, samozřejmě, rušení položky. Provedeme ji příkazem del:

```
>>> del seznam[1]
>>> print seznam
[42]
```

Pokud chceme spojit dva seznamy do jednoho, můžeme použít stejný operátor pro zřetězení '+', který jsme již použili dříve pro řetězce:

```
>>> print seznam
[42]
>>> print jinySeznam
[1, 2, 7]
>>> novySeznam = seznam + jinySeznam
>>> print novySeznam
[42, 1, 2, 7]
```

Povšimněte si, že se výsledek tentokrát liší od dříve uvedeného příkladu, kdy jsme spojovali dva seznamy operací append(). Tehdy jsme dostali seznam s dvěmi prvky, přičemž druhým prvkem byl připojovaný seznam. V tomto případě dostáváme seznam s čtyřmi prvky, protože do nového seznamu byly vloženy prvky z obou spojovaných seznamů — každý prvek samostatně. Pokud v tomto případě přistupujeme k prvku s indexem 1, nedostaneme se tentokrát k podseznamu, jak tomu bylo v předchozím případě, ale pouze k prvku s hodnotou 1:

```
>>> print novySeznam[1]
1
```

Pro naplnění seznamu více položkami se stejnou hodnotou můžeme využít operátoru pro opakování — zápis se podobá násobení:

```
>>> seznamNul = [0] * 5
>>> print seznamNul
[0, 0, 0, 0, 0]
```

Pro prvek seznamu s určitou hodnotou můžeme nalézt jeho index operací index():

```
>>> print [1,3,5,7].index(5)
2
>>> print [1,3,5,7].index(9)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
```

Povšimněte si, že pokus o nalezení něčeho, co není prvkem seznamu, má za následek chybu. V dalších částech učebnice si ukážeme způsoby, jak lze zjistit, zda se něco v seznamu nachází, či nikoliv.

Délku seznamu (počet jeho položek) můžeme zjistit voláním zabudované funkce len():

```
>>> print len(seznam)
1
>>> print len(novySeznam)
4
>>> print len(seznamNul)
5
```

Ani JavaScript ani VBScript typ seznam přímo nepodporují. Ale o kousek dál si ukážeme, že podporují typ Array (pole), se kterým lze napodobit funkčnost pythonovských seznamů.

N-tice

Typ n-tice (anglicky *tuple*) není řadou jazyků vůbec podporován. Ale v těch jazycích, kde jsou n-tice podporovány, se ukazuje, že je to velmi užitečný rys. N-tice je ve skutečnosti jen libovolná uspořádaná kolekce hodnot, se kterou můžeme zacházet jako s jedním celkem. V mnoha ohledech se n-tice podobá seznamu, ale jeden významný rozdíl spočívá v tom, že n-tice jsou *neměnné* (anglicky *immutable*). To znamená, že jakmile je jednou n-tice vytvořena, nelze ji měnit (tj. nelze měnit, přidávat nebo rušit jednotlivé položky n-tice). V jazyce Python se n-tice zapisují jako posloupnost hodnot oddělených čárkami, která je uzavřená v kulatých závorkách — takto:

```
>>> ntice = (1, 3, 5)
>>> print ntice[1] # zpřístupníme položku indexem, jako u seznamu
3
>>> ntice[2] = 7   # chyba: položku n-tice nelze měnit
Traceback (most recent call last):
File "", line 1, in ?
```

```
ntice[2] = 7
TypeError: object doesn't support item assignment
objekt nepodporuje přiřazování hodnot do položek
```

Zapamatujme si hlavně to, že kulaté závorky používáme při vytváření n-tice, že hranaté závorky se používají pro uvedení indexu při přístupu k jejím položkám a že jednou vytvořenou n-tici nelze později měnit. V ostatních případech lze na n-tice aplikovat většinu operací, které se používají pro seznamy.

Ačkoliv n-tici nemůžeme měnit, můžeme operátorem pro sčítání (plus) jakoby přidat další členy. Ve skutečnosti se tím totiž vytvoří nová n-tice:

```
>>> ntice1 = (1, 2, 3)
>>> ntice2 = ntice1 + (4,) # čárka způsobí, že se zápis chápe jako n-tice a ne jako číslo
>>> print ntice2
(1, 2, 3, 4)
```

Pokud bychom za číslem 4 neuvedli čárku, Python by to chápal jako zápis celého čísla uzavřeného v závorkách a ne jako zápis n-tice. Ale protože k n-ticím nelze přičítat čísla, vedlo by to k chybě. Přidáním čárky Pythonu říkáme, aby zápis v závorkách chápal jako zápis n-tice. Kdykoliv budete chtít Pythonu dát najevo, že jednoprvková n-tice skutečně *je* n-ticí, přidejte k zápisu čárku stejně, jako jsme to udělali zde.

VBScript ani JavaScript koncept n-tic nepodporují.

Slovník (vyhledávací tabulka)

V tištěném slovníku je k jednotlivým slovům uveden jejich význam. Podobně je tomu i u datového typu slovník (anglicky *dictionary* [dykšnri]), kde jsou k jednotlivým klíčům přidruženy hodnoty. Přitom hodnoty mohou, ale nemusí, mít podobu řetězce. Hodnotu můžeme získat tím, že klíč použijeme pro slovník jako *index*. Narozdíl od tištěného slovníku, ani klíč nemusí být znakovým řetězcem — ačkoliv řetězec se často používá. Může to být hodnota libovolného neměnného typu (immutable), včetně čísel a n-tic. Hodnota, která je s klíčem svázaná, může být libovolného datového typu jazyka Python. Slovníky jsou obvykle implementovány s využitím programovací techniky známé jako *hash table*[5] [heš tejbl]. Z tohoto důvodu se pro datový typ slovník občas používá zkrácený pojem *hash*. V české terminologii se spíše využívá druhá část úplného pojmu, tedy tabulka. Nemá to nic společného s drogami. :-)

K hodnotám ve slovníku můžeme přistupovat pouze prostřednictvím klíče, takže do slovníku můžeme vložit pouze prvky s jednoznačným klíčem (pro jeden klíč nelze současně uchovávat dvě hodnoty). Slovníky jsou velmi užitečnými strukturami. Python je poskytuje jako zabudovaný typ, ačkoliv v mnoha dalších jazycích musíte použít odpovídající modul nebo si dokonce musíte typ slovník naprogramovat sami. Slovníky můžeme používat mnoha způsoby a později si ještě ukážeme řadu příkladů. V tomto okamžiku si ukažme alespoň to, jak v jazyce Python slovník vytvoříme, jak do něj vložíme některé položky a jak je opět zpřístupníme (přečteme):

```
>>> dct = {}
>>> dct['boolean'] = "Hodnota, která je buď True nebo False"
>>> dct['integer'] = "Celé číslo"
>>> print dct['boolean']
Hodnota, která je buď True nebo False
```

Všimněte si, že počáteční hodnotu slovníku nastavíme pomocí složených závorek (zde prázdný slovník — složené závorky jsou konstruktorem slovníku tak, jako jsou hranaté závorky konstruktorem seznamu nebo kulaté závorky konstruktorem n-tice). Poté používáme hranaté závorky pro přiřazování a čtení hodnot.

Slovník můžeme naplnit počátečními hodnotami v okamžiku jeho vytvoření podobně, jako jsme si to ukázali u seznamů:

```
>>> adresy = {
... 'Mirek' : ['Mirek', 'Kolbenova 15', 'Olomouc', '585 456 231'],
... 'Hanka' : ['Hanka', 'Ypsilantiho 42', 'Brno', '525 698 444']
... }
```

Klíč a hodnota se od sebe oddělují dvojtečkou a tyto páry se od dalších oddělují čárkou. Tentokrát jsme náš adresář vytvořili jako slovník, kde jsme jako klíč použili jméno a jako slovníkovou hodnotu ukládáme původní seznamy s hodnotami. Místo zjišťování a používání číselného indexu nyní můžeme potřebné informace získat na základě jména, a to takto:

```
>>> print adresy['Hanka']
['Hanka', 'Ypsilantiho 42', 'Brno', '525 698 444']
>>> print adresy['Mirek'][3]
585 456 231
```

V druhém případě jsme vrácený seznam dále indexovali, abychom obdrželi jen telefonní číslo. Práci si můžeme dále zjednodušit zavedením pomocných proměnných, které naplníme příslušnými hodnotami indexů:

```
>>> jmeno = 0
>>> ulice = 1
>>> mesto = 2
>>> tel = 3
```

Pokud nyní chceme zjistit, ve kterém městě bydlí Hanka, můžeme napsat:

```
>>> print adresy['Hanka'][mesto]
Brno
```

Povšimněte si, že zatímco jméno 'Hanka' uvádíme v apostrofech, protože jde o klíč typu řetězec, zápis mestouvádíme bez apostrofů, protože jde o jméno proměnné, které Python převede na hodnotu indexu, kterou jsme do ní uložili (konkrétně 2). V tomto okamžiku začíná náš adresář připomínat použitelnou databázovou aplikaci. Můžeme za to poděkovat síle datového typu slovník. Moc práce nám nedá ani doplnění kódu pro ukládání a načítání dat a přidání dotazovacího řádku, přes který budeme moci určit, jaká data požadujeme. Při probírání dalších témat této učebnice si to ukážeme. Řada operací nad kolekcemi, se kterými jsme se do této chvíle seznámili, není u slovníků — vzhledem k jejich vnitřní struktuře — podporována. Nefunguje zde ani operátor zřetězení, operátor opakování, ani operace append(). Abychom si mohli zpřístupnit hodnoty všech klíčů, máme k dispozici operaci keys(). Ta vrací seznam všech klíčů, které seznam používá. Pokud například chceme získat seznam všech jmen z našeho adresáře, můžeme napsat:

```
>>> print adresy.keys()
['Hanka', 'Mirek']
```

Zde musíme upozornit na to, že klíče nejsou ve slovníku uloženy v pořadí, v jakém byly vkládány. Z tohoto důvodu se vám může zdát, že se objevují v nějakém divném pořadí, které se dokonce může během používání slovníku měnit. Nedělejte si

s tím starosti. Nic to nemění na skutečnosti, že prostřednictvím klíčů můžete přistupovat k vašim datům. Vždy korektně obdržíte tu správnou hodnotu.

Slovníky ve VBScript

V jazyce VBScript máme k dispozici objekt typu slovník, který má podobné vlastnosti, jako slovník v jazyce Python, ale používá se trochu jinak. Nejdříve musíme deklarovat proměnnou, která bude objekt zpřístupňovat. Poté vytvoříme vlastní objekt typu slovník a nakonec do něj přidáme položky:

```
Dim dict              ' Vytvoříme proměnnou.
Set dict = CreateObject("Scripting.Dictionary")
dict.Add "a", "Athens" ' Přidáme nějaké položky.
dict.Add "b", "Belgrade"
dict.Add "c", "Cairo"
```

Povšimněte si, že ve funkci CreateObject() (tj. *vytvoř objekt*) uvádíme, že chceme vytvořit objekt"Scripting.Dictionary". To znamená, že chceme vytvořit objekt Dictionary (slovník), který je pro interpret VBScript definován v modulu Scripting. Podrobnostmi se zatím nebudeme zatěžovat. Dostaneme se k nim později, až se budeme zabývat objekty. Doufám, že se vám alespoň vybavuje koncepce používání objektů definovaných v modulech — zmínili jsme se o ní v podkapitole Jednoduché posloupnosti. Povšimněte si také, že při přiřazování objektu do proměnné musíme v jazyce VBScript použít klíčové slovo Set ([set] = nastavit).

Nyní můžeme k datům přistupovat takto:

```
item = dict.Item("c")         ' Získání hodnoty položky.
dict.Item("c") = "Casablanca" ' Změna hodnoty položky.
```

Kromě toho máme k dispozici i operace k odstranění položky, k získání seznamu všech klíčů, k otestování existence klíče a podobně.

Následující příklad představuje úplnou (i když trochu zjednodušenou) podobu našeho adresáře, zapsaného v jazyce VBScript:

```
<script type="text/vbscript">
Dim adresy
Set adresy = CreateObject("Scripting.Dictionary")
adresy.Add "Mirek", "Mirek, Kolbenova 15, Olomouc, 585 456 231"
adresy.Add "Hanka", "Hanka, Ypsilantiho 42, Brno, 525 698 444"

MsgBox adresy.Item("Hanka")
</script>
```

Pro uchovávání informací tentokrát místo seznamu používáme jediný řetězec. Na ukázku zpřístupňujeme data záznamu s klíčem Hanka a zobrazujeme je v dialogovém okně.

Slovníky v JavaScript

JavaScript nemá k dispozici svůj datový typ slovník. Pokud ovšem používáte prohlížeč Internet Explorer, můžete použít objekt typu Scripting.Dictionary, který je součástí VBScript. Bavili jsme se o něm v předešlém textu. I při použití z JavaScript má naprosto stejné vlastnosti. Je to tentýž typ objektu a proto se jím zde nebudeme dále zabývat. Místo slovníků můžeme v JavaScript využít podobným způsobem typ pole — viz níže.

Pokud už toho začínáte mít plné zuby, můžete v tomto místě přejít k další kapitole. Jakmile se začnete setkávat s datovými typy o kterých jsme se ještě nezmínili, nezapomeňte se vrátit zpět k této kapitole a dočíst si ji.

Další typy kolekcí

Pole nebo vektor

Z hlediska historie počítačů patří pole k jednomu z prvních typů kolekcí. V podstatě se jedná o seznam prvků, ke kterým lze snadno a rychle přistupovat na základě indexu. Obvykle musíme předem určit, kolik prvků má pole uchovávat. A právě pevná velikost je tím rysem, který se pole liší od seznamu, o němž byla řeč výše. V Pythonu máme typ pole k dispozici prostřednictvím modulu array. Používá se však velmi zřídka, protože místo něj můžeme obvykle použít vestavěný typ seznam.

Poznámka překladatele: Modul array umožňuje v Pythonu definovat pole pouze pro prvky základních typů a to znak, celé číslo a reálné číslo. Pro jeho použití se rozhodneme pravděpodobně jen v případě, kdy nám velmi záleží na efektivnosti použití ve smyslu výkonnějšího kódu.

V jazycích VBScript i JavaScript jsou pole k dispozici v podobě datového typu. Podívejme se, jak se používají.

Pole v jazyce VBScript

V jazyce VBScript mají pole podobu kolekce dat s pevnou délkou. K prvkům se přistupuje přes číselný index. Deklarace a přístup k prvkům se zapisují takto:

```
Dim pole(42)      ' Pole s 43 prvky.
pole(0) = 27      ' Indexovat se začíná od nuly.
pole(1) = 49
promenna = pole(1) ' Čtení hodnoty.
```

Povšimněte si, že používáme klíčové slovo Dim, které vyjadřuje, že proměnné bude přidělen paměťový prostor. Tímto způsobem interpretu VBScript řekneme, že chceme pracovat s uvedenou proměnnou. Pokud na začátku skriptu uvedeme OPTION EXPLICIT, pak bude VBScript vyžadovat, abychom klíčovým slovem Dim uvedli každou proměnnou, kterou budeme chtít používat. Řada odborníků přes programování pokládá tento přístup za žádoucí, protože věří, že vede k tvorbě spolehlivějších programů. Povšimněte si také, že v deklaraci uvádíme hodnotu posledního platného indexu — v našem případě 42. Indexovat se začíná od nuly, takže to znamená, že deklarujeme pole o 43 prvcích.

Povšimněte si také, že u jazyka VBScript používáme pro deklaraci rozměru pole a pro indexování kulaté závorky, nikoliv hranaté závorky, jak je tomu u jazyka Python (viz dále), JavaScript a u řady dalších programovacích jazyků.

Deklarovat můžeme i vícerozměrná pole — modelujeme datovou tabulku. Podobný efekt jsme si ukázali v případě pythonovských seznamů. Pro náš příklad adresáře bychom mohli psát:

```
Dim MojeTabulka(2, 3)          ' 3 řádky, 4 sloupce
MojeTabulka(0,0) = "Mirek"     ' Naplníme položky pro Mirka.
MojeTabulka(0,1) = "Kolbenova 15"
MojeTabulka(0,2) = "Olomouc"
MojeTabulka(0,3) = "585 456 231"
MojeTabulka(1,0) = "Hanka"     ' Naplníme položky pro Hanku.
... a tak dále...
```

Bohužel však neexistuje způsob, jak bychom mohli předepsat naplnění tabulky daty najednou — jak jsme to ukázali u pythonovských seznamů. Položky tabulky musíme plnit jednu po druhé. Pokud zkombinujeme vlastnosti polí a slovníků, dosáhneme pro verzi ve VBScript stejných užitkových vlastností, jako tomu bylo u pythonovské verze:

```
<script type="text/vbscript">
Dim adresy
Set adresy = CreateObject("Scripting.Dictionary")
Dim Mirek(3)
Mirek(0) = "Mirek"
Mirek(1) = "Kolbenova 15"
Mirek(2) = "Olomouc"
Mirek(3) = "585 456 231"
adresy.Add "Mirek", Mirek

MsgBox adresy.Item("Mirek")(3) ' Vytiskneme telefonní číslo.
</script>
```

Poslední věc, o které bych se chtěl zmínit, je skutečnost, že pole v jazyce VBScript nemusí mít vůbec pevnou velikost. To ovšem neznamená, že si můžeme dovolit jednoduše přidávat prvky způsobem, jak jsme to dělali u seznamů. Změnu rozměrů pole musíme předepsat příkazem. Abychom to mohli udělat, musíme pole deklarovat jako *dynamické pole*. Dosáhneme toho jednoduše tím, že neuvedeme jeho rozměr:

```
<script type="text/vbscript">
Dim DynPole()
ReDim DynPole(5)  ' Počáteční velikost.
DynPole(0) = 42
DynPole(4) = 26
MsgBox "Před: " & DynPole(4) ' Dokažme, že to funguje.

' Rozměr pole změníme na 21 prvků při zachování stávajících dat.
ReDim Preserve DynPole(20)
DynPole(15) = 73
MsgBox "Po: " & DynPole(4)   ' Hodnota byla zachována.

' Opět změníme rozměr (51 prvků), ale dojde ke ztrátě dat.
ReDim DynPole(50)
MsgBox "A ještě... " & DynPole(4) & " Kam se poděla data?"
</script>
```

Z příkladu je zřejmé, že použití dynamického pole není tak příjemné, jako použití seznamu, který automaticky přizpůsobuje svou délku. Na druhou stranu má ale programátor k dispozici více prostředků pro jemné ovládání chování programu. Uvedená úroveň kontroly nad chováním programu může mimo jiné zvýšit bezpečnost, protože například některé viry mohou datové struktury s dynamicky měnitelnou velikostí zneužít.

Pole v jazyce JavaScript

Typem *pole* (Array) je v jazyce JavaScript z mnoha pohledů vyjádřeno něco jiného, než co bychom typicky čekali. To, čemu se zde říká pole, ve skutečnosti vykazuje podivnou směsici vlastností seznamů, slovníků a klasických polí. V nejjednodušším případě můžeme pole 10 prvků nějakého typu deklarovat takto:

```
var pole = new Array(10);
```

Prvky poté můžeme naplnit a zpřístupňovat takto:

```
pole[4] = 42;
pole[7] = 21;
var hodnota = pole[4];
```

Ale typ hodnoty není u prvků pole v JavaScript omezen pouze na jediný. Do každého z prvků pole můžeme přiřadit cokoliv:

```
pole[9] = "Krátký řetězec.";
var zprava = pole[9];
```

Při vytváření pole můžeme zadat dokonce seznam položek:

```
var jinePole = new Array("jedna", "dvě", "tři", 4, 5, 6);
hodnota = jinePole[3];
zprava = jinePole[0];
```

K dalším rysům polí v jazyce JavaScript patří to, že můžeme zjistit jejich délku (počet prvků) použitím skryté vlastnosti nazvané length (tj. délka):

```
var velikost = pole.length
```

Všimněte si, formát zápisu jméno.vlastnost připomíná volání funkce z pythonovského modulu, ale bez závorek. Jak je obvyklé i v jiných jazycích, pole se v JavaScript indexují od nuly. Nicméně v roli indexu nemusí u JavaScript vystupovat jen číslo. Můžeme použít i řetězec! V takovém případě se vlastnosti polí téměř shodují s vlastnostmi slovníků. Pole můžeme zvětšit jednoduše tím, že přiřadíme nějakou hodnotu prvku s indexem, který přesahuje aktuální maximální index. Následující kousky kódu zmíněné vlastnosti ilustrují:

```
pole[42] = 7;
jinePole["něco"] = 42;
zprava = jinePole["něco"];
```

A nakonec se podívejme, jak by při využití polí v JavaScript vypadal náš příklad adresáře:

```
<script type="text/javascript">
var adresy = new Array();
adresy["Mirek"] = "Mirek, Kolbenova 15, Olomouc, 585 456 231";
adresy["Hanka"] = "Hanka, Ypsilantiho 42, Brno, 525 698 444";

document.write(adresy.Hanka);
</script>
```

Povšimněte si, že v posledním příkazu se ke klíči chováme, jako kdyby to byla vlastnost objektu — podobně jako k výše zmíněné vlastnosti length.

Zásobník

O zásobníku (anglicky *stack* [stek]) můžeme uvažovat jako o na sobě naskládaných podnosech v samoobslužnérestauraci. Zaměstnanec restaurace přidává čisté podnosy na vrchol sloupce podnosů a zákazníci je jeden po druhém z vrcholu zase odebírají. Podnosy ve spodní části zásobníku se používají jako poslední (a také nejméně). Datový zásobník se chová stejně: každou položku buď do zásobníku vložíme (operace se označujepush [puš]) nebo ji ze zásobníku vybereme (pop). Vybírá se vždy ta položka, která byla do zásobníku vložena jako poslední. Tato vlastnost zásobníku je někdy označována jako *Last In First Out* (poslední dovnitř, první ven) nebo *LIFO*. Jednou z užitečných vlastností zásobníku je to, že jej můžeme využít k obrácení pořadí položek seznamu tím, že jednotlivé položky seznamu postupně vložíme do zásobníku a poté je postupně vybíráme z vrcholu zásobníku a vkládáme do seznamu. Výsledkem bude počáteční seznam s obráceným pořadím položek.

Typ zásobník není vestavěným typem jazyků Python, VBScript ani JavaScript. Jeho chování musíme vyjádřit v kódu programu. Nejvhodnější bývá obvykle vyjít z typu seznam, protože — jako v případě zásobníku — počet položek seznamu může narůstat podle potřeby.

Poznámka překladatele k zásobníku: Pokud použijeme seznam v roli zásobníku, pak jeho metoda append()realizuje stejnou funkčnost jako operace push(). Kromě toho Python pro seznam (ale i pro další struktury) definuje metodu pop() s typickým významem. Pokud nám vadí, že nemáme k dispozici přímo metodu push(), ale stačí nám zavedení *nějakého* zásobníku a vlastních funkcí push() a pop(), které pracují právě a jen s tímto zásobníkem, není to ani tak složité — viz následující ukázka:

```
>>> zasobnik = []
>>> push = zasobnik.append
>>> pop = zasobnik.pop
>>> push
<built-in method append of list object at 0x009C5B70>
>>> pop
<built-in method pop of list object at 0x009C5B70>
>>> push(1)
>>> push(2)
>>> push(3)
>>> zasobnik
[1, 2, 3]
>>> print pop()
3
>>> zasobnik
[1, 2]
>>> pop()
2
>>> pop()
1
>>> zasobnik
[]
>>> pop()

Traceback (most recent call last):
File "<pyshell#13>", line 1, in -toplevel-
pop()
IndexError: pop from empty list
>>>
```

Na prvním řádku vytvoříme prázdný seznam a na dalších dvou řádcích navážeme jména push a pop na příslušný kód objektu zasobnik (viz výpis na dalších řádcích). Pokud potom napíšeme push(1), provede se naprosto stejná činnost, jako kdybychom provedli zasobnik.append(1). Povšimněte si také, že pokus o pop() nad prázdným zásobníkem nelze tolerovat — je vyvolána výjimka.

Jakmile se naučíte pracovat s třídami a objekty, zjistíte, že není obtížné vytvořit pro zásobník vlastní třídu, která bude zveřejňovat jen požadované operace push() a pop(), případně další, dle vaší volby. Pokud se vám nastíněné, čistě objektové řešení zdá při vašich momentálních schopnostech a dovednostech nedostižné, *nepropadejte panice*. Je to naprosto normální.

Multimnožina

Multimnožina (anglicky *bag*) představuje kolekci položek, u kterých není definováno pořadí a která může obsahovat více položek se stejnou hodnotou. Tento datový typ obvykle poskytuje operace pro přidávání, vyhledávání a odstraňování položek. V našich jazycích se pro tento účel používají seznamy.

Množina

Množina (anglicky *set*) může uchovávat pouze jeden výskyt každé položky. Obvykle můžeme testovat, zda daná položka je či není prvkem množiny. Položky můžeme do množiny přidávat a odstraňovat. Dvě množiny můžeme spojovat dohromady různými způsoby, které známe z matematické teorie množin (jako je například sjednocení, průnik, atd.). Jazyky VBScript a JavaScript datový typ množiny přímo nepodporují, ale vlastnosti množin můžeme docela snadno napodobit použitím slovníků.

V jazyce Python jsou od verze 2.3 množiny dostupné v podobě modulu sets. Jeho implementace se považuje za experimentální. Od verze 2.4 se podpora množin stane součástí jádra jazyka. Poznámka překladatele: Zabudování do jádra jazyka bylo potvrzeno při vydání alfa verze Python 2.4. Množiny jsou implementovány v jazyce C, takže budou efektivnější, než v Python 2.3.

Základní použití množin (Python 2.3) vypadá nějak takto:

```
>>> import sets
>>> A = sets.Set()       # Vytvoř prázdnou množinu.
>>> B = sets.Set([1,2,3]) # Vytvož 3prvkovou množinu.
>>> C = sets.Set([3,4,5])
>>> D = sets.Set([6,7,8])
>>> # Teď si vyzkoušíme nějaké operace.
```

```
>>> B.union(C)                    # sjednocení
Set([1,2,3,4,5])
>>> B.intersection(C)             # průnik
Set([3])
>>> B.issuperset(sets.Set([2]))   # je nadmnožinou
True
>>> sets.Set([3]).issubset(C)     # je podmnožinou
True
>>> C.intersection(D) == A        # rovnost množin
True
```

Výčet množinových operací je mnohem širší, ale pro tento okamžik považuji výše uvedené za dostačující.

Fronta

Fronta (anglicky *queue* [kjú]) se podobá zásobníku ale s tím rozdílem, že první položka, která se dostane dovnitř, je zároveň první položkou, která se dostane ven. Tomuto chování se říká *First In First Out* (první dovnitř, první ven) nebo *FIFO*. K implementaci fronty se obvykle využívá pole nebo seznam.

Existuje celá řada dalších datových typů s vlastnostmi kolekce, ale ty, o kterých jsme se zmínili, patří mezi hlavní, se kterými se pravděpodobně setkáte. (V této učebnici se ve skutečnosti budeme zabývat jen některými z výše zmíněných, ale o dalších typech se můžete dozvědět v různých článcích a v diskusních skupinách věnovaných programování.)

Poznámka překladatele k frontě: Podobně, jako v případě zásobníku, můžeme i frontu v jazyce Python jednoduše implementovat s využitím operací nad seznamem. Pro operaci zařazení do fronty opět použijeme metodu append() (této operaci se někdy říká queueUp() nebo pushBack()). Pro operaci výběru ze začátku fronty můžeme použít pop(0). Parametr říká, z kterého místa prvek odstraňujeme. V případě zásobníku jsme parametr nezadávali, takže se použila jeho implicitní hodnota -1 s významem *index posledního prvku*.

Soubory

Jako uživatelům počítače by vám pojem soubor měl být dobře známý, protože soubory tvoří základ pro téměř vše, co s počítači děláme. Zjištění, že většina programovacích jazyků poskytuje speciální datový typ *file*(soubor), by vás tedy nemělo překvapit. Soubory a jejich zpracování jsou natolik důležité, že se jimi budeme zabývat až o něco později, v samostatné kapitole (Práce se soubory).

Datum a čas

Pro datum a čas bývá často vyhrazen samostatný datový typ. Někdy se pro jejich reprezentaci jednoduše používá velké číslo (typicky se jím vyjadřuje počet sekund, které uplynuly od zvoleného pevného data a času). Jindy se pro jejich uložení používá datový typ, který označujeme jako *složený* a který bude popsán v následujícím textu. Takový datový typ obvykle umožňuje snadnější zjištění měsíce, dne, hodiny atd. V dalších tématech se stručně seznámíme s používáním pythonovského modulu time. Jazyky VBScript i JavaScript používají pro práci s časem své vlastní mechanismy, ale těmi se zabývat nebudeme.

Složený, uživatelem definovaný typ

Někdy se ukáže, že výše popsané základní jednoduché typy nevyhovují a to ani po jejich uspořádání s využitím kolekcí. Někdy prostě chceme sdružit skupinu různorodých datových položek dohromady a pracovat s nimi jako s celkem. Příkladem může být položka adresy: číslo domu, ulice, město a směrovací číslo.

Většina programovacích jazyků dovoluje podobné informace sdružit to takzvaného *záznamu* (anglicky *record*[rikód]) nebo *struktury* (*structure* [strakčr]) nebo její modernější, objektově orientované podoby, *třídy* (*class*[klás]).

VBScript

V jazyce VBScript vypadá definice takového záznamu následovně:

```
Class Adresa
Public CisloDomu
Public Ulice
Public Mesto
Public PSC
End Class
```

Klíčové slovo Public zajistí přístupnost dat v celém zbytku programu. Datové položky mohou být označeny také jako Private, ale k tomu se v této učebnici dostaneme až později.

Python

Zápis stejného případu v jazyce Python se příliš neliší:

```
>>> class Adresa:
...     def __init__(self, Dum, Ul, Mesto, PSC):
...         self.CisloDomu = Dum
...         self.Ulice = Ul
...         self.Mesto = Mesto
...         self.PSC = PSC
...
```

Zápis se vám může zdát poněkud záhadný, ale nemějte obavy. V kapitole o objektově orientovaném programování si vysvětlíme, co znamená def __init__(...) a self. Povšimněme si jen toho, že identifikátor__init__ obsahuje na obou stranách *dvojici znaků podtržení*. Jde o pythonovskou konvenci, o které se zmíníme později. Když někteří lidé zkoušeli zapsat uvedený příklad na vyzývacím řádku interpretu jazyka Python, měli s tím určité poblémy. Na konci této kapitoly naleznete zvýrazněný úsek textu, který části příkladu podrobněji vysvětluje. Ale pokud vám to víc vyhovuje, můžete s jeho studiem počkat až na pozdější dobu, až se v průběhu kurzu dozvíte všechny podrobnosti. Pokud se pokoušíte o zapsání příkladu na vyzývací řádek (anglicky *prompt*)jazyka Python, ujistěte se, že jste použili stejný způsob odsazení. Jak uvidíme později, Python je v otázce odsazování úrovní zdrojového textu velmi puntičkářský.

Hlavní poznatek, který byste si měli z tohoto příkladu odnést, by měl být ten, že lze sloučit několik kousků dat do jediné struktury.

JavaScript

V jazyce JavaScript se pro definici struktury používá poněkud podivné jméno, a to function. Funkce si obvykle spojujeme s operacemi, ne s datovými kolekcemi, ale v případě jazyka JavaScript se funkce používají i pro tento účel. Vytváření objektu Adresa se v jazyce JavaScript zapisuje takto:

```
function Adresa(Dum, Ul, Mesto, PSC)
```

```
{
this.CisloDomu = Dum;
this.Ulice = Ul;
this.Mesto = Mesto;
this.PSC = PSC;
}
```

Zopakujme to ještě jednou. Výsledkem je skupina datových položek, na kterou se díváme jako na jeden celek.

Přístup ke složeným datovým typům

Hodnotu složeného datového typu můžeme také přiřadit do proměnné. Ale abychom mohli přistupovat k jednotlivým *složkám* hodnoty složeného typu, musíme použít speciální zápis, který je určen konkrétním programovacím jazykem. Obvykle se k zápisu používá tečka.

V jazyce VBScript

Pokud budeme uvažovat výše uvedenou třídu Adresa, pak bychom v jazyce VBScript mohli napsat:

```
Dim Adr
Set Adr = New Adresa

Adr.CisloDomu = 7
Adr.Ulice = "Havlíčkova"
Adr.Mesto = "Staré Město"
Adr.PSC = "790 58"

MsgBox Adr.Ulice & " " & Adr.CisloDomu & ", " & Adr.Mesto
```

Nejdříve jsme použitím klíčového slova Dim vyhradili prostor pro novou proměnnou Adr. S využitím Setvytvoříme novou *instanci* třídy Adresa. Poté do položek instance nového objektu adresy přiřadíme hodnoty a nakonec obsah zobrazíme v dialogovém okně pro zprávu (message box).

V jazyce Python

V jazyce Python — za předpokladu, že jste již napsali výše uvedenou definici třídy Adresa — můžeme psát:

```
Adr = Adresa(7, "Havlíčkova", "Staré Město", "790 58")
print Adr.Ulice, Adr.CisloDomu
print Adr.Mesto
```

Tím se vytvoří instance našeho typu Adresa a přiřadí se do proměnné Adr. V Pythonu můžeme předat hodnoty položek v okamžiku vytváření *objektu*. (Konstruktoru objektu lze předávat parametry.) Poté s využitím tečkového operátoru tiskneme složky CisloDomu, Ulice a Mesto. Můžete, samozřejmě, vytvořit více instancí třídy Adresa a do každé z nich přiřadit jiné číslo domu, ulici, a tak dále. Vyzkoušejte si to. Tušíte, jak byste mohli třídu Adresa využít pro náš příklad adresáře?

A ještě v jazyce JavaScript

Mechanismus používaný v jazyce JavaScript se velmi podobá mechanismům v ostatních jazycích. Přesto zde můžeme nalézt — jak uvidíme za chvíli — pár zvláštností. Nicméně, základní mechanismus je jednoduchý:

```
var Adr = new Adresa(7, "Havlíčkova", "Staré Město", "790 58");
document.write(Adr.Ulice + " " + Adr.CisloDomu + ", " + Adr.Mesto);
```

K zpřístupnění položek můžeme použít ještě jeden mechanismus, kdy se na objekt díváme jako na slovník a jméno pole používáme jako klíč:

```
document.write(Adr['Ulice'] + " " + Adr['CisloDomu'] + " " + Adr['Mesto']);
```

Jediný rozumný důvod k použití tohoto způsobu, který mě napadá, je ten, že jména položek získáváte za běhu programu v podobě řetězce — třeba jako výsledek čtení ze souboru nebo jako vstup zadaný uživatelem (viz dále).

Operace definované uživatelem

V některých programovacích jazycích mohou mít uživatelské datové typy uživatelem definovány i operace. Tento rys patří k základům takzvaného *objektově orientovaného programování*. Tomuto tématu bude věnována samostatná kapitola, ale v tomto okamžiku si uveďme alespoň to, že objekt se v podstatě tvořen datovými složkami a operacemi definovanými nad těmito datovými složkami. Vše je zabaleno dohromady a vystupuje to jako jediný celek. Python objekty široce využívá ve své standardní knihovně modulů a současně nám jako programátorům umožňuje vytváření svých vlastních typů objektů. Operace objektu se zpřístupňují stejným způsobem, jako datové členy uživatelsky definovaného typu — prostřednictvím tečkového operátoru —, ale jinak vypadají jako funkce. Těmto zvláštním funkcím se říká*metody*. Už jsme se s tím setkali u seznamu v podobě operace append(). Vzpomeňte si, že abychom ji mohli použít, museli jsme volanou funkci spojit se jménem proměnné:

```
>>> seznam = []       # prázdný seznam
>>> seznam.append(42) # volání metody objektu seznam
>>> print seznam
[42]
```

Pokud je typ objektu — říká se mu *třída* — definován uvnitř nějakého modulu, musíme tento modul importovat (jako jsme si již dříve ukázali v případě modulu sys). Jménu objektového typu předřadíme jméno modulu a vytvoříme instanci třídy (tj. objekt), který bude uložen v proměnné. Tu již můžeme používat aniž bychom uváděli jméno modulu. Ukážeme si to na fiktivním modulu meat, který definuje třídu Spam[6]. Importujeme uvedený modul, vytvoříme instanci třídy Spam, dáme jí jméno mySpam a poté použijeme mySpam pro přístup k jejím operacím a datovým složkám takto:

```
>>> import meat
>>> mySpam = meat.Spam()     # vytvoření instance, užití jména modulu a třídy
>>> mySpam.slice()           # užití operace objektu třídy Spam (ukrojit)
>>> print mySpam.ingredients # přístup k datům objektu
{'Pork': '40%', 'Ham': '45%', 'Fat': '15%'}
```

Na prvním řádku importujeme do našeho programu modul nazvaný meat (jde o fiktivní, neexistující modul). Na druhém řádku používáme modul meat k vytvoření instance třídy Spam tím, že identifikátor třídy použijeme, jako kdyby se jednalo o volání funkce. Na třetím řádku používáme jednu z operací třídy Spam, a sice slice(). K objektu mySpam se chováme, jako kdyby to byl modul a operace jako kdyby byla funkcí definovanou uvnitř modulu. Nakonec zpřístupňujeme některá data uchovávaná uvnitř objektu mySpam. Opět používáme zápis, který se podobá práci s modulem.

Pokud pomineme nutnost vytvoření instance objektu, pak neexistuje podstatný rozdíl mezi používáním objektů, které moduly poskytují, a funkcí, které se v modulech nacházejí. O jménu objektu můžeme uvažovat jako o visačce, která drží odpovídající funkce a proměnné seskupené dohromady.

Jiný způsob v pohledu na věc je takový, že objekty reprezentují skutečné věci v našem světě, se kterými můžeme — jako programátoři — něco dělat. Právě toto je pohled, který původně vedl ke zrození myšlenky používání objektů v programech. Týkal se zápisu počítačové simulace situací v reálném světě.

S objekty můžeme pracovat i v jazycích VBScript i JavaScript. Ve výše uvedených příkladech s typem Adresa jsme ve skutečnosti nedělali nic jiného. Definovali jsme třídu, vytvořili jsme její instanci a proměnnou, přes kterou můžeme zpřístupňovat vlastnosti instance. Znovu si projděte předchozí text a zaměřte se na to, co jsme si řekli o třídách a objektech. Zamyslete se nad tím, že třídy poskytují mechanismus pro definici nových datových typů tím, že svazují dohromady data a operace.

Specifické operátory jazyka Python

Mým prvotním cílem, kterému jsem zasvětil tuto učebnici, je naučit vás programovat. A ačkoliv zde používám jazyk Python, nevidím žádný důvod, proč byste si po přečtení tohoto textu nemohli nastudovat něco o jiném jazyce a zaměřit se na něj. Dokonce očekávám, že právě toto uděláte, protože neexistuje jediný programovací jazyk, který se hodí na všechno. Python není výjimkou. Na druhou stranu, protože jsem si vytknul takový cíl, nevěnuji se výuce všech rysů jazyka Python, ale zaměřuji se na ty, které můžete obvykle nalézt i u jiných jazyků. Výsledkem tohoto rozhodnutí je skutečnost, že některé specifické rysy jazyka Python — i když jsou poměrně mocné — nepopisuji vůbec. Patří mezi ně i speciální operátory. U většiny programovacích jazyků můžeme nalézt některé operace, které jiné jazyky nepodporují. Často jsou to právě tyto *unikátní* operátory, které dávají novým programovacím jazykům vzniknout a které jsou určitě důležitým faktorem určujícím jak populárním se jazyk stane.

Python například podporuje takové netradiční operace, jako jsou získání výřezu vnitřní části seznamu (nebo řetězce nebo n-tice — anglicky *slicing* [slajsing], zapisujeme spam[X:Y]) a operaci přiřazení n-tice (X, Y = 12, 34), které nám umožňuje zapsat přiřazení více hodnot více proměnným najednou. (Posledně uvedenému příkazu se říká také násobný nebo paralelní přiřazovací příkaz.)

Python poskytuje i prostředek k provedení požadované operace nad každým členem kolekce — slouží k tomu funkce map(). Takových věcí je mnohem více. Často se říká, že "Python dostáváte i s přiloženými bateriemi". Pokud se budete chtít dozvědět, jak tyto specifické operace jazyka Python fungují, budete muset nahlédnout do jeho dokumentace. Nakonec bych chtěl upozornit na to, že ačkoliv říkám, že tyto operace jsou specifické pro jazyk Python, neříkám, že je nemůžete nalézt v žádném jiném jazyce. Spíše chci říci, že je v každém jazyce nenalezne*tevšechny*. Operace, kterými se zabýváme v hlavním textu, jsou v nějaké podobě obecně dostupné ve všech moderních programovacích jazycích.

Tím uzavíráme náš pohled na programátorské suroviny. Nyní se posuňme k více vzrušujícímu tématu postupů (programovacích technik) a uvidíme, jak můžeme zmíněné suroviny využít.

Podrobněji vysvětlený příklad Adresa

Jak už jsem řekl dříve, detaily tohoto příkladu budou vysvětleny později. Někteří čtenáři však měli se zprovozněním pythonovského příkladu problémy. Tato poznámka vysvětluje jeho kód řádek po řádku. Úplný zápis příkladu vypadá následovně:

```
>>> class Adresa:
...     def __init__(self, Dum, Ul, Mesto, PSC):
...         self.CisloDomu = Dum
...         self.Ulice = Ul
...         self.Mesto = Mesto
...         self.PSC = PSC
...
>>> Adr = Adresa(7, "Havlickova", "Stare Mesto", "790 58")
>>> print Adr.CisloDomu, Adr.Ulice
```

Zde je vysvětlení:

```
>>> class Adresa:
```

Příkaz class (třída) říká, že hodláme definovat nový typ, který se v tomto případě nazývá Adresa. Dvojtečka vyjadřuje skutečnost, že všechny následující odsazené řádky budou součástí definice třídy. Definice končí prvním neprázdným řádkem, který není vůči prvnímu řádku definice třídy Adresa odsazen. Pokud používáte prostředí IDLE, pak si můžete všimnout, že editor po stisku klávesy Enter další řádek automaticky odsadil. Pokud jste Python spustili z příkazového řádku okna MS-DOS, pak na vyzývacím řádku překladače jazyka Python musíte provést požadované odsazení ručně, vložením mezer. Překladači jazyka Python nezáleží na tom, o kolik pozic odsadíte, pokud budete odsazovat pořád o stejnou hodnotu.

```
...     def __init__(self, Dum, Ul, Mesto, PSC):
```

První položkou uvnitř definice naší třídy je to, čemu říkáme *definice metody*. Důležitým detailem je to, že jméno konkrétně této metody začíná a končí dvojicí znaků podtržení. Jde o konvenci pro zápis jmen, kterým Python přisuzuje zvláštní význam. Tato konkrétní metoda se nazývá __init__ a jde o speciální operaci, kterou Python automaticky provede hned po vytvoření instance naší nové třídy — jak uvidíme za chvíli. Dvojtečka, tak jako v předchozím případě, jednoduše překladači jazyka Python říká, že následující skupina odsazených řádků tvoří definici této metody.

```
...         self.CisloDomu = Dum
```

Tento řádek a tři následující řádky přiřazují hodnoty vnitřním (datovým) položkám našeho objektu. Jsou odsazeny vůči řádku s příkazem def, abychom překladači jazyka Python naznačili, že představují skutečnou definici těla operace __init__. Prázdný řádek říká interpretu jazyka Python, že definice třídy byla ukončena, takže Python znovu zobrazí vyzývací řádek ve tvaru '>>> '.

```
>>> Adr = Adresa(7, "Havlickova", "Stare Mesto", "790 58")
```

Tento řádek zajistí vytvoření nové instance (tj. nového objektu) typu Adresa a Python automaticky použije výše definovanou operaci __init__ k přiřazení zadaných hodnot do vnitřních položek objektu. Vytvořená instance je přiřazena do proměnné Adr stejným způsobem, jako by byla přiřazena hodnota jiného datového typu.

```
>>> print Adr.CisloDomu, Adr.Ulice
```

Nyní tiskneme hodnoty dvou vnitřních položek objektu, které jsme zpřístupnili pomocí tečkového operátoru.

Jak jsem již řekl, dalšími detaily se budeme v této učebnici zabývat později. Klíčový poznatek, který byste si z tohoto měli odnést je, že nám Python umožňuje vytvořit náš vlastní datový typ a používat ho stejně snadno jako vestavěné typy.

Zapamatujte si

- Přes proměnné se odkazujeme na data. Někdy musí být proměnné deklarovány před tím, než je definována jejich hodnota.

- Hodnoty dat mohou být různých typů. Operace, které lze úspěšně použít, souvisí s typem dat, se kterými pracujeme.
- K jednoduchým datovým typům patří typy pro znakové řetězce[7], čísla a boolovské neboli pravdivostní hodnoty.
- Ke složeným datovým typům můžeme řadit kolekce, soubory, datum (v podobě záznamu se složkami pro den, měsíc, atd.) a uživatelem definované datové typy.
- Každý programovací jazyk definuje řadu operátorů. Součástí jeho studia je vždy seznámení se s jeho datovými typy a s operacemi, které jsou pro tyto typy k dispozici.
- Stejný operátor (například sčítání) může být užíván pro různé datové typy, ale výsledky se nemusí shodovat nebo dokonce spolu nemusí nijak souviset. (Viz příklad sčítání čísel a 'sčítání' řetězců.)

Další posloupnosti a jiné věci

O čem si budeme povídat?

- Seznámíme se s novým nástrojem pro psaní programů v jazyce Python.
- Zopakujeme si, jak využívat proměnné k uložení informací, které budeme chtít použít později.
- Řekneme si něco o komentářích a probereme si, proč je potřebujeme.
- Ukážeme si, jak kombinovat delší posloupnosti příkazů s cílem vyřešit nějaký úkol.

Takže… Nyní už víme, jak v jazyce Python zapisujeme jednotlivé příkazy. Začali jsme také brát v úvahu data a to, co s nimi můžeme dělat. Napsali jsme při tom pár delších posloupností s rozsahem, dejme tomu, 5 až 10 řádků. Přibližujeme se k tomu, abychom začali psát opravdu užitečné programy. Je tu ale jeden háček. Naše programy se ztrácejí, kdykoliv skončíme práci s Pythonem. Pokud jste zkoušeli příklady v jazycích VBScript a JavaScript, museli jste si je uložit do souboru. Proto jste je mohli spouštět opakovaně. Totéž bychom potřebovali u programů v jazyce Python. Už jsem se zmínil o tom, že k tomu můžete použít libovolný textový editor, jako je například Notepad nebo pico, a výsledek uložit do souboru s příponou .py. Poté můžete výsledek spustit z příkazového řádku — viz kapitola Začínáme. Ale existuje jednodušší způsob.

The joy of being IDLE[1]

Když jste si instalovali Python, nainstalovala se vám současně užitečná aplikace, která je sama napsána v jazyce Python. Nazývá se IDLE a je tím, čemu se říká *integrované vývojové prostředí*. To znamená, že jde o jednu aplikaci, která v sobě zahrnuje několik programátorských nástrojů. Co se týká IDLE, nemíním zde zabíhat do hloubky. Chci vyzvednout jen dvě z jeho vlastností. Najdete v něm zdokonalenou verzi toho, čemu jsme říkalipythonovský příkazový řádek (>>>), s podporou zvýrazňování syntaxe (anglicky syntax highlighting [syntax hailaiting] — různé prvky jazyka zobrazuje různými barvami) a s dalšími šikovnými vlastnostmi. A naleznete zde také dobrý textový editor (specializovaný na pythonovské zdrojové texty), který vám zprostředkuje spouštění vašich programů přímo z IDLE.

Pokud jste tak již neučinili, pak vám vřele doporučuji, abyste si IDLE vyzkoušeli. Jakmile zjistíte, jak se ve vašem operačním systému IDLE spouští, pak za nejlepší považuji shlédnout vynikající příručku Dannyho Yoo(anglicky).

Na domovských stránkách systému Python naleznete pod heslem IDLE úplnou učebnici. (Je ale v angličtině a místy může být poněkud zastaralá. Odpovídá verzi IDLE 0.5, přičemž s instalací Python 2.3.4 byla k dispozici již verze IDLE 1.0.3.)

Pokud používáte MS Windows je tu ještě další možnost v podobě aplikace Pythonwin, kterou lze stáhnout samostatně nebo jako součást balíku pywin32. Tento balík vám zpřístupní všechny nízkoúrovňové funkce architektury MFC pro MS Windows (Microsoft Foundation Classes) a — což je důležité — představuje velmi dobrou alternativu k IDLE. PythonWin lze použít pouze pod MS Windows, ale podle mého názoru je o něco lepší, než IDLE. Na druhou stranu, IDLE je standardní součástí instalace systému Python, takže jej používá více lidí. Navíc funguje na většině platforem. Ať už je to jakkoliv, vždy je lepší mít možnost volby.

Pokud dáváte přednost jednoduššímu přístupu, můžete použít některý z textových editorů, které různým způsobem programování v Pythonu podporují. Editor vim podporuje zvýrazňování syntaxe (barevné zobrazování klíčových slov, atd.), editor emacs definuje pro Python plnohodnotný editační režim, malý editor SciTE podporuje zvýrazňování syntaxe a má další pěkné rysy. Poznámka překladatele: Za vyzkoušení určitě stojí i v Javě napsaný editor jEdit.

Pokud půjdete cestou používání textového editoru, pak pravděpodobně zjistíte, že nejšikovnější bude používat tři najednou otevřená okna:

1. Editor, kde píšete zdrojový text a ukládáte jej do souboru,
2. okno s interaktivně spuštěným Pythonem, ve kterém si na příkazovém řádku (>>>) zkoušíte věci, které poté přidáváte do zdrojového textu, a
3. okno s příkazovým řádkem operačního systému, ve kterém spouštěním testujete vytvářený program.

Autor této učebnice dává přednost výše popsanému způsobu používání tří oken, ale zdá se, že většina začátečníků dává přednost stylu *vše v jednom* v podobě IDLE nebo Pythonwin. Výběr je pouze na vás.

Poznámka překladatele: Důvodem k používání přístupu *se třemi okny* bývá skutečnost, že pokročilejší uživatelé jsou již zvyklí na svůj oblíbený textový editor, ve kterém dokáží pracovat efektivněji, než v jiných editorech. Při jeho výběru se uplatňuje tolik vlivů, že prakticky o žádném editoru nelze tvrdit, že je nejlepší či horší. Je to jako s auty. Někdo má rád pohodlnější limuzíny s automatickou převodovkou a s posilovačem řízení, někomu vyhovují standardní přibližovadla s ne příliš výraznými rysy v žádném směru, někdo vyznává rychlé sportovní modely, ze kterých vymačká maximální akceleraci a výkon, i za cenu toho, že se mu od řadicí páky dělají mozoly.

Osobně jsem prošel používáním editorů vi/vim, emacs, JED (v režimu emacs) až k momentálně používanému jEdit. Kromě toho jsem používal několik editorů z integrovaných prostředí různých jazyků, pár editorů integrovaných do nástrojů typu Norton Commander a další malé editory typu Notepad a podobné. Ačkoliv bych se asi nechal přesvědčit k vyzkoušení nějakého nového editoru, nechtějte po mě, abych se vracel k vim. Nesedí mi, tak jako vi-kingům nebude sedět emacs nebo jEdit. Nepřemlouvejte mě. Já taky nebudu nikoho přemlouvat.

Pokud ve věci editoru dosud nemáte jasno, vyzkoušejte výše zmíněny SciTE. Zdá se, že má v režimu pro Python docela pěkné vlastnosti. Kromě toho se v kapitole o používání grafického uživatelského rozhranízmíníme o balíku wxPython, který mimo jiné obsahuje modul stc. Nemám s nimi velké zkušenosti, ale jak SciTe, tak modul stc využívají stejnou editační komponentu, která se vyvíjí v rámci projektu Scintilla. To jinými slovy znamená, že do svých pythonovských programů můžete zakomponovat okno fungující podobně, jako jinak samostatná aplikace SciTE. Když si spustíte demo k wxPython, pak v pravém okně pod záložkou *Demo Code* uvidíte modul stc v akci.

Za zmínku stojí další nástroj, nazývaný *PyCrust*. Svého času se jednalo o samostatný projekt, který měl být ve vztahu k wxPython tím samým, čím je IDLE ve vztahu Tkinter. Z původně samostatného projektu se stala součást wxPython. V době psaní tohoto textu existoval ve verzi 0.9.4 a na první pohled se jevil velice slibně. Konečným cílem projektu je kvalitnější náhrada IDLE: Python shell, editor (využívající modul stc) a další nástroje, které usnadňují vývoj.

Pokud zkoušíte příklady v JavaScript nebo VBScript, doporučuji využít některý z výše zmíněných editorů a nějaký vhodný prohlížeč, dejme tomu Internet Explorer, ve kterém si soubor otevřete. Když budete chtít vyzkoušet v editoru provedené

úpravy, jednoduše stiskněte tlačítko prohlížeče Aktualizovat (Reload). Poznámka překladatele: Nezapomeňte nejdříve soubor po úpravách v editoru uložit.

Krátce o komentářích

K jednomu z nejdůležitějších programátorských nástrojů patří jeden, který začátečníci často při prvním seznámení vnímají jako zbytečný — komentáře. Komentáře jsou prostě jen řádky v programu, které popisují, o co jde. Nemají žádný vliv na funkčnost programu, jsou v něm *na ozdobu*. Komentáře ovšem hrají velmi důležitou roli. Říkají programátorovi, o co v daném místě jde. A co je ještě důležitější, říkají *proč* to tak má být. Zvlášť důležité je to v případě, kdy programátor, který zdrojový text studuje, není tím, kdo jej napsal. Případně může jít o jeho dílo, ale od jeho vzniku už uplynula dlouhá doba. Význam dobrých komentářů po určitých programátorských zkušenostech jistě sami doceníte. Komentáře byly použity již u některých dříve uvedených úryvků kódu. Jsou to ony jinou barvou zvýrazněné části řádků, které začínají symbolem # pro Python nebo symbolem ' pro VBScript. Od tohoto okamžiku budu části programu komentovat pečlivěji. Množství okolního vysvětlujícího textu se bude postupně zmenšovat, protože vysvětlení budou uvedena právě v komentářích.

Každý jazyk definuje svůj způsob zápisu komentářů. V jazyce VBScript komentář začíná slovem REM (zkratka slova *remark* [rimák] = komentář) nebo, což je běžnější, apostrofem '. Ignoruje se vše, co se nachází za uvedenou značkou (do konce řádku):

```
REM print "Tento text se nikdy nevytiskne."
' Toto je také komentářový řádek.
print "Tento text se vytiskne."
```

Pokud jste někdy psali dávkové soubory v systému MS-DOS (ty s příponou .bat), může vám být slovo REM povědomé, protože v dávkových souborech se používá stejný způsob označování komentářů.

Všimněte si, že možnost používání apostrofu pro označení komentářů zabraňuje tomu, aby se apostrof mohl ve VBScript používat pro vymezování řetězců. VBScript si prostě myslí, že jde o komentář.

Python používá pro označení komenářů znak #. Následující znaky, až do konce řádku, se ignorují:

```
v = 12  # přiřaď do v hodnotu 12
x = v*v # x je v na druhou
```

Uvedený příklad shodou okolností používá velmi *špatný* styl komentování. Vaše poznámka by neměla jenom znovu popisovat význam (sémantiku) příkazu. To přece vidíme na první pohled ze zápisu samotného příkazu! Komentář by měl vysvětlovat *proč* se to dělá:

```
v = 3600   # 3600 je počet sekund za jednu hodinu
s = t*3600 # t obsahuje čas v hodinách, s v sekundách
```

Takové komentáře jsou mnohem prospěšnější.

JavaScript používá v roli komentářové značky dvojici lomítek //. A jako v předchozích případech, vše za touto značkou se ignoruje.

Některé jazyky umožnují používání víceřádkových komentářů, které jsou uzavřeny párem komentářových značek. Pokud ovšem není korektně vložena značka pro ukončení komentáře, může to vést k podivným chybám. V jazyce JavaScript víceřádkové komentáře používat lze. Zahajují se komentářovou značkou /*. Následuje text komentáře a vše je uzavřeno značkou */:

```
<script type="text/javascript">
document.write("Tohle se vytiskne.<br>");

// Toto je jednořádkový komentář

/* Toto je víceřádkový komentář. Pokračuje od tohoto řádku
dále přes tento řádek a dokonce
na tento, třetí řádek. Neobjeví se na výstupu skriptu.
Komentář je ukončen zrcadlovým obrazem otevírací značky. */

document.write("Tento text se také vytiskne.");
</script>
```

Poznámka překladatele: Pokud používáme editor se zvýrazňováním syntaxe, pak bývají komentáře zobrazovány jinou barvou nebo jiným typem písma. V takovém případě se pravděpodobnost toho, že si nevšimneme neuzavřeného víceřádkového komentáře, výrazně snižuje.

Na komentářích je důležité především to, aby vysvětlovaly význam částí programu komukoliv, kdo se program snaží číst. Měli byste na to vždy pamatovat a vysvětlovat všechny záhadné úseky, jako je například používání nějakých na první pohled podivných hodnot, složitých matematických výrazů a podobně. A myslete na to, že oním zmateným čtenářem můžete být za pár týdnů nebo měsíců právě vy.

Posloupnosti příkazů používajících proměnné

V kapitole Data, datové typy a proměnné jsme se seznámili s proměnnými. Zmiňovali jsme se o nich jako o nálepkách, kterými si označujeme naše data, abychom se na ně mohli později odkazovat. V různých příkladech se seznamy a v příkladu kódu pro ukládání adres jsme mohli pozorovat příklady použití proměnných. Nicméně, proměnné mají při programování tak podstatnou důležitost, že před probíráním dalších věcí raději používání proměnných znovu shrneme.

Nyní použijte IDLE nebo příkazový řádek Pythonu spuštěného v okně MS-DOS nebo v terminálovém okně systému Unix. Napište následující příkazy:

```
>>> v = 7
>>> w = 18
>>> x = v + w # využíváme naše proměnné při výpočtu
>>> print x
```

V této ukázce jsme vytvořili proměnné (v, w a x) a manipulovali jsme s jejich obsahem. Podobá se to použití tlačítka M na vaší kalkulačce. Výsledek uložíme do paměti s cílem použít tuto hodnotu někdy později.

Tisk výsledku můžeme vylepšit použitím formátovacího řetězce:

```
>>> print "Soucet cisel %d a %d je: %d" % (v, w, x)
```

Jednou z výhod formátovacího řetězce je to, že jeho obsah můžeme *také* uložit do proměnné:

```
>>> s = "Soucet cisel %d a %d je: %d"
>>> print s % (v, w, x) # užitečné pro stejný výstup různých hodnot
```

Příkaz print se tím velmi zkrátí, a to především v případech, kdy pracujeme s mnoha hodnotami najednou. Na druhou stranu se tím příkaz více zatemní. Rozhodnutí, zda jsou velmi dlouhé řádky lépe nebo hůře čitelné (ve srovnání s variantou uložení formátovacího řetězce do proměnné), musíte učinit sami. Pokud formátovací řetězec najdete poblíž příkazu print, jako v našem případě, pak to není tak špatné. Můžeme si pomoci ještě jedním způsobem. Proměnné pojmenováváme takovým způsobem, aby vysvětlovaly účel svého použití. Místo jména s bych například mohl přidělit naší proměnné pro uložení formátovacího řetězce jméno formatProSoucet. Potom by příklad vypadal takto:

```
>>> formatProSoucet = "Soucet cisel %d a %d je: %d"
>>> print formatProSoucet % (v, w, x) # užitečné pro stejný výstup různých hodnot
```

Pokud pak v programu používáme několik různých formátovacích řetězců, můžeme snadněji předepsat, který z formátů se má při tisku použít. Používání výstižných názvů proměnných je užitečné za všech okolností. V dalších příkladech se toho budeme držet. Prozatím naše proměnné neměly zrovna význam, který by určoval, jak by se proměnná měla jmenovat.

Na pořadí záleží

V tomto okamžiku byste si mohli myslet, že postup konstruování posloupnosti příkazů je zřejmý a že mu věnujeme zbytečně velkou pozornost. V této chvíli můžete mít pravdu, protože v našem případě je poměrně jasné, o co jde. Ale ne vždy je to tak jednoduché, jak to vypadá. Můžeme narazit na skrytou léčku. Uvažujme případ, kdy chceme posunout úroveň všech nadpisů v HTML dokumentu o jeden stupeň výše.
V HTML vytváříme nadpisy tím, že jejich text obalíme značkami:

- <h1>text</h1> pro první (nejvyšší) úroveň nadpisů,
- <h2>text</h2> pro druhou úroveň nadpisů,
- <h3>text</h3> pro třetí úroveň nadpisů, atd. (Norma HTML 4.01 definuje značky pro 6 úrovní nadpisů.)

Problém spočívá v tom, že jakmile se dostanete na pátou úroveň nadpisů, text v nadpisu je často menší, než text odstavců — což vypadá divně. Takže se můžete například rozhodnout, že posunete všechny úrovně nadpisů o jednu nahoru (tj. směrem k menšímu číslu ve značce). Můžete to poměrně snadno provést v textovém editoru při využití funkce náhrady jednoho řetězce jiným: všechny řetězce <h2 nahradíte řetězcem <h1, řetězce </h2nahradíte řetězcem </h1 a tak dále.
Ale uvažte, co se stane, když začnete nahrazovat značky s nejvyššími čísly — řekněme h4 značkou h3, potom h3značkou h2 a nakonec h2 značkou h1. Všechny nadpisy by se posunuly na první úroveň! Takže pořadí provedení posloupnosti akcí je důležité. Totéž by platilo, pokud bychom napsali program, který by zmíněné náhrady řetězců prováděl za nás (což bychom docela dobře mohli chtít udělat, protože posunování úrovní nadpisů může být úkol, který potřebujeme řešit velmi často).
V kapitole Data, datové typy a proměnné bylo uvedeno několik dalších příkladů používajících proměnné a uvádějících různé posloupnosti příkazů — a to zejména na příkladech s adresami lidí. A proč byste si vlastně nemohli vymyslet pár vlastních příkladů? Jakmile s tím budete hotovi, budeme pokračovat směrem k případové studii. Budeme ji budovat postupně a vylepšovat ji vždy, když se při probírání dalších témat naučíme nějakou novou techniku.

Tabulka násobků

Nyní uvedu příklad, který budeme rozvíjet v průběhu několika následujících kapitol. Tak jak se budeme učit novým programovacím technikám, řešení příkladu se bude postupně vylepšovat
Vzpomínáte si, že můžeme psát dlouhé řetězce uzavřené v trojitých uvozovkách? Využijeme toho k vytvoření tabulky násobků:

```
>>> s = """
1 x 12 = %d
2 x 12 = %d
3 x 12 = %d
4 x 12 = %d
"""
>>> # Pozor - komentáře nemůžeme psát dovnitř
>>> # řetězce. Staly by se jeho součástí!
>>> print s % (12, 2*12, 3*12, 4*12)

1 x 12 = 12
2 x 12 = 24
3 x 12 = 36
4 x 12 = 48

>>>
```

Poznámka překladatele: Uvedený tvar vstupu a výstupu můžete pozorovat, pokud použijete IDLE. Ve skutečnosti by byl zobrazený obsah okna o něco barevnější, protože IDLE zvýrazňuje význam různých částí zdrojového textu různými barvami. Pokud pro stejný příklad použijete interaktivní režim překladače Python (Python spuštěný v okně s příkazovým řádkem operačního systému), bude se vzhled lišit tím, že Python vypisuje vyzývací řetězec neukončeného příkazu ve tvaru '... '. Teprve za tímto řetězcem se zobrazuje text, který vkládáte z klávesnice.
Rozšířením tohoto příkladu bychom mohli dosáhnout vytištění celé tabulky násobků čísla 12 od 1 do 12. Ale neexistuje nějaký lepší způsob? Odpověď zní ano. Podívejme se, jak na to.

Zapamatujte si

- IDLE je vývojový nástroj pro psaní programů v jazyce Python. IDLE lze používat na různých platformách (jak pod operačním systémem MS-Windows, tak pod operačním systémem Unix a jeho variantami).
- Komentáře mohou zvýšit čitelnost programů, ale nemají žádný vliv na jeho funkčnost.
- Proměné mohou uchovávat mezivýsledky, které chceme použít později.

Příkazy cyklu, aneb umění opakování se

O čem si budeme povídat?

- Řekneme si, jak užívat cykly, abychom se vyhnuli opakovanému psaní příkazů.
- Ukážeme si různé typy cyklů a to, kdy je máme použít.

Na konci předchozí kapitoly jsme si vytiskli část tabulky násobků čísla 12. Ale stálo nás to mnoho psaní a pokud bychom potřebovali tabulku rozšířit, bylo by to velmi časově náročné. Naštěstí existuje lepší způsob a na něm si ukážeme opravdovou sílu, kterou nám programovací jazyky nabízejí.

Cyklus typu FOR

Nyní si ukážeme, jak v programovacím jazyku zapíšeme, že se něco má opakovat. Budeme dosazovat hodnotu proměnné a při každém opakování ji současně budeme zvyšovat. Zápis v jazyce Python vypadá takto:

```
>>> for i in range(1, 13):
...     print "%d x 12 = %d" % (i, i*12)
...
```

Poznámka 1: V příkazu range(1, 13) musíme použít 13, protože range() generuje čísla od dolní hranice včetně až po horní zadanou hranici vyjma. Na první pohled se vám to může zdát podivné, ale jsou pro to dobré důvody a časem si na to zvyknete.

Poznámka 2: Operátor for jazyka Python je ve skutečnosti operátorem, který bývá označován jako *foreach*(doslova *pro každý*). Následující posloupnost příkazů je provedena pro každý prvek kolekce. V našem případě je touto kolekcí seznam čísel generovaných funkcí range(). Můžete si to ověřit zapsáním příkazuprint range(1, 13) na příkazový řádek interpretu jazyka Python. Uvidíte, co se vytiskne.

Poznámka 3: Řádek s příkazem print je *odsazen* více, než předcházející řádek s příkazem for. To je velmi důležité, protože tím překladači jazyka Python dáváme najevo, že chceme opakovat právě příkaz print. Mohou následovat i další odsazené řádky. Python bude pro každý prvek z kolekce opakovat všechny odsazené řádky. Nezáleží na tom, jak velké odsazení použijete, ale zvolenou hodnotu odsazení musíte dodržovat.

Poznámka 4: Pokud s překladačem jazyka Python pracujete v interaktivním režimu, spustí se program až po dvojím stisku klávesy Enter. Důvod spočívá v tom, že po prvním stisku překladač jazyka Python nepozná, zda chcete k posloupnosti opakovaných příkazů přidat další řádek, či nikoliv. Pokud stisknete klávesu Enter podruhé, Python předpokládá, že jste již dokončili vkládání příkazů a program spustí.

Takže jak uvedený program pracuje? Projdeme si jej krok po kroku.

Python nejdříve použije funkci range() pro vytvoření seznamu čísel od 1 do 12. Poté přiřadí proměnné i první hodnotu seznamu. Následuje provedení odsazeného kódu při použití hodnoty i = 1:

```
print "%d x 12 = %d" % (1, 1*12)
```

Potom se Python vrátí zpět na řádek příkazem for a přiřadí i další hodnotu se seznamu, tentokrát 2. Opět se provede odsazený kus kódu, tentokrát s hodnotou i = 2:

```
print "%d x 12 = %d" % (2, 2*12)
```

Python bude odsazenou posloupnost příkazů opakovat až do doby, kdy byly proměnné i přiřazeny všechny hodnoty seznamu. V ten okamžik, po provedení těla cyklu s poslední hodnotou v proměnné i, se provádění přesune na další příkaz, který *není* odsazen. V našem příkladu žádné další příkazy nemáme, takže dojde k ukončení programu.

Stejný cyklus v jazyce VBScript

```
<script type="text/vbscript">
For I = 1 To 12
MsgBox I & " x 12 = " & I*12
Next
</script>
```

V tomto zápisu je mnohem lépe na první pohled vidět, co se děje. Hodnota I se mění od 1 až do 12 a provádí se kód uvedený před klíčovým slovem Next. V našem případě se výsledky jednoduše zobrazují v dialogovém okně. Odsazení příkazu MsgBox je nepovinné, ale díky němu se kód čte snadněji.

Poznamenejme, že ačkoliv zápis v jazyce VBScript vypadá jasněji, pythonovská verze je mnohem pružnější. Proč, to uvidíme za chvíli.

Poznámka překladatele: Zápis příkladu v jazyce Python můžeme přiblížit podobě příkladu v jazyce VBScript tím, že nevyužijeme výhod použití formátovacího řetězce:

```
>>> for i in range(1, 13):
...     print i, "x 12 =", i*12
...
```

Jednou z výhod použití formátovacího řetězce je ale například možnost předepsat, jak se má číslo naformátovat — tj. na kolik pozic se má tisknout. Vyzkoušejte:

```
>>> for i in range(1, 13):
...     print "%2d x 12 = %3d" % (i, i*12)
...
```

A totéž v JavaScript

Konstrukci for, která je běžná ve více programovacích jazycích, přebírá JavaScript z jazyka C. Vypadá takto:

```
<script type="text/javascript">
for (i = 1; i <= 12; i++) {
document.write(i + " x 12 = " + i*12 + "<br>");
}
</script>
```

Poznámka: Tato konstrukce má v kulatých závorkách uvedeny tři části:

- *Inicializační část* i = 1 se provede pouze jednou, před vším ostatním.
- *Část s testem (podmínková část)* i <= 12 se vykonává před každou iterací, tj. před každou obrátkou cyklu.
- *Přírůstková část* i++, což je zkrácený zápis příkazu pro "zvýšení proměnné i o jedničku", se vykonává po každé iteraci.

Povšimněte si, že v JavaScript uzavíráme příkazy, které se mají opakovat (*tělo cyklu*) do složených závorek {}. Ačkoliv z technického hlediska je to vše, co musíme s tělem cyklu udělat, z praktického hlediska se považuje za vhodné, aby byl kód v závorkách odsazen. Zvyšuje se tím čitelnost zdrojového textu.

Tělo cyklu se provede pouze v případě, kdy je podmínka v *testové* části splněna. Každá z uvedených částí může obsahovat libovolný kód, ale výsledkem výrazu v testové části musí být boolovská hodnota.

Poznámka překladatele: Povšimněte si použití operátoru pro zvýšení proměnné cyklu o jedničku i++. V tomto případě se jedná o takzvaný *postfixový* operátor ++. To znamená, že je zapsán až *za* identifikátor proměnné. Kromě toho existuje *prefixová* varianta téhož operátoru ++i — operátor je uveden *před* identifikátorem proměnné. Oba zvýší obsah proměnné o jedničku. V čem se tedy jejich použití liší? V tomto případě v ničem, ale...

Operátory ++ a -- pocházejí z jazyka C. V něm celá řada příkazů může vystupovat jako výraz. To znamená, že vracejí nějakou hodnotu a mohou tedy být použity například na pravé straně přiřazovacího příkazu. Mimo jiné se to týká i samotného příkazu přiřazení a zmíněných operátorů pro zvýšení a snížení o jedničku. Místo jinak běžného zápisu...

```
a = 0;
b = 0;
c = 0;
d = 0;
```

...tedy můžeme psát:

```
d = c = b = a = 0;
```

V tomto okamžiku nabývá prefixová a postfixová varianta zmíněných operátorů svůj význam. Výsledkem postfixové varianty i++ je totiž původní hodnota proměnné i (nejdříve se získá hodnota a *potom* se provádí zvýšení o jedničku), zatímco výsledkem prefixové varianty ++i je nová hodnota proměnné i (*nejdříve* se provádí zvýšení o jedničku a pak se vrací hodnota proměnné i). Pokud v předchozím případě místo nuly použijeme další proměnnou, jejíž obsah upravujeme variantami operátoru ++, dostaneme po provedení příkazů na řádku v proměnných hodnoty, které jsou uvedeny v komentářích:

```
i = 1;
a = i++;   /* a: 1, i: 2 */
b = ++i;   /* b: 3, i: 3 */
c = ++i;   /* c: 4, i: 4 */
d = i++;   /* d: 4, i: 5 */
```

Při použití postfixové verze operátoru ++ musí překladač jazyka udělat jeden krok navíc — původní hodnotu proměnné si musí někam uložit (do registru nebo do jiné, skryté, pomocné proměnné) a teprve potom může provést zvýšení původní proměnné o jedničku. Pokud překladač neprování optimalizaci, pak je v případě, kdy se můžeme rozhodnout pro prefixovou nebo postfixovou variantu příkazu ++, vždy efektivnější použít *prefixovou* variantu, tedy ++i. Výše uvedený příklad cyklu v JavaScript bychom tedy měli psát spíše takto:

```
<script type="text/javascript">
for (i = 1; i <= 12; ++i) {
document.write(i + " x 12 = " + i*12 + "<br>");
}
</script>
```

Ale v jazyce Python musíme dát pozor! Ten totiž operátory ++ a -- vůbec nezná. A zatímco pro výrazy i++nebo i-- zahlásí chybu, pro výrazy ++i nebo --i nezahlácí vůbec nic. Pokud jsme z jazyků rodiny C zvyklí používat ++i, budeme se divit, proč to nefunguje. Python jednoduše rozdělí jeden operátor na dvojici znamének. Na zápis se tedy dívá jako na výraz (+(+i)) nebo (-(-i)), což z matematického hlediska znamená prázdnou operaci.

Další informace o pythonovské konstrukci *for*

Pythonovský cyklus for prochází (říkáme také, že *iteruje*) přes všechny prvky posloupnosti. Posloupností v jazyce Python — pro případ, že byste zapoměli — je řetězec, seznam nebo n-tice. To tedy znamená, že můžeme psát cykly for, které zpracovávají libovolný ze zmíněných typů. Vytiskněme si na zkoušku jednotlivá písmena slova s využitím cyklu for aplikovaného na řetězec:

```
>>> for znak in 'slovo': print znak
...
```

Povšimněte si, že se každé písmeno vytiskne na jeden řádek. Povšimněte si také, že pokud se tělo cyklu skládá z jediného řádku, můžeme je napsat na ten samý řádek, za dvojtečku. Právě dvojtečka říká překladači jazyka Python, že bude následovat blok kódu.

Iterovat můžeme i přes n-tici:

```
>>> for slovo in ('jedno', 'slovo', 'a', 'zas', 'jine'): print slovo
...
```

Tentokrát se nám na řádcích objeví jednotlivá slova. Mohli bychom je samozřejmě při tisku spojit na jeden řádek. Využijeme triku s uvedením čárky na konci příkazu print. Pokud zde uvedeme čárku, nebude Python přecházet na další řádek a další tisk bude pokračovat tam, kde předchozí skončil. Poznámka překladatele: Každá čárka v příkazu print vygeneruje oddělovací mezeru. Platí to i pro čárku na konci příkazu print. Vypsaná slova tedy budou uvedena na jednom řádku a budou oddělena mezerou.

```
>>> for slovo in ('jedno', 'slovo', 'a', 'zas', 'jine'): print slovo,
...
```

Vidíte, jak se slova poskládala na jeden řádek?

Použití příkazu for nad seznamem jsme již viděli, protože dříve použitá funkce range() generuje právě seznam. Ale pro úplnost si uveďme příklad s přímo zapsaným seznamem:

```
>>> for prvek in ['jedna', 2, 'tri']: print prvek
...
```

S uvedeným typem cyklu, který slouží k průchodu všemi prvky, je spojen jeden zádrhel. V průběhu dostáváte *kopii* toho, co se v procházené kolekci nachází. Obsah kolekce nemůžete měnit přímo. Pokud kolekci modifikovat potřebujeme, musíme použít nevzhledný obrat, který do hry zatahuje indexy prvků v kolekci:

```
mujSeznam = [1, 2, 3, 4]
for index in range(len(mujSeznam)):
mujSeznam[index] += 1
print mujSeznam
```

Uvedený program zvětšuje každou položku uvnitř mujSeznam o jedničku. Pokud bychom nepoužili trik s indexem, pak bychom pouze zvyšovali hodnoty okopírovaných prvků, ale neměnili bychom prvky originálního seznamu.

Poznámka překladatele 1: Ono to ve skutečnosti není tak přímočaré. Problematika modifikace seznamu souvisí s tím, že u některých objektů můžeme měnit hodnotu a u některých ne. Do seznamu se vždy vkládají *odkazy* na objekty. V uvedeném příkladu jsou těmito objekty celočíselné hodnoty, které nikdy nemůžeme měnit. Můžeme se na ně dívat jako na konstanty. Přičtením jedničky k číselné konstantě dostaneme jiné číslo — jinou konstantu, odkaz na zcela jiný objekt. Tento nový odkaz však s původním odkazem v seznamu nemá nic společného.

Pokud chceme dosáhnout toho, že se v seznamu objeví jiná čísla, musíme na příslušné pozice v seznamu uložit odkazy na jiné konstantní objekty s číslem. Situace by byla jiná, pokud bychom do seznamu zařadili objekty, které mohou během své existence měnit svůj stav. O tom ale až později.

Pokud této poznámce nerozumíte nebo vás děsí, nepropadejte panice. Je to úplně normální. Časem vám to bude jasné. Chtěl jsem jen, aby nad tím nesouhlasně nekroutili hlavou ti, kteří už tomu trochu víc rozumí.

Poznámka překladatele 2: V Pythonu verze 2.3 se objevil nový rys, kterému bychom měli při řešení podobného problému dávat přednost. Místo nepěkného obratu pro získání seznamu indexů bychom měli vždy použít mnohem elegantnější a také výkonnější verzi, využívající zabudované funkce enumerate():

```
mujSeznam = [1, 2, 3, 4]
for index, hodnota in enumerate(mujSeznam):
    mujSeznam[index] = hodnota + 1
print mujSeznam
```

Použitím funkce enumerate() se zajistí, že v každé obrátce cyklu získáme sobě odpovídající dvojici (index, hodnota). V uvedeném příkladu její složky přiřazujeme do stejnojmenných proměnných a následně používáme. (... *Nepropadejte panice!*)

Další problém s cykly typu for spočívá v tom, že nemůžeme rušit prvky kolekce, přes kterou procházíme. Došlo by ke zmatku. Podobá se to trochu situaci postavy ze starých grotesek, která odřezává větev, na níž sedí. K řešení podobných situací se lépe hodí jiný typ cyklu, o kterém si něco řekneme za chvíli. K porozumění problému bezpečného odstraňování prvků kolekce však budeme potřebovat znalosti z další tématické kapitoly, která je věnovaná větvení. Vysvětlení naznačeného problému tedy uvedeme později.

Od verze Python 2.2 byly do jazyka přidány další triky, které činí cyklus for ještě mocnějším. Budeme se jimi zabývat později. Prozatím stojí za to poznamenat, že i v jazycích VBScript a JavaScript existují konstrukce cyklu pro průchod všemi prvky kolekce. Detaily se zde zabývat nebudeme. Zápis konstrukce ve VBScript vypadá symbolicky takto: for each ... in .... Zápis v jazyce JavaScript vypadá takto: for ... in .... Pokud máte zájem, můžete detailní popis nalézt na odpovídajících stránkách s nápovědou.

Cyklus typu WHILE

Cykly typu FOR nepředstavují jediný možný typ konstrukce cyklu. A to je dobře, protože u cyklu FOR musíme vědět, nebo musíme být schopni předem vypočítat, počet prováděných iterací. Takže co máme dělat v případech, kdy chceme pokračovat v provádění určitého úkolu až do doby, kdy nastane určitá situace, ale když přitom nevíme, kdy k dané situaci dojde? Můžeme například chtít načítat a zpracovávat data ze souboru, ale předem nevíme, kolik datových položek soubor obsahuje. Chtěli bychom prostě pokračovat ve zpracování dat až do dosažení konce souboru. Lze k tomu sice použít i cyklus FOR, ale je to obtížnější.

K řešení tohoto problému se hodí jiný typ cyklu — cyklus typu *WHILE*.

Zápis v jazyce Python vypadá takto:

```
>>> j = 1
>>> while j <= 12:
...     print "%d x 12 = %d" % (j, j*12)
...     j = j + 1
```

Projděme si, co jednotlivé příkazy dělají.

1. Nejdříve inicializujeme proměnnou j na 1. Nastavení počáteční hodnoty *řídicí proměnné* cyklu whilepředstavuje velmi důležitý první krok. Jeho opomenutí bývá častou příčinou chyb.
2. Poté začneme provádět samotný příkaz while. V něm se vyhodnocuje *boolovský výraz*.
3. Pokud je výsledkem výrazu hodnota True, dochází k provedení následujícího odsazeného bloku. V našem případě nabývá proměnná j hodnoty menší než 12, takže zahájíme provádění bloku kódu.
4. Provede se příkaz print, který vytiskne první řádek naší tabulky.
5. Na dalším řádku se zvyšuje (inkrementuje) hodnota řídicí proměnné j. V našem případě je to poslední stejněodsazený řádek, což znamená, že blok cyklu while končí.
6. Vracíme se opět k příkazu while a provádíme kroky 4 až 6, vždy s novými hodnotami proměnné j.
7. Uvedená posloupnost akcí se opakuje až do doby, kdy j dosáhne hodnoty 13.
8. V tom okamžiku vrátí test cyklu while hodnotu False a provádění odsazeného bloku se přeskočí. Pokračovat se bude řádkem, který má stejné odsazení, jako řádek s příkazem while.
9. V našem případě žádné další řádky nenásledují, takže program skončí.

V tomto okamžiku už by se vám to mohlo zdát docela jasné. Chtěl bych vás jen upozornit na jednu věc. Vidíte tu dvojtečku na konci řádku s příkazem while a před tím na konci řádku s for? Právě ta překladači jazyka Python říká, že bude následovat úsek kódu (*blok*). Jiné jazyky, jak uvidíme za chvíli, definují své vlastní způsoby, jak naznačit překladači skutečnost, že skupina řádků patří k sobě. Python používá kombinaci dvojtečky a odsazení.

VBScript

Podívejme se, jak vypadá zápis cyklu while v jazyce VBScript:

```
<script type="text/vbscript">
DIM J
J = 1
While J <= 12
MsgBox J & " x 12 = " & J*12
J = J + 1
Wend
</script>
```

Uvedený příklad produkuje stejné výsledky. Povšimněte si, že blok příkazů cyklu je tentokrát uzavřen klíčovým slovem Wend (což je samozřejmě zkratka pro While End). Až na tento rozdíl příklad funguje naprosto stejně, jako jeho pythonovská verze.

JavaScript

```
<script type="text/javascript">
j = 1;
while (j <= 12) {
document.write(j, " x 12 = ", j*12, "<br>");
j = j + 1;
}
</script>
```

Jak vidíte, struktura programu je velmi podobná. Jen místo Wend (VBScript) se objevily složené závorky. Ani VBScript, ani JavaScript (na rozdíl od Pythonu) nevyžadují jakékoliv odsazování. Kód se odsazuje jen proto, aby byl čitelnější.

V JavaScript ještě stojí za to, abychom porovnali cykly for a while. Připomeňme, že cyklus for vypadal nějak takto:

```
for (j = 1; j <= 12; j++) {....}
```

Má tedy naprosto stejnou strukturu, jako cyklus while, jen s tím rozdílem, že je vše stlačeno do jednoho řádku. Jasně zde vidíme inicializační část, testovanou podmínku a část úprav pro další obrátku cyklu. Takže v jazyce JavaScript představuje cyklus for pouze kompaktnější formu cyklu while. Bez cyklu for bychom se mohli zcela obejít. Stačí nám pouze cyklus while. Některé jazyky volí právě takový přístup.

Poznámka překladatele: Podoba cyklu for je do JavaScript převzata z jazyka C. To, že kopíruje činnost cyklu while je známkou jeho nižší úrovně abstrakce. (Jazyk C je někdy nazýván *vysokoúrovňovým assemblerem*.) Porovnejte si stejný příklad opět s jazykem Python, kdy naopak musíme převést abstrakci jednoduchého čísla na *posloupnost* hodnot, abychom vůbec mohli konstrukci for použít. Pythonovský cyklus for je z jazykového hlediska modernější. Až poznáte všechny jeho možnosti, určitě nebudete ve většině případů dávat přednost cyklu while. Užitečnost cyklu pracujícího s vyššími abstrakcemi je také důvodem, proč VBScript a JavaScript definují i dříve zmíněné formy cyklu pro iteraci přes všechny prvky kolekce.

Pružnější zápis cyklů

Vraťme se zpět k naší tabulce násobení číslem 12 ze začátku této kapitoly. Cyklus, který jsme vytvořili, se pro tisk takové tabulky velmi dobře hodí. Ale jak by to bylo s jinými hodnotami? Mohli bychom cyklus upravit tak, aby produkoval tabulku násobků třeba číslem 7? Mělo by to vypadat nějak takto:

```
>>> for j in range(1, 13):
...     print "%d x 7 = %d" % (j, j*7)
```

Při úpravě jsme museli hodnotu 12 změnit na hodnotu 7 a to na dvou místech. A pokud bychom chtěli použít jinou hodnotu, museli bychom ji, opět na dvou místech, změnit znovu. Nebylo by lepší, kdybychom mohli nějakým obecnějším způsobem zadat požadovaného násobitele?

Můžeme toho dosáhnout tím, že místo konkrétní hodnoty použijeme další proměnnou. Hodnotu této proměnné nastavíme před zahájením cyklu:

```
>>> nasobitel = 12
>>> for j in range(1,13):
...     print "%d x %d = %d" % (j, nasobitel, j*nasobitel)
```

Takto získáme naši starou známou tabulku násobení číslem 12. Ale pokud nyní budeme chtít násobit sedmi, stačí, když změníme pouze hodnotu proměnné nasobitel.

Povšimněte si, že zde kombinujeme zápis posloupnosti příkazů a příkaz cyklu. Nejdříve jsme použili jednoduchý příkaz nasobitel = 12, za kterým následuje *v pořadí další* příkaz cyklu for.

Vnořené cykly

Použijme nyní předchozí příklad k dalšímu kroku. Dejme tomu, že chceme vytisknout všechny tabulky násobků čísel od 2 do 12 (násobení číslem 1 je příliš jednoduché než abychom se jím zabývali). Jediné, co musíme učinit, je použít proměnnou nasobitel jako součást dalšího cyklu:

```
>>> for nasobitel in range(2, 13):
...     for j in range(1, 13):
...         print "%d x %d = %d" % (j, nasobitel, j*nasobitel)
```

Všimněte si, že odsazená část uvnitř prvního cyklu for je zápisem přesně téhož cyklu, s kterým jsme začínali. Funguje to následovně:

1. Nastavíme nasobitel na první hodnotu (2) a provedeme druhý cyklus.
2. Poté hodnotu proměnné nasobitel změníme na následující hodnotu (3) a znovu provedeme vnitřní cyklus,
3. a tak dále.

Tato technika je známa jako *vnořování* cyklů.

Drobnou nepříjemností je to, že se nám všechny tabulky spojí dohromady. Můžeme ji odstranit tím, že na konci prvního cyklu vytiskneme oddělovací čáru:

```
>>> for nasobitel in range(2, 13):
...     for j in range(1, 13):
...         print "%d x %d = %d" % (j, nasobitel, j*nasobitel)
...     print "--------------------"
```

Všimněte si, že druhý příkaz print je odsazený o stejnou hodnotu, jako řádek s druhým cyklem for — jde tedy o druhý příkaz v posloupnosti příkazů cyklu. (Prvním příkazem je zde vnořený cyklus.) Zapamatujte si, že úroveň odsazení je v jazyce Python velmi důležitá.

Jen pro porovnání uveďme, jak by to vypadalo v jazyce JavaScript:

```
<script type="text/javascript">
for (nasobitel = 2; nasobitel < 13; nasobitel++) {
    for (j = 1; j <= 12 ; j++) {
document.write(j, " x ", nasobitel, " = ", j*nasobitel, "<br>");
    }
document.write("----------------<br>");
}
</script>
```

Pokuste se vytvořit oddělovač tabulek, který by říkal, jaká tabulka mu předchází — popis pod tabulkou. Nápověda: Pravděpodobně budete muset použít proměnnou nasobitel a pythonovský formátovací řetězec.

Ostatní typy cyklů

Některé jazyky umožňují více typů konstrukcí cyklu, ale obvykle podporují něco jako for a while. (Modula 2 a Oberon poskytují pouze cykly typu while, protože cykly for jimi můžeme nasimulovat — jak jsme viděli výše.) Jiné typy cyklů, se kterými se můžete setkat jsou:

do-while

Tento typ cyklu je stejný jako while, ale test se provádí až na konci, za tělem cyklu. To znamená, že se cyklus provede vždy alespoň jednou.

repeat-until

Podobá se předchozímu typu s tím, že logika testu je opačná.

GOTO, JUMP, LOOP, atd.

Lze se s nimi setkat hlavně ve starších jazycích. V kódu se obvykle definuje značka a skáče se přímo na takto označené místo.

Zapamatujte si

- Cyklus FOR opakuje sadu příkazů při pevně daném počtu iterací (obrátek cyklu).

- Cyklus WHILE opakuje sadu příkazů dokud je splněna podmínka. Může se stát, že se *tělo* cyklu neprovede nikdy. Stane se tak v případě, kdy se pokračovací podmínka hned na začátku vyhodnotí jako *false*(nepravda).
- Existují i jiné typy cyklů, ale FOR a WHILE jsou k dispozici téměř ve všech jazycích.
- Cyklus for v jazyce Python je skutečnosti cyklem typu foreach — pracuje nad položkami seznamu.
- Cykly lze do sebe vnořovat.

Styl zápisu kódu

O čem si budeme povídat?

- Další možnosti využití komentářů.
- Používání odsazování pro zvýšení čitelnosti programu.
- Úvod k používání modulů pro ukládání našich programů.

Komentáře

V kapitole Další posloupnosti jsme se již o komentářích bavili. Nicméně, s komentáři lze dělat i další věci. O některých z nich se zde zmíním.

Informace o historii verzí

Bývá dobrým zvykem vytvořit na začátku každého souboru *hlavičku*. Měly by v ní být uvedeny takové detaily, jako je datum vytvoření, jméno autora, datum poslední změny, verze a obecný popis, týkající se obsahu. Často se uvádí seznam záznamů o změnách. Takový blok textu se uvádí formou komentáře:

```
# -*- coding: cp1250 -*-
################################
# Modul: Spam.py
# Autor: A.J.Gauld
# Datum: 1999/09/03
# Verze: Draft 0.4
u'''
Tento modul definuje třídu Spam, kterou můžeme kombinovat
s kteroukoliv jinou variantou třídy Jidlo, abychom vytvořili
zajímavé  pokrmy.
'''
################################
# Log:
# 1999/09/01 AJG - Vytvoření souboru.
# 1999/09/02 AJG - Opravena chyba v cenové strategii.
# 1999/09/02 AJG - Tentokrát už jsem to udělal správně!
# 1999/09/03 AJG - Přidána grilovací metoda (viz požadavek #1234)
################################
import sys, string, food
...
```

Takže když soubor poprvé otevřete, měli byste na začátku uvidět souhrnnou informaci o tom, k čem soubor slouží, co se během času změnilo, kdo a kdy změnu provedl. Důležité je to zejména tehdy, když je soubor součástí týmového projektu, a vy potřebujete zjistit, koho se máte zeptat na věci kolem návrhu a změn. Existují nástroje pro správu verzí, které vám mohou pomoci vytváření takové dokumentace zautomatizovat, ale jejich vysvětlování přesahuje rámec této učebnice[1].

Povšimněte si, že jsem popis modulu vložil mezi dvě posloupnosti tvořené trojicí apostrofů. Jde o pythonovský trik známý jako *dokumentační řetězec*. Za chvíli si ukážeme, jak můžeme jeho obsah zpřístupnit prostřednictvím zabudované funkce help()

Poznámka překladatele: Proti anglickému originálu se tato ukázka liší ve dvou detailech. Na prvním řádku je uveden komentář ve speciálním tvaru:

```
# -*- coding: cp1250 -*-
```

Tento řádek pythonovskému překladači říká, že zdrojový text byl zapsán s využitím znakové sady cp1250 (známé také pod normalizovanou zkratkou windows-1250). Pokud tedy překladač narazí na znak s kódem, který leží mimo definici kódu podle normy ASCII, může ověřit, zda jde o znak z uvedené sady (zda je v ní definován) a podle potřeby jej může převést do jiného kódování.

Druhá odlišnost spočívá v tom, že úvodní trojici apostrofů předchází písmeno u. To znamená, že se dokumentační řetězec v době překladu uloží v takzvaném *unicode* kódování. Zjednodušeně řečeno jde o to, že se pro jeden znak vyhradí větší prostor, než pouhý jeden bajt. To umožní jedním kódem rozlišit mnohem více různých znaků. Typicky se pro uložení používají dva bajty, které umožňují rozlišit přibližně 65 tisíc možností (různých znaků). Při 8bitovém kódování znaků se dá rozlišit pouze 256 možností, přičemž mnohé mají již historickými okolnostmi pevně přidělený význam.

Praktický dopad z obecného pohledu je ten, že při zápisu řetězců v kódování unicode vystačíme pro všechny nám blízké (i méně blízké) jazyky s jediným systémem kódování znaků. Pro budoucí verze Pythonu se počítá s tím, že se unicode stane jediným používaným způsobem kódování řetězců uvnitř interpretu.

Praktický dopad z pohledu běhu našeho programu je ten, že v případě důsledného používání kódování unicode budou zobrazované texty vypadat stejně při běhu v anglicky mluvících zemích, v Německu, v Řecku, v Rusku... Případná databáze aplikace bude moci dohromady bez omezení kombinovat texty v libovolném jazyce. V jednom řetězci můžeme uvést anglický text, jeho český ekvivalent, ruskou podobu zapsanou azbukou, atd.

Uvědomte si, že při zápisu zdrojového textu v 8bitovém kódování musí dojít k převodu kódování na unicode. To je možné pouze tehdy, když se ví, jaké 8bitové kódování je při zápisu zdrojového textu použito. Právě proto je první řádek velmi důležitý. Pokud není uveden, pak si novější verze Pythonu budou stěžovat v případě, že naleznou znak s kódem větším, než 127.

Zakomentování nadbytečného kódu

Tato technika se často používá k izolaci chybného úseku kódu. Předpokládejme například, že program čte nějaká data, zpracovává je, tiskne výstup a potom ukládá výsledek zpět do datového souboru. Pokud výsledky neodpovídají našemu očekávání, pak bychom rádi zabránili zpětnému ukládání (chybných) dat, abychom tento soubor neporušili. Odpovídající kód bychom mohli jednoduše odstranit. Méně radikální přístup spočívá v tom, že dané řádky jednoduše změníme na komentář, například takto:

```
data = ctiData(soubor)
```

```
for polozka in data:
    vysledky.pridej(vypoctiVysledek(polozka))
tiskniVysledky(vysledky)
######################
# Následující kód zakomentujeme až do doby,
# kdy bude opravena chyba ve funkci vypoctiVysledek,
# která vyhodnocuje výslednou položku.
#
# for polozka in vysledky:
#     soubor.ulozit(polozka)
######################
print 'Konec programu'
```

Jakmile je chyba odstraněna, můžeme jednoduše smazat komentářové značky a kód se tím stane znovu aktivním. Některé nástroje pro editaci zdrojových textů, včetně IDLE, definují v menu akce pro zakomentování vybraného úseku kódu, případně k jeho pozdějšímu odkomentování.

Poznámka překladatele: U některých editorů lze k této akci využít vlastností editace sloupcových bloků. Například volně dostupný editor jEdit vám umožní definovat sloupcový blok o nulové šířce. Jakmile začnete něco psát, začne se chovat jako sada pod sebou umístěných kurzorů pro vepisování textu — vše se opisuje na všech řádcích v místě, kde je sloupcový blok označen. Vepsáním jediného znaku '#' tedy zakomentujete celý úsek, před který jste takový sloupcový blok umístili.

Dokumentační řetězce

Komentáře, které dokumentují, co daná funkce nebo modul dělají, můžeme použít ve všech jazycích. Ale jen pár jazyků — jako je Python a Smalltalk — jdou o krok dále a umožňují nám dokumentovat funkci takovým způsobem, že mohou být použity v integrovaném prostředí pro realizaci interaktivní nápovědy během programování. V jazyce Python toho dosáhneme použitím """dokumentačních řetězců""":

```
class Spam:
    """Maso určené ke kombinování s ostatními potravinami.

    Ve spojení s jinými potravinami se dají vytvořit zajímavá jídla.
    Obsahuje mnoho živin a může být připraveno mnoha různými způsoby."""

    def __init__(self):
        ...

print Spam.__doc__
```

Poznámka překladatele k českým znakům v ukázce: Pokud si bude Python stěžovat, že používáte nepovolené znaky, zkuste je nahradit *česká* písmenka *ceskymi* (bez diakritiky). K tomuto problému se ještě vrátíme na jiných místech.

Poznámka: *Dokumentační řetězec* můžeme vytisknout jako obsah speciální proměnné __doc__. Dokumentační řetězce mohou být definovány pro moduly, funkce, třídy a metody tříd. Vyzkoušejte například:

```
import sys
print sys.__doc__
```

Od verze 2.2 definuje Python i zabudovanou funkci help(), která vyhledá a zobrazí veškerou dostupnou dokumentaci k objektu, který je zadán parametrem. Pokud například chceme něco zjistit o sys.exit, můžeme na pythonovském vyzývacím řádku napsat:

```
>>> import sys
>>> help(sys.exit)
Help on built-in function exit:

    exit(...)
    exit([status])

    Exit the interpreter by raising SystemExit(status).
    If the status is omitted or None, it defaults to zero (i.e., success).
    If the status is numeric, it will be used as the system exit status.
    If it is another kind of object, it will be printed and the system
    exit status will be one (i.e., failure).

>>>
```

Pokud tento příkaz napíšete v interaktivním režimu Pythonu v DOSovém okně a výpis zabírá více, než jednu obrazovku, zobrazuje se další text až po stisku mezerníku. Pokud chcete režim výpisu nápovědy ukončit, stiskněte 'q' (jako quit). Pokud příkaz help napíšete v interaktivním režimu v IDLE nebo v jiném integrovaném prostředí, pak se výpis po stránce nezastavuje. Vypíše se jednoduše celý text, protože se ke všem řádkům výpisu můžete dostat pomocí posouvací lišty na pravé straně okna.

Poznámka překladatele: Výše zmíněné chování při výpisu nápovědy příkazem help() si můžete vyzkoušet například na dlouhém výpisu k modulu array:

```
>>> import array
>>> help(array)
```

Příkaz help() není v Pythonu nijak utajen. Spíše naopak. Po interaktivním spuštění Pythonu verze 2.3.4 můžete číst výzvu k tomu, že máte napsat "help":

```
Python 2.3.4 (#53, May 25 2004, 21:17:02) [MSC v.1200 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Když napíšeme help bez závorek, Python nám napoví:

```
>>> help
Type help() for interactive help, or help(object) for help about object.
>>>
```

Odtud vidíme, že zabudovanou funkci help() (tj. se závorkami) můžeme volat i bez parametrů. V takovém případě vstoupíme do interaktivního režimu nápovědy. Povšimněte si, že výpis končí vyzývací posloupností 'help>':

```
>>> help()

Welcome to Python 2.3!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://www.python.org/doc/tut/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

Jak nám Python napovídá, interaktivní režim můžeme ukončit napsáním slova quit (stačí i jediné písmeno q):

```
help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

Jak vidíte, Python nás informuje o tom, že se dostáváme opět do interaktivního *příkazového* režimu.

Odsazování bloku

Jde o jedno z nejžhavějších témat, o kterém se mezi programátory debatuje. Skoro se zdá, že každý programátor má svou vlastní představu o nejlepším způsobu odsazování kódu. Byly vypracovány některé studie, které ukazují, že přinejmenším některé z faktorů mají význam větší, než kosmetický — usnadňují nám pochopení kódu.

Důvody pro debatu jsou jednoduché. Ve většině programovacích jazyků má odsazování čistě kosmetický význam; jde o pomůcku pro čtenáře. (Pokud se týká jazyka Python, zde je odsazování nutné a má zásadní vliv na správnou funkčnost programu!) Takže například:

```
<script type="text/vbscript">
For I = 1 To 10
MsgBox I
Next
</script>
```

Tento zápis je interpretem jazyka VBScript chápán jako naprosto shodný se zápisem:

```
<script type="text/vbscript">
For I = 1 To 10
  MsgBox I
Next
</script>
```

Zápis s odsazením se nám prostě lépe čte.

Podstatné je, že odsazování by mělo odrážet logickou strukturu kódu. Mělo by vizuálně odrážet tok řízení v programu. Tady nám pomáhá, když bloky kódu vypadají jako bloky (v geometrickém smyslu)…

```
XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
```

… což se čte lépe než následující zápis…

```
XXXXXXXXXXXXXXXXXXXXX
XXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXX
```

protože je lépe vidět, co k bloku patří. Studie prokázaly výrazné zlepšení srozumitelnosti v případech, kdy odsazování odráží strukturu logických bloků. V malých příkladech, se kterými jsme se dosud setkali, se to nemusí zdát důležité. Ale jakmile začnete psát programy se stovkami a tisíci řádků, důležitost vhodného odsazování výrazně vzroste.

Jména proměných

Jména proměných, která jsme dosud používali, nevyjadřovala žádný význam. Důvodem bylo především to, že jsme je použili jen pro ilustraci použitých technik. Obecně je ale mnohem lepší, kdy jména vašich proměnných vyjadřují to, co jimi chcete reprezentovat. Například v naší ukázce programu pro generování tabulek násobků jsme použili proměnnou nasobitel, která určovala, která z tabulek se tiskne. Je to určitě smysluplnější, než kdybychom použili jenom n — fungovalo by to stejně a ušetřili bychom si psaní.

Jde o kompromis mezi srozumitelností a úsilím. Obecně se dá říct, že nejlepší je volit krátká, ale výstižná jména. Příliš dlouhá jména by nás mátla a opakované zápisy jejich správné podoby by byly obtížné. Místo proměnné nasobitel jsme mohli použít například proměnnou tabulka_kterou_tiskneme, ale takové jméno je mnohem delší a přitom není o nic jasnější.

Ukládání vašich programů

Použití příkazového řádku interpretu jazyka Python v interaktivním režimu (>>>)je velmi vhodné pro rychlé vyzkoušení nápadů. Ale jakmile činnost interpretu ukončíme, vše je ztraceno. Z dlouhodobého hlediska chceme umět napsat

programy, které budeme opakovaně spouštět. Dosáhneme toho vytvořením textového souboru s připonou .py. (Jde pouze o konvenci — můžete použít jakoukoliv jinou příponu. Ale podle mého názoru je dobré takové konvence dodržovat.) Poté můžeme program spustit tím že na příkazový řádek operačního systému napíšeme:

```
$ python spam.py
```

kde spam.py je jméno našeho souboru s pythonovským programem a '$' je vyzývacím znakem příkazového řádku operačního systému. (Znak '$' se používá v unixových systémech. V systémech MS Windows se jako vyzývací znak používá '>'.)

Další výhoda ukládání programů do souborů spočívá v tom, že můžete opravovat chyby, aniž byste museli znovu napsat celý fragment nebo, to se týká IDLE, aniž byste museli kurzor přesunovat nahoru, nad chybová hlášení za účelem opakovaného výběru kódu. IDLE umožňuje otevření souboru pro editaci a jeho současné spuštění prostřednictvím menu Edit|Run module.

Od této chvíle v příkladech obvykle nebudu ukazovat vyzývací posloupnost >>>. Budu předpokládat, že program zapisujete do souboru a soubor spouštíte buď z IDLE nebo z příkazového řádku shellu nebo DOSového okna, což je můj oblíbený způsob.

Poznámka pro uživatele systému Windows

Pod Windows lze v aplikaci Průzkumník nastavit vazbu pro soubory s příponou .py tak, abyste mohli pythonovské programy spouštět jednoduše poklepáním na ikonu souboru. Instalátor systému Python by to měl udělat již dříve za vás. Můžete si to ověřit tím, že naleznete nějaký soubor s příponou .py a zkusíte ho takto spustit. Pokud k jeho spuštění dojde (a to i v případě, že se objeví pythonovské chybové hlášení), je vazba pro příponu nastavena. Problém, se kterým se v takovém případě pravděpodobně setkáte, spočívá v tom, že se program spustí v DOSovém okně, které se po dokončení programu hned zavře. Někdy to proběhne tak rychle, že si vytvoření DOSového okna stěží všimnete. Problém lze vyřešit takto:

- První, nejjednodušší způsob spočívá ve vložení následujícího řádku na konec každého programu:

```
raw_input(u'Pro ukončení programu stiskněte Enter')
```

Příkaz prostě zobrazí uvedenou zprávu a čeká, až uživatel stiskne klávesu Enter. O příkazu raw_input() se budeme bavit v jednom z následujících témat.

- Druhý způsob využívá úpravy nastavení Průzkumníka Windows. Nejedná se o nic nestandardního, ale postup, jakým výsledku dosáhneme, se může mírně lišit podle toho, jakou verzi Windows používáte. Níže uvedený postup odpovídá systému Windows XP Home. V menu Průzkumníka vyberte položkuNástroje — Možnosti složky.... V zobrazeném dialogovém okně vyberte záložku Typy souborů. Posunujte se v seznamu dolů, až uvidíte typ souboru PY. (Hledání si můžete usnadnit tím, že nejdříve stiknete klávesu P. Seznam se posune tak, abyste viděli první položku, která začíná na P.) Položku označte tím, že na ni kliknete. Poté klikněte na spodní tlačítko Upřesnit. Objeví se nové dialogové okno. Vyberte v něm akci open ([oupn] = otevřít; touto položkou je definována akce, která se má provést pro otevření souboru) a klikněte na tlačítko Upravit.... V dalším zobrazeném dialogu uvidíte popis Aplikace použitá k provedení akce a pod ním řádek vypadající přibližně takto:

```
C:\Python23\python.exe "%1" %*
```

Upravte jej tak, že za python.exe připíšete -i, takže bude vypadat následovně:

```
C:\Python23\python.exe -i "%1" %*
```

Nyní zavřete všechny dialogy. Uvedená úprava zajistí, že Python po dokončení vašeho programu neskončí, ale přejde do interaktivního režimu a zobrazí vyzývací posloupnost >>>. Poté můžete například zjišťovat obsah proměnných nebo prostě Python ručně ukončit.

Jiný možný trik spočívá v tom, že k zmíněné položce open přidáme další, kterou nazveme například Test. Poté budeme moci klinout na ikonu pythonovského souboru pravým tlačítkem myši a vybrat si buď položku open, která program spustí a poté okno automaticky uzavře, nebo si vybereme položku menu Test, která program spustí a po ukončení zůstane v interaktivním režimu Pythonu. Získáme tedy možnost volby.

Poznámka překladatele: Položku Test vytváříme prostřednictvím aplikace Průzkumník podobným způsobem, jako jsme upravovali parametry položky open. Zopakujme raději celý postup: V menu vybereme Nástroje — Možnosti složky.... V zobrazeném dialogovém okně vybereme záložku Typy souborů. Označíme položku odpovídající typu souboru PY. Klikneme na spodní tlačítko Upřesnit. Objeví se nové dialogové okno. Klikneme do seznamu s akcemi (například na položku open) a poté klikneme na tlačítko Nová.... V dalším dialogovém okně do horní části vepíšeme Test a do spodní části:

```
C:\Python23\python.exe -i "%1" %*
```

Jinými slovy to znamená, že výše popisovanou úpravu nebudeme provádět v rámci akce open, ale že si můžeme pro tento účel vytvořit svoji akci, kterou vyvoláváme pravým tlačítkem myši. Nemusíme ji dokonce pojmenovat Test. Můžeme jí dát popisnější jméno, jako například Test pythonovského programu s ukončením v interaktivním režimu.

Poznámka pro uživatele systému Unix

Systém Unix se k pythonovským souborům chová jako k ostatním skriptům, proto by první řádek pythonovského *skriptového souboru* měl obsahovat posloupnost #! následovanou plnou cestou k aplikaci python — do místa, kde ji máte nainstalovanou ve vašem systému. Plnou cestu můžete nalézt tak, že na příkazovém řádku shell napíšete:

```
$ which python
```

Na mém systému to pak vypadá takto:

```
#! /usr/local/bin/python
```

To vám umožní soubor spouštět aniž byste museli přímo volat Python. Souboru ještě musíte příkazem chmodpřidělit příznak spustitelnosti, ale jsem si jistý, že tohle jste již věděli:

```
$ spam.py
```

Na moderních unixových systémech (včetně všech distribucí systému Linux) můžete dokonce použít ještě šikovnější trik, kdy zápis cesty k aplikaci můžeme nahradit zápisem /usr/bin/env/python takto:

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

Tím se zajistí automatické vyhledání Pythonu mezi cestami v path. Jediná nepříjemnost, se kterou byste se pak mohli potkat, by nastala v případě, kdy máte v systému nainstalovaných více verzí Pythonu a skript funguje pouze při použití jedné z nich. (Může například využívat zcela nový jazykový rys, který je podporován jen v poslední verzi Pythonu.) V takovém případě je výhodnější zůstat u použití plné cesty k aplikaci.

Řádek #! nezpůsobí žádný problém ani při spouštění pod Windows nebo Mac. Vypadá prostě jako komentář. To znamená, že tento řádek mohou do svých skriptů uvádět i uživatelé systémů Windows nebo Mac pokud předpokládají, že by jejich skript mohl být užitečný a použitelný i v unixových systémech.

VBScript a JavaScript

Pokud chcete používat pouze VBScript a JavaScript, pak výše uvedený text můžete ignorovat. Vaše programy jste museli ukládat do souborů již dříve, protože to byla jediná možnost, jak je spustit.

Zapamatujte si

- Komentáře můžeme použít k dočasnému zablokování provádění kódu, což je užitečné při testování nebo při ladění kódu.
- Komentáře můžeme použít k vytvoření vysvětlující hlavičky souboru a seznamu historie změn verzí.
- Dokumentační řetězce mohou být použity k poskytnutí informace o modulu a objektech uvnitř modulu *za běhu*.
- Odsazování bloků kódů čtenáři pomáhá rozpoznat logickou strukturu kódu.
- Když program v jazyce Python místo za příkazový řádek interpretu uložíme do souboru, můžeme jej podle požadavků spouštět tím, že na příkazový řádek napíšeme $ python jmeno_programu.py nebo ve Windows poklepeme na jméno souboru.

Konverzace s uživatelem

O čem si budeme povídat?

- Řekneme si, jak máme dát uživateli najevo, že má vložit data, a jak je můžeme přečíst poté, když je vložil.
- Ukážeme si načítání jak numerických dat, tak řetězců.
- Zmíníme se o koncepci *stdin* a *stdout* (zařízení pro standardní vstup a standardní výstup).
- Podíváme se na rozhraní s charakterem příkazového řádku a na to, jak můžeme získat údaje, zapsané v podobě parametrů na příkazovém řádku při spuštění programu.

Naše programy zatím používaly pouze statická data. Taková data jsme mohli, pokud jsme to potřebovali, prozkoumat ještě před spuštěním programu, takže jsme program mohli napsat tak, aby jim vyhovoval. Ale většina programů taková není. Většina programů očekává, že budou řízeny uživatelem. Přinejmenším očekávají, že jim uživatel sdělí jaký soubor mají otevřít, upravit jeho obsah a podobně. Jiné programy si v kritických místech od uživatele vyžádají vstup dat. A právě to se v programování nazývá *uživatelským rozhraním*. Návrhem a budováním uživatelského rozhraní se v komerčních programech zabývají specialisté, kteří byli školeni v oblastech styku člověka se strojem (human-machine interaction) a ergonomie. Běžný programátor je takového luxusu ušetřen, takže se musí řídit citem a musí pečlivě zvažovat, jakým způsobem budou uživatelé program používat.

Nejzákladnějším rysem uživatelského rozhraní je zobrazování výstupu. S jeho nejprimitivnější formou jsme se již seznámili v podobě pythonovského příkazu print (a také v podobě funkce write() jazyka JavaScript a v podobě dialogu MsgBox z jazyka VBScript). Další krok v návrhu uživatelského rozhraní spočívá v přímém získávání *vstupu* od uživatele.

Nejjednodušší způsob, jak to zařídit, spočívá v tom, že se program za běhu na vstupní údaj zeptá. Druhý nejjednodušší způsob vyžaduje od uživatele, aby potřebné informace zadal při spuštění programu (jako parametry programu). A konečně se dostáváme i ke *grafickému uživatelskému rozhraní*(GUI = Graphical User Interface [grafikl júzr interfejs]) s různými vstupními poli a dalšími prvky. V této části se podíváme na první dvě metody. S programováním grafického uživatelského rozhraní se seznámíme v této učebnici až mnohem později, protože jde o problematiku výrazně složitější.

Při využití interaktivního režimu systému Python v okně IDLE nebo v terminálovém okně operačního systému si nyní ukážeme, jak lze od uživatele získat data. Poté si ukážeme, jak lze totéž provést přímo z programu.

```
>>> print raw_input(u"Něco napište: ")[1]
```

Poznámka překladatele: Při pokusech s českými texty *v interaktivním režimu* můžete narazit na problémy. Při pokusech proto doporučuji nepoužívat řetězce v kódování Unicode (viz předchozí odkaz na podrobnější poznámku). V interaktivním režimu se raději úplně vyhněte používání českých znaků. Pokud ovšem kód uložíte do souboru a dodržíte pravidla pro práci s řetězci v kódování Unicode, měly by programy korektně fungovat.

Jak můžete pozorovat, raw_input() jednoduše zobrazí zadanou *výzvu* — v našem případě "Něco napište: " — a zachytí vše, co uživatel napíše jako odpověď. Příkaz print pak tuto odpověď zobrazí. Odpověď bychom ale mohli místo zobrazení přiřadit do proměnné:

```
>>> odpoved = raw_input(u"Jak se jmenuješ? ")
>>> print u"Tak ty se jmenuješ %s! Jsem rád, že jsem tě poznal." % odpoved
```

Příkaz raw_input() má bratrance jménem input(). Rozdíl spočívá v tom, že raw_input() sesbírá uživatelem napsané znaky a chápe je jako textový řetězec, zatímco input() se z nich bude snažit vytvořit číslo. Pokud například uživatel napíše znaky '1', '2' a '3', pak input tyto tři znaky přečte a převede je na číslo *123*.

Použijme nyní příkaz input() k tomu, aby uživatel rozhodl, která tabulka násobků se má tisknout:

```
nasobitel = input(u"Vyberte hodnotu násobitele: ")
for j in range(1, 13):
    print "%d x %d = %d" % (j, nasobitel, j * nasobitel)
```

Použití příkazu input je naneštěstí velmi záludné. Je to dáno tím, že input() se nesnaží vyhodnocovat jen čísla, ale snaží se na libovolný vstup pohlížet jako na kód v jazyce Python a snaží se jej provést. To znamená, že by znalý uživatel se zákeřnými úmysly mohl napsat příkaz jazyka Python, který by například vymazal nějaký soubor na vašem PC! Z tohoto důvodu bude lepší, když se budete držet příkazu raw_input() a výsledný řetězec si budete převádět na potřebný datový typ použitím zabudovaných konverzních funkcí jazyka Python. Je to v podstatě velmi snadné:

```
>>> nasobitel = int(raw_input(u"Vyberte hodnotu násobitele: "))
>>> for j in range(1, 13):
...     print "%d x %d = %d" % (j, nasobitel, j * nasobitel)
```

Vidíte? Volání raw_input() jsme jednoduše obalili voláním int(). Efekt je stejný, jako kdybychom použiliinput(), ale je to mnohem bezpečnější. Existují i jiné konverzní funkce, takže uživatelský vstup můžete stejně dobře převádět na reálná čísla a na další typy.

Takže co kdybychom si to vyzkoušeli v opravdovém programu? Vzpomínáte si na příklady realizující adresář lidí, které využívaly strukturu typu slovník? Zabývali jsme se jimi v rámci tématu Data, datové typy a proměnné. Podívejme se na tento příklad znovu, v situaci, kdy již známe cykly a umíme číst vstup od uživatele.

```
# Vytvoříme prázdný slovník používaný jako adresář osob.
adresy = {}

# Načítáme jeho položky, dokud není zadán prázdný řetězec.
```

```
print
jmeno = raw_input(u'Zadejte jméno (nic => konec): ')
while jmeno != '':
polozka = raw_input(u'Zadejte ulici, město, telefon (nic => konec): ')
adresy[jmeno] = polozka
jmeno = raw_input(u'Zadejte jméno (nic => konec): ')

# Teď se budeme ptát, co se má zobrazit.
jmeno = raw_input(u'Které jméno se má zobrazit? (nic = konec): ')
while jmeno != '':
print jmeno + ':', adresy[jmeno]
jmeno = raw_input(u'Které jméno se má zobrazit? (nic = konec): ')
```

Prozatím je to náš nejrozsáhlejší program. Uživatelské rozhraní je sice trochu neučesané, ale funguje. V dalších tématech uvidíme, jak je můžeme vylepšit. U tohoto programu se zmiňme o jedné věci, a sice o boolovském testu v příkazu cyklu while. Podle jeho výsledku se určuje, zda chce uživatel ukončit činnost (zda se má ukončit provádění cyklu). Povšimněte si také, že datovou část uchováváme v podobě jednoho řetězce, zatímco v příkladu v části Data, datové typy a proměnné jsme informaci uchovávali v samostatných buňkách. Zatím jsme se totiž nezabývali tím, jak bychom mohli řetězec rozdělit na několik částí. Dostaneme se k tomu v pozdějších tématech. S programem implementujícím adresář osob se dále v učebnici ještě setkáme. Postupně jej budeme upravovat do užitečnější podoby.

Vstup v jazyce VBScript

V jazyce VBScript se vstup zadávaný uživatelem čte příkazem InputBox. Můžeme psát:

```
<script type="text/vbscript">
Dim vstup
vstup = InputBox("Zadejte své jméno")
MsgBox ("Zadali jste: " & vstup)
</script>
```

Funkce InputBox() zobrazí dialogové okno s textem výzvy a s polem pro zadávání vstupu. Zadaný obsah tohoto pole získáme jako návratovou hodnotu funkce. Při volání můžeme funkci navíc předat další parametry, jako je například řetězec zobrazovaný v titulku okna. Pokud uživatel stiskne tlačítko Storno (Cancel) vrací funkce prázdný řetězec nezávisle na tom, co bylo zadáno do vstupního pole.

Následující příklad ukazuje implementaci adresáře osob v jazyce VBScript.

```
<script type="text/vbscript">
Dim adresy, jmeno, polozka ' Vytvoříme proměnné.
Set adresy = CreateObject("Scripting.Dictionary")
jmeno = InputBox("Zadejte jméno", "Položka adresáře osob")
While jmeno <> ""
polozka = InputBox("Zadejte detaily - ulice město, telefon", "Položka adresáře osob")
adresy.Add jmeno, polozka   ' Přidej klíč a detaily.
jmeno = InputBox("Zadejte jméno", "Položka adresáře osob")
Wend

' Teď se budeme ptát, co se má zobrazit.
jmeno = InputBox("Zadejte jméno", "Zobrazení údaje z adresáře osob")
While jmeno <> ""
MsgBox(jmeno & " - " & adresy.Item(jmeno))
jmeno = InputBox("Zadejte jméno", "Zobrazení údaje z adresáře osob")
Wend
</script>
```

Základní struktura progamu se zcela shoduje s Pythonovskou verzí, až na pár řádků navíc. VBScript vyžaduje, aby byly proměnné předem deklarovány příkazem Dim. Každý příkaz cyklu musí být navíc ukončen příkazem Wend.

Čtení vstupu v jazyce JavaScript

JavaScript pro nás předtavuje určitou výzvu, protože jde o jazyk, který se používá především pro webovské prohlížeče. Jako takový nemá žádný speciální příkaz pro vstup. Místo toho můžeme číst z HTML elementuform. V prohlížeči Internet Explorer můžeme případně použít technologii Active Scripting firmy Microsoft a nechat si zobrazit dialog InputBox stejně, jako jsme to udělali v jazyce VBScript. Abychom se seznámili s různými možnostmi, ukáži zde techniku využívající HTML formuláře. Pokud vám pojem HTML formulář nic neříká, můžete nahlédnout do referenční příručky HTML (viz norma HTML 4.01) nebo do nějaké učebnice, která jej vysvětluje. Můžete také jednoduše okopírovat to, co uvádím dále. Doufám, že vysvětlování významu ani není třeba. Slibuji, že se to budu snažit načrtnout co nejjednodušším způsobem.

Základem struktury našeho příkladu v HTML bude javascriptový kód vložený do funkce. Zatím jsme se s tím ještě nesetkali. Prozatím můžete detaily kolem definice funkce ignorovat.

```
<script type="text/javascript">
function mujProgram(){
alert("Získali jsme hodnotu " + document.formular.pole.value);
}
</script>

<form name='formular'>
<p>Zadej hodnotu a potom klikni myší mimo vstupní pole</p>
<input type='text' name='pole' onChange='mujProgram()'>
</form>
```

Program se skládá z jediného řádku, který zobrazí dialog alert (velmi se podobá MsgBox() v jazyce VBScript). V něm se zobrazuje hodnota získaná ze vstupního pole. Ve formuláři je zobrazen vyzývací text (uzavřený do párových značek a) a vstupní pole. V kontextu dokumentu document má formulář jméno formular. Vstupní pole jsme pojmenovali jednoduše pole. Na jeho hodnotu, která byla vložena uživatelem, se tedy můžeme v javascriptovém programu odkázat takto:

```
document.formular.pole.value
```

Příklad implementace našeho adresáře osob v JavaScript zde ukazovat nebudu. Vzhledem k použití v HTML bude vše o něco složitější a zvýší se i četnost používání funkcí. S příkladem tedy chci počkat až do doby, kdy potřebnou problematiku probereme v samostatných tématech.

Pár slov o stdin *a* stdout

Poznámka: Slovem *stdin* se v počítačovém žargonu označuje standardní vstupní zařízení (obvykle klávesnice). Slovo *stdout* se vztahuje ke standardnímu výstupnímu zařízení (obvykle k obrazovce). V diskusích o programování se s termíny stdin a stdout setkáváme poměrně často. Abychom mohli využít kód pro práci se soubory, je to zařízeno tak, že se stdin a stdout chovají jako soubory.

V systému Python jsou *stdin* a *stdout* součástí modulu sys a jmenují se sys.stdin a sys.stdout. Funkce raw_input() používá automaticky *stdin*, příkaz print používá zase *stdout*. Ze standardního vstupu můžeme číst a na standardní výstup můžeme zapisovat také přímo. Má to určité výhody ve smyslu jemnějšího řízení vstupu a výstupu. Uveďme si příklad čtení jednoho řádku ze standardního vstupu:

```
import sys
print "Zadej hodnotu: ",       # čárka zabrání přechodu na další řádek
hodnota = sys.stdin.readline() # explicitní použití stdin
print hodnota
```

Funkčnost se téměř shoduje s chováním příkazu:

```
print raw_input("Zadej hodnotu: ")
```

Výhodou explicitního použití *stdin* je to, že standardní vstup můžeme svázat se skutečným souborem, takže program svůj vstup nebude číst z klávesnice, ale z daného souboru. Tento obrat je užitečný při realizaci dlouhých testů programu, kdy místo ručního vkládání vstupních údajů v okamžiku, kdy je program požaduje, necháme vstup přečíst z předem připraveného souboru. (Další výhodou je v takovém případě skutečnost, že test můžeme spouštět opakovaně, přičemž jsme si jisti, že vstup bude pokaždé stejný, takže by tomu měl odpovídat stejný výstup. Tuto techniku opakovaného spouštění dřívějších testů, které mají ověřit, že se nic nepokazilo, programátoři nazývají *regresní testování*.)

Na závěr si ukažme příklad přímého výstupu na standardní výstupní zařízení sys.stdout. Tento výstup může být rovněž přesměrován do fyzického souboru. Funkčnost příkazu print se přibližně shoduje s funkčností následujícího zápisu:

```
sys.stdout.write("Ahoj, vy tam!\n") # \n = nový řádek
```

V praxi tento obrat použijeme hlavně v případech, kdy chceme obejít vlastnost příkazu print, který vždy vkládá mezi výstupní hodnoty alespoň mezeru. Při použití stdout se tomu můžeme vyhnout. Srovnejte obsah dvou řádků na výstupu následujícího příkladu:

```
import sys
for polozka in ['jedna', 'je', 1]:
    print polozka,                  # čárka potlačí přechod na nový řádek
print
for polozka in ['jedna', 'je', str(1)]: # musíme explicitně převést na řetězec
    sys.stdout.write(polozka)           # zcela bez mezer!
```

Poznámka překladatele — výstup by měl vypadat takto:

```
jedna je 1
jednaje1
```

Pokud předem víme, jak budou data vypadat, pak stejného výsledku můžeme samozřejmě dosáhnout i použitím formátovacího řetězce. Ale pokud to nevíme, pak je jednodušší jednoduše posílat vše na stdout, než abychom se snažili o generování složitého formátovacího řetězce za běhu programu.

Přesměrování stdin a stdout

Takže jak vlastně můžeme přesměrovat stdin a stdout z a do souborů? Můžeme toho dosáhnout přímo z našeho programu, když použijeme běžné pythonovské techniky pro práci se soubory (budeme se tím zabývat za chvíli). Ale nejjednodušší způsob spočívá ve využití vlastností operačního systému.

Vyzkoušejte si, jak funguje jeden z příkazů operačního systému, když na příkazovém řádku předepíšeme přesměrování:

```
C:\> dir
C:\> dir > dir.txt
```

První z příkazů vypíše obsah adresáře na obrazovku, druhý jej zapíše do souboru. Použitím znaku '>' programu říkáme, že chceme stdout přesměrovat do souboru dir.txt.

Totéž bychom mohli udělat s našim pythonovským programem:

```
$ python mujprogram.py > vysledek.txt
```

Poznámka překladatele: Vyzývací znak '$' v uvedeném příkladu napovídá, že jde o použití v unixovém systému (například v systému Linux). V systému MS Windows bychom to udělali úplně stejně.

Uvedený zápis vede ke spuštění mujprogram.py, ale jeho výstup by se místo na obrazovce objevil v souboru vysledek.txt. Jeho obsah si poté můžeme prohlédnout prostřednictvím nějakého textového editoru.

K přesměrování stdin z nějakého souboru jednoduše místo znaku '>' použijeme znak '<'. Následuje úplný příklad. Nejdříve vytvoříme soubor opisvstupu.py a vložíme do něj následující zdrojový text:

```
import sys
vstup = sys.stdin.readline()
while vstup.strip() != '':
    print vstup
    vstup = sys.stdin.readline()
```

Poznámka: Metoda strip() odsekne znak, který reprezentuje přechod na nový řádek. Ten je součástí načítaného řádku ze stdin. Příkaz raw_input() by to udělal za vás.

Poznámka překladatele: Metoda strip() ve skutečnosti odstraňuje všechny *bílé znaky* (whitespaces) ze začátku i z konce řetězce. To znamená, že odstraní všechny úvodní a koncové mezery, tabulátory a znaky přechodu na nový řádek. Při použití metody readline() ovšem můžeme získat řetězec, ve kterém se znak přechodu na nový řádek vyskytuje jenom jednou a vždy zcela na konci.

Teď si to můžeme vyzkoušet spuštěním z příkazového řádku:

```
$ python opisvstupu.py
```

Výsledkem by měl být program, který vše opisuje na výstup, dokud nezadáte prázdný řádek.

Teď si vytvořte jednoduchý textový soubor nazvaný vstup.txt, který bude obsahovat pár řádků textu. Spusťte program znovu a přesměrujte do něj vstup ze souboru vstup.txt:

```
$ python opisvstupu.py < vstup.txt
```

Python opisuje na výstup vše, co našel v souboru vstup.txt. Vzpomínáte si, že jsme si řekli, že příkaz print a standardní funkce raw_input() ve skutečnosti vnitřně pracují se stdin a stdout? To znamená, že v příkladu opisvstupu.py můžeme práci se stdin nahradit voláním raw_input() následovně:

```
vstup = raw_input()
while vstup != '':
    print vstup
    vstup = raw_input()
```

... což je ve většině případů mnohem jednodušší.

Postupným přesměrováním vstupu z více různých souborů můžeme rychle a snadno otestovat chování našich programů v různých situacích (například při zadávání špatných hodnot nebo údajů špatného typu). Můžeme tak učinit opakovatelným a spolehlivým způsobem. Techniku přesměrování můžeme použít také při zpracování velkých objemů dat z připravených souborů. Přitom se při používání stejného programu nezbavujeme možnosti ručního vstupu dat v případech, kdy je objem dat malý. Mechanismus přesměrování stdin a stdout představuje pro programátora velmi užitečný trik. Zkoušejte a uvidíte sami, jaká další použití se vám podaří najít.

V systému Windows se vyskytuje známá chyba, která se projevuje při přesměrování vstupu. Pokud svůj program spustíte prostým uvedením jména skriptu, místo abyste před něj explicitně napsali python, nebudou Windows vypisovat výsledky v konzolovém okně! Na webové stránce firmy Microsoft naleznete popis zásahu do registry, který tuto chybu opravuje. Ani tento popis však není zcela korektní. Musíte se dívat pod HKEY_CURRENT_USER a ne pod HKEY_LOCAL_MACHINE, jak vám radí zmíněný dokument. Pokud pracujete s přesměrováním vstupu nebo výstupu, pak raději uveďte přímé spuštění programu python. [Děkuji Timu Graberovi, který si uvedeného problému všimnul, a Timu Petersovi, který mi prozradil onu opravu zásahem do registry.]

Parametry z příkazového řádku

Další typ vstupu představuje vstup zadaný z příkazového řádku. Dejme tomu, že nějak takto spustíte svůj textový editor z příkazového řádku:

```
$ EDIT Foo.txt
```

Operační systém spustí program nazvaný EDIT a předá mu jméno souboru, který se má editovat — v našem případě Foo.txt. Ale jak si editor přečte jméno souboru?

Ve většině jazyků systém poskytuje pole nebo seznam řetězců, které obsahují slova z příkazového řádku. Takže první prvek bude obsahovat jméno příkazu, druhý prvek bude obsahovat první argument, atd. Někdy máme k dispozici další magickou proměnnou (často je pojmenována argc z anglického *argument count* — počet argumentů), ve které je uložen počet prvků uvedeného seznamu.

V systému Python je zmíněný seznam součástí modulu sys a nazývá se argv (z anglického argument values, čili hodnoty argumentů). V Pythonu nepotřebujeme znát hodnotu argc, protože počet předaných argumentůmůžeme zjistit jako délku (počet prvků) seznamu argv pomocí standardní funkce len(). Většinou nepotřebujeme dělat ani to, protože průchod celým seznamem můžeme předepsat pythonovským cyklem fortakto:

```
import sys
for polozka in sys.argv:
    print polozka

print u'První argument byl:', sys.argv[1]
```

Poznamenejme, že to bude fungovat jen v případě, kdy uvedený kód uložíte do souboru (dejme tomu args.py) a spustíme jej z příkazového řádku operačního systému, například takto:

```
C:\Python\Projekty> python args.py 1 23 fred
args.py
1
23
fred
První argument byl: 1
C:\Python\Projekty>
```

VBScript a JavaScript

Tyto jazyky jsou určeny pro tvorbu webových stránek, takže možnost načítání parametrů zadaných na příkazovém řádku u nich nepřipadá v úvahu. Pokud bychom je používali v prostředí Microsoft Windows Script Host, situace by se změnila. WHS nabízí možnost extrakce těchto argumentů z objektu WshArguments, který je naplněn v době běhu.

A to je k tématu uživatelského vstupu opravdu vše, co budeme v této učebnici potřebovat. Je to sice velmi primitivní, ale přesto s tím vystačíte při psaní užitečných programů. V počátcích systému Unix nebo u prvních osobních počítačů představovala tato podoba interakce s uživatelem jedinou dostupnou možnost. V programech s grafickým uživatelským rozhraním můžeme samozřejmě vstup načítat také. K tomu, jak se to dělá, se ale v této učebnici dostaneme mnohem později.

Zapamatujte si

- Pro čtení čísel můžeme použít input, pro čtení znaků či řetězců použijeme raw_input.
- Jak příkaz input tak raw_input mohou zobrazit řetězec s výzvou pro uživatele.
- Parametry z příkazového řádku můžeme získat ze seznamu argv, který v systému Python můžeme importovat z modulu sys. První prvek seznamu obsahuje jméno programu.

Větvení, aneb nechť padne rozhodnutí

O čem si budeme povídat?

- Třetí programová konstrukce — větvení.
- Jednoduché a násobné větvení.
- Použití boolovských výrazů.

Třetím z našich základních stavebních kamenů je *větvení* nebo také *podmíněný příkaz*. Jde jednoduše o pojmy, které popisují schopnost provést jednu z několika možných posloupností příkazů (jednu z větví) a to v závislosti na nějaké podmínce.

Dříve, v době programování v assembleru, bylo větvení realizováno nejjednodušším možným způsobem — použitím instrukce JUMP. Program v tomto místě doslova skočil na určenou adresu v paměti. Obvykle to bylo podmíněno tím, že výsledkem předchozí instrukce byla nula. I když nebylo možné použít jiný způsob realizace podmíněného příkazu, byly takto napsány úžasně složité programy. To potvrzovalo správnost Dijkstrových tvrzení o minimálních požadavcích

potřebných pro programování. Když se objevily vyšší programovací jazyky, objevila se i nová podoba instrukce JUMP pod názvem GOTO. V jazyce QBASIC, který byl dodáván na instalačních CD ROM starších verzí Windows (před XP), lze GOTO stále používat. Pokud máte QBASIC nainstalován, můžete si vyzkoušet následující úsek kódu:

```
10 PRINT "Začínáme na řádku 10"
20 J = 5
30 IF J < 10 GOTO 50
40 Print "Tento řádek se nevytiskne"
50 STOP
```

Povšimněte si, že dokonce i u tak krátkého programu trvá několik sekund, než přijdete na to co se stane. Kód nemá žádnou strukturu. Musíte si ji během čtení doslova vytvořit. U velkých programů to začne být prakticky nemožné. Z tohoto důvodu většina moderních programovacích jazyků — včetně jazyků Python, VBScript a JavaScript — buď příkazy skoku JUMP nebo GOTO nemají, nebo vás od jejich používání odrazují. Takže co bychom vlastně místo nich měli použít?

Příkaz if

Intuitivně nejzřejmější podobou podmíněného příkazu je konstrukce if, then, else. Sleduje logiku anglické věty v tom smyslu, že *if* (jestliže) je nějaká boolovská podmínka splněna (o boolovských podmínkách se zmíníme dále v textu), *then* (pak) se provede blok příkazů, v opačném případě (nebo *else* (jinak)) se provede jiný blok.

Python

V jazyce Python vypadá zápis takto:

```
import sys  # jen proto, abychom mohli program ukončit
print "Začínáme zde"
j = 5
if j > 10:
print "Toto se nikdy nevytiskne"
else:
sys.exit()
```

Takový zápis se ve srovnání s předchozím příkazem s GOTO lépe čte a je srozumitelnější. Za slovo *if* můžeme samozřejmě dosadit libovolnou podmínku testu za předpokladu, že ji lze vyhodnotit jako True nebo False — to znamená jako boolovskou hodnotu. Zkuste změnit operátor > na < a pozorujte, co se stane.

VBScript

Zápis v jazyce VBScript vypadá podobně:

```
<script type="text/vbscript">
MsgBox "Začínáme zde"
Dim J
J = 5
If J > 10 Then
MsgBox "Toto se nikdy nevytiskne."
Else
MsgBox "Konec programu."
End If
</script>
```

Vždyť je to téměř shodné, že ano? Hlavní rozdíl spočívá v použití End If pro označení konce konstrukce.

A ještě v JavaScript

V jazyce JavaScript nalezneme samozřejmě příkaz if také:

```
<script type="text/javascript">
var j;
j = 5;
if (j > 10){
document.write("Toto se nikdy nevytiskne.");
}
else {
document.write("Konec programu.");
}
</script>
```

Povimněte si, že JavaScript používá uvnitř částí if a else pro vymezení bloku kódu složené závorky. Boolovský test je rovněž uzavřen v závorkách. Klíčové slovo then se zde nepoužívá. Co se týká stylu, složené závorky můžeme umístit na libovolnou pozici. Rozhodl jsem se, že je zarovnám pod sebe prostě proto, abych zdůraznil strukturu bloku. Pokud se v bloku nachází jen jediný řádek (jako v našem případě), můžeme závorky úplně vynechat. Potřebujeme je jen v případech, kdy mají ohraničit skupinu řádků, které patří do jednoho bloku.

Boolovské výrazy

Možná si ještě vzpomínáte, že jsme se v kapitole o datech zmínili o datovém typu *boolean*. Řekli jsme si, že má pouze dvě hodnoty: True a False. Boolovské proměnné vytváříme velmi zřídka[1], ale dočasné boolovské hodnoty často vznikají jako výsledek vyhodnocení *výrazů*. Výrazem rozumíme kombinaci proměnných a hodnot, spojených operátory s cílem vyprodukovat výslednou hodnotu. V následujícím příkladu

```
if x < 5:
print x
```

je zápis x < 5 výrazem. Pokud je x menší než 5, bude jeho výsledkem hodnota True. Pokud je x větší nebo rovno 5, bude výsledkem výrazu hodnota False.

Výrazy mohou být libovolně složité s tím, že výsledkem jejich vyhodnocení musí být nakonec vždy jediná hodnota. V případě větvení musí být výsledkem pravdivostní hodnota *True* nebo *False*. Nicméně, definice těchto dvou pravdivostních hodnot se jazyk od jazyka liší. V mnoha jazycích je hodnota *false*[2] ztotožněna s hodnotou 0 nebo s hodnotou vyjadřující neexistenci (té se často říká NULL, Nil nebo None). Takže v boolovském kontextu bude například prázdný seznam nebo prázdný řetězec vyhodnocen jako *False*. Takovým způsobem se chová i Python. To znamená, že například můžeme využít cyklu while pro zpracování seznamu, které má skončit v okamžiku, kdy je seznam prázdný:

```
while seznam:
# Proveď nějakou operaci, která vede ke zkrácení seznamu.
```

V příkazu if můžeme tento obrat použít k testování prázdnosti seznamu, aniž bychom použili funkci len():

```
if seznam:
# něco zde udělej (seznam je prázdný)
```

Boolovské výrazy můžeme kombinovat pomocí boolovských operátorů. Často tím můžeme zmenšit počet příkazů if. Uvažujme následující příklad:

```
if hodnota > maximum:
print "Hodnota je mimo rozsah!"
else:
if hodnota < minimum:
print "Hodnota je mimo rozsah!"
```

Povšimněte si, že blok prováděného kódu je v obou případech shodný. Zkombinováním obou testů do jednoho dosáhneme úspory práce jak pro počítač, tak pro nás:

```
if (hodnota > maximum) or (hodnota < minimum):
print "Hodnota je mimo rozsah!"
```

Oba testy jsme spojili operátorem or (*nebo*, čili logickým součtem). Dostáváme jediný výraz. Python nejdříve vyhodnotí výraz uzavřený v první dvojici závorek, potom výraz v druhých závorkách a nakonec vypočtené hodnoty zpracuje do podoby jediné hodnoty — True nebo False.

Poznámka překladatele: Výše uvedený odstavec chápejte spíše z obecného pohledu. Python ve skutečnosti zpracovává boolovský výraz zleva doprava a skončí v okamžiku, kdy už následující části výrazu nemohou ovlivnit výsledek. Říká se tomu zkrácené vyhodnocování výrazu. Pokud v uvedeném příkladu získáváme vyhodnocením první závorky hodnotu True, pak při použití operátoru or nemá výsledek vyhodnocování druhé závorky na celkový výsledek vliv. Proto se vůbec neprovádí.

Zkráceného vyhodnocování boolovských výrazů se většinou spíš výhodně využívá. Mohou však nastat případy, kdy díky tomuto jevu vzniká obtížněji odhalitelná chyba. Pokud bychom v místě druhé části výrazu volali funkci, která vrací boolovský výsledek, ale kromě toho má nějaký vedlejší efekt (například něco vypisuje), pak musíme myslet na to, že se také nemusí vůbec zavolat. Na dosažení zmíněného vedlejšího efektu proto nemůžeme při vyhodnocování výrazu spoléhat.

Pokud o prováděných testech uvažujeme v pojmech přirozeného jazyka, velmi často používáme spojky jako *a*(anglicky and), *nebo* (or), *negace* (ne, není, anglicky not). V takovém případě je velmi pravděpodobné, že se nám místo více jednoduchých testů podaří zapsat jeden složený.

Poznámka překladatele: Pokud uvažujeme v českém jazyce, pak vám doporučuji, abyste si operátor andpřekládali jako *a zároveň*. Pouhé *a* může vést k chybám, kdy tuto spojku můžeme chápat ve významu *nebo*.

Překlad *a zároveň* zdůrazní význam operátoru *and* a pomůže nám snadněji vytvořit mentální obraz situace.

Zřetězení příkazů if

Příkazy if/then/else můžeme do sebe *vnořovat*. V jazyce Python to můžeme vyjádřit následovně:

```
# Předpokládáme, že cena byla předem stanovena...
cena = int(raw_input(u"Kolik to stojí? "))
if cena == 100:
print u"Vezmu si to."
else:
if cena > 500:
print u"Tak to nechci ani náhodou!"
else:
if cena > 200:
print u"Co kdybyste přihodil zdarma podložku pod myš?"
else:
print u"Neočekávaná cena."
```

Poznámka 1: V prvním příkazu if jsme pro *test na rovnost* použili operátor == (tj. zdvojený znak =). Jednoduchý znak = se používá pro přiřazování hodnot proměnným. Při programování v Pythonu (a také v C a v C++) patří použití jednoduchého = v místě, kde bychom chtěli použít == k nejčastějším chybám. Python vás v takovém případě naštěstí varuje, že jste se dopustili syntaktické chyby. Někdy se ale musíte pořádně podívat, než si všimnete, o co vlastně jde.

Poznámka 2: Za povšimnutí stojí ještě jeden detail. Testy *větší než* provádíme od největší hodnoty k nejmenší. Kdybychom postupovali obráceně a začali bychom testem cena > 200, pak bychom se nikdy nedostali k testucena > 500. Při používání po sobě jdoucích testů *menší než* musíme naopak začít testovat na nejmenší hodnotu a postupovat směrem k hodnotám vyšším. Jde o další past, do které se můžeme při troše nepozornosti snadno chytit.

VBScript & JavaScript

Příkazy if můžeme řetězit i v jazycích VBScript a JavaScript. Postup je zcela zřejmý. Proto si to ukážeme jen na příkladu v jazyce VBScript:

```
<script type="text/vbscript">
DIM Cena
cena = InputBox("Kolik to stojí?")
cena = CInt(cena)
If cena = 100 Then
MsgBox "Vezmu si to."
Else:
if cena > 500 Then
MsgBox "Tak to nechci ani náhodou!"
else:
if cena > 200 Then
MsgBox "Co kdybyste přihodil zdarma podložku pod myš?"
else:
MsgBox "Neočekávaná cena."
End If
End If
End If
</script>
```

Za zmínku zde stojí jedině to, že ke každému příkazu if musíme uvést odpovídající příklaz End If. Poznamenejme ještě, že pro převod řetězcové hodnoty na celočíselnou jsme použili funkci CInt().

Příkazy typu Case

S používáním zanořených příkazů if/else souvisí jedna potíž. Postupné odsazování způsobí, že se zdrojový text rychle roztáhne přes celou šířku stránky. Posloupnost zanořených if/else/if/else… však patří k tak běžným konstrukcím, že některé jazyky poskytují speciální způsob větvení.

Zmíněné speciální konstrukce se často označují jako příkazy *case* nebo *switch*. V jazyce JavaScript vypadá příkaz switch následovně:

```
<script type="text/javascript">
  function vypoctiPlochu() {
  var tvar, sirka, delka, plocha;
tvar   = document.plocha.tvar.value;
sirka  = parseInt(document.plocha.sirka.value);
delka  = parseInt(document.plocha.delka.value);
switch (tvar) {
case 'ctverec':
plocha = delka * delka;
alert("Plocha tvaru " + tvar + " = " + plocha);
break;
case 'obdelnik':
plocha = delka * sirka;
alert("Plocha tvaru " + tvar + " = " + plocha);
break;
case 'trojuhelnik':
plocha = delka * sirka / 2;
alert("Plocha tvaru " + tvar + " = " + plocha);
break;
default: alert("Neznámý tvar: " + tvar)
};
}
</script>

<form name="plocha">
Délka:  <input type="text" name="delka">
Šířka:  <input type="text" name="sirka">
Tvar:   <select name="tvar" size="1" onChange="vypoctiPlochu()">
<option value="ctverec">čtverec
<option value="obdelnik">obdélník
<option value="trojuhelnik">trojúhelník
</select>
</form>
```

Detaily jsou zachyceny v rámci HTML kódu formuláře. Jakmile si uživatel vybere tvar, zavolá se naše funkce. Na prvních řádcích se vytvářejí lokální proměnné a podle potřeby se řetězce převádějí na čísla. Zajímá nás úsek, který je vyznačen tučně. Podle vybraného tvaru se v něm vybírá příslušná akce. Povšimněte si kulatých závorek kolem identifikátoru tvar za klíčovým slovem switch. Jsou povinné — musí být uvedeny. Mohlí byste předpokládat, že bloky kódu uvnitř case by měly být uzavřeny do složených závorek, ale není tomu tak. Místo toho jsou ukončovány příkazem break. Nicméně celá sada příkazů case, která odpovídá části switch, již je svázána do podoby bloku jedním párem složených závorek.

Povšimněte si, že poslední podmínka v příkladu má podobu default. V této části se zachytí všechny případy, které se nezachytily v předchozích částech case.

Vyzkoušejte si, zda byste uměli výše uvedený příklad rozšířit tak, aby pracoval i s kruhem. Do HTML formuláře nezapomeňte přidat novou volbu a do příkazu switch přidejte další variantu case.

Příkaz Select Case v jazyce VBScript

Verzi příkazu pro výběr jedné z několika variant nalezneme i v jazyce VBScript:

```
<script type="text/vbscript">
Dim tvar, delka, sirka, CTVEREC, OBDELNIK, TROJUHELNIK
CTVEREC = 0
OBDELNIK = 1
TROJUHELNIK = 2
tvar  = CInt(InputBox("Čtverec(0), obdélník(1) nebo trojúhelník(2)?"))
delka = CDbl(InputBox("Délka?"))
sirka  = CDbl(InputBox("Šířka?"))
Select Case tvar
Case CTVEREC
plocha = delka * delka
MsgBox "Plocha = "  & plocha
Case OBDELNIK
plocha = delka * sirka
MsgBox "Plocha = " & plocha
Case TROJUHELNIK
plocha = delka * sirka / 2
MsgBox "Plocha = " & plocha
Case Else
MsgBox "Neznámý tvar"
End Select
</script>
```

Na několika prvních řádcích se od uživatele získávají data a převádějí se na správný typ — stejně, jako tomu bylo u příkladu v jazyce JavaScript. Tučně vyznačená část Select znázorňuje konstrukci typu case, jak se používá v jazyce VBScript. Zasebou uvedené příkazy Case vždy ukončují blok předchozího. Celou konstrukci Select uzavírá

příkaz End Select. Nalezneme zde také část Case Else, ve které se (jako v části default jazyka JavaScript) zachytí vše, co nebylo zpracováno dříve uvedenými částmi Case.

Za zmínku stojí ještě použití *symbolických konstant* místo čísel. Proměnné zapsané velkými písmeny CTVEREC, OBDELNIK a TROJUHELNIK jsou zde jen kvůli tomu, aby se zdrojový text snadněji četl. Použití proměnných zapsaných velkými písmeny je předepsáno pouze konvencí. Dáváme tím najevo, že bychom je neměli chápat jako běžné proměnné, ale jako proměnné udržující konstantní hodnoty. Jazyk VBScript vám ale dovolí pojmenovat si proměnné podle své libosti.

case v jazyce Python

Python explicitní konstrukci typu *case* nepodporuje. Místo toho nabízí kompromis v podobě if/elif/else:

```
menu = """
Vyberte si tvar (1-3):
1) Ctverec
2) Obdelnik
3) Trojuhelnik
"""
tvar = int(raw_input(menu))
if tvar == 1:
strana = float(raw_input("Strana: "))
print "Plocha ctverce = ", strana ** 2
elif tvar == 2:
delka = float(raw_input("Delka: "))
sirka = float(raw_input("Sirka: "))
print "Plocha obdelniku = ", delka * sirka
elif tvar == 3:
zakladna = float(raw_input("Zakladna: "))
vyska =    float(raw_input("   Vyska: "))
print "Plocha trojuhelniku = ", zakladna * vyska / 2
else:
print "Neplatny tvar. Zkute to znovu"
```

Poznámka překladatele: Abychom se zatím vyhnuli problémům s českými znaky, použili jsme texty bez diakritických znamének. Způsob řešení, kdy používáme i české znaky s diakritikou, můžete nalézt v dalších kapitolách.

Povšimněte si použití elif a skutečnosti, že se (v porovnání s příkladem se zanořenými if) nemění odsazení, které je v Pythonu tak důležité. Za zmínku stojí i to, že *oba* zápisy — jak poslední zápis, tak dříve uvedený zápis využívající vnořených konstrukcí if/else — jsou funkčně shodné. Zápis využívající elif zvyšuje čitelnost v případech, kdy použijeme větší množství testů. V koncové větvi else se zachytí všechny případy, které nebyly zachyceny v předchozích testech. Odpovídá to použití default v JavaScript nebo Case Else v jazyce VBScript.

O něco těžkopádnější podobu stejné konstrukce naleznete i v jazyce VBScript. Konstrukce ElseIf...Then se používá naprosto stejným způsobem, jako elif v jazyce Python. Ale setkáte se s ní zřídka, protože použití alternativního příkazu Select Case je jednodušší.

Teď to dáme všechno dohromady

Až dosud byly mnohé z našich příkladů velmi abstraktní. Na závěr se podívejme na příklad, který používá téměř vše, co jsme se zatím naučili. Uvedeme si běžnou programovací techniku, konkrétně zobrazení menu pro řízení uživatelského vstupu.

Zde máme kód, za kterým následuje krátká diskuse.

```
menu = """
Vyberte si tvar (1-3):
1) Ctverec
2) Obdelnik
3) Trojuhelnik

4) Konec
"""
tvar = int(raw_input(menu))
while tvar != 4:
if tvar == 1:
strana = float(raw_input("Strana: "))
print "Plocha ctverce = ", strana ** 2
elif tvar == 2:
delka = float(raw_input("Delka: "))
sirka = float(raw_input("Sirka: "))
print "Plocha obdelniku = ", delka * sirka
elif tvar == 3:
zakladna = float(raw_input("Zakladna: "))
vyska =    float(raw_input("   Vyska: "))
print "Plocha trojuhelniku = ", zakladna * vyska / 2
else:
print "Neplatny tvar. Zkute to znovu"
tvar = int(raw_input(menu))
```

K předchozímu příkladu jsme přidali pouhé tři řádky (označeny tučně), ale tato jednoduchá úprava výrazně zvýšila použitelnost našeho programu. Doplněním volby Konec a přidáním cyklu jsme uživateli umožnili pokračovat ve výpočtech ploch různých tvarů až do doby, kdy získá všechny potřebné informace. Program již nemusí pokaždé ručně znovu a znovu spouštět. Kromě již zmíněných řádků jsme přidali pouze jeden řádek s raw_input(menu), který slouží k opakovanému výběru tvaru. To znamená že uživatel může volit různé tvary a na závěr také činnost programu ukončit.

Program tedy uživateli vytváří iluzi, že ví, co uživatel potřebuje. Na základě jeho volby se chová různým způsobem a správně provede odpovídající činnost. Uživateli se v podstatě zdá, že postup řídí, zatímco ve skutečnosti má řízení v rukou

programátor, který předvídal, jak mají vypadat všechny platné vstupy a jak má na ně program reagovat. Projevovaná inteligence tedy patří programátorovi, nikoliv stroji. Počítače jsou ve své podstatě hloupé!

Povšimněte si jak snadno můžeme svůj program zdokonalit přidání pouhých pár řádků a zkombinováním posloupností (bloků pro výpočet plochy), cyklů (zde cyklus while) a podmíněných příkazů (konstrukceif/elif/else). Jde o tři z Dijkstrových základních programátorských stavebních kamenů. Pokud zvládnete všechny tři, můžete teoreticky naprogramovat cokoliv. Ale můžeme se naučit ještě několik technik, které nám programování dále usnadní. Takže zatím mějte ještě trochu strpení.

Změny v kolekci během provádění cyklu

Když jsme se bavili o cyklech, zmínili jsme se o tom, že úprava kolekce během průchodu cyklem, konkrétné rušení prvků v procházené kolekci, nemusí být zcela jednoduché. Ale nevysvětlili jsme proč! Důvodem pro tento odklad byla skutečnost, že jsme museli nejdříve vysvětlit pojem *větvení*. Vraťme se tedy k řešení problému.

Pokud potřebujeme měnit obsah kolekce během jejího zpracování (bez kopírování do jiné kolekce), můžeme k tomu využít vlastností cyklu while. Při použití konstrukce while totiž přímo pracujeme s obsahem indexové proměnné. Srovnejte to se situací, kdy se použije cyklus for, který indexovou proměnnou upravuje automaticky. Podívejme se, jak můžeme ze seznamu vypustit všechny prvky s nulovou hodnotou:

```
seznam = [1, 2, 3, 0, 4, 5, 0]
index = 0
while index < len(seznam):
if seznam[index] == 0:
del seznam[index]
else:
index += 1
print seznam
```

Povšimněte si, že v případě odstraňování prvku *neprovádíme zvyšování indexu*. Spoléháme na to, že se při smazání položky vše posune, takže původní hodnota indexu bude poté ukazovat na další prvek kolekce. Zvyšování indexu se tedy provádí jen v jedné větvi konstrukce if/else. Při podobných obratech se můžeme velice snadno dopustit chyby, proto vždy funkčnost pečlivě otestujte.

V Pythonu můžeme používat jistou sadu funkcí, které byly přímo navrženy pro manipulaci s obsahy senamů. Seznámíme se s nimi v rámci tématu Funkcionální programování, tedy v části pro pokročilé.

Zapamatujte si

- Pro větvení používáme if/else.
- Část else je nepovinná.
- K rozhodnutí podle více z možných hodnot lze využít konstrukci typu Case nebo konstrukci if/elif.
- Boolovské výrazy vracejí hodnoty True nebo False.
- Kombinování menu s konstrukcemi Case nám umožňuje vytvářet širokou škálu aplikací řízených uživatelem.

Programování s využitím modulů

O čem si budeme povídat?

- Co to je modul?
- Funkce jako modul.
- Používání souborů s moduly.
- Zápis našich vlastních funkcí a modulů.
- Úvod do Windows Script Host.

Co to je modul?

Čtvrtý element programování do hry zapojuje použití *modulů* (viz Společné vlastnosti programů). Ve skutečnosti není naprosto nezbytné a i bez něj, s použitím toho, o čem jsme se zatím bavili, můžete vytvořit docela působivé programy. Jenomže s tím, jak se program stává větším, stává se stále obtížnějším sledování toho, co a kde se v něm děje. Potřebujeme najít způsob jak potlačit nutnost uvažování o některých detailech, abychom mohli uvažovat o řešeném problému a abychom nemuseli řešit vše, co souvisí s fungováním počítače. Python, VBScript a JavaScript a další jazyky to do jisté míry řeší již svými zabudovanými schopnostmi — nemusíme se například zabývat detaily počítačového technického vybavení (hardware), nemusíme se zabývat tím, jak se čtou stisky jednotlivých kláves na klávesnici a podobně.

Používání modulů má programátorovi umožnit *rozšiřování* zabudovaných schopností jazyka. Zabývá se obalováním kousků programů do modulů tak, abychom je mohli využít v našich programech. První formou modulu byl *podprogram*, což byl blok kódu, na který bylo možno skočit (něčím jako příkaz GOTO, o kterém jsme se zmínili v kapitole o větvení). Jakmile byl blok dokončen, bylo možné skočit zpět do místa odkud byl *volán*. Tato konkrétní podoba modularity je známa pod pojmy *procedura* nebo *funkce*. V jazyce Python a v některých dalších jazycích se slovo *modul* spojilo s přesněji vymezeným významem, o kterém si za chvíli něco povíme. Ale nejdříve se podívejme podrobněji na funkce.

Používání funkcí

Dříve, než se budeme zabývat tím, jak můžeme funkci vytvořit, podívejme se, jak používáme velké množství funkcí, které se k libovolnému programovacímu jazyku dodávají (často se dodávají v podobě takzvaných*knihovny*).

Použití některých funkcí a seznam dalších funkcí jsme již viděli, když jsme se bavili o operátorech. Nyní se zaměříme na to, co mají všechny funkce společné a jak je můžeme používat v našich programech. Základní struktura *volání funkce* vypadá následovně:

```
Promenna = nejakaFunkce(argument, dalsi, atd...)
```

To znamená, že proměnné Promenna je přiřazena hodnota, která je získána jako výsledek volání funkce. Funkce může mít žádný nebo více *argumentů*, ke kterým se chová jako k interním proměnným. Funkce mohou uvnitř svého kódu volat další funkce. Ve většině programovacích jazyků (i když ne ve všech) musíme v zápisu volání funkce uvádět závorky i v případě, kdy nemá žádné argumenty.

Vezměme si některé příklady v našich vybraných jazycích a podívejme se, jak to funguje:

VBScript: Mid(retezec, start, delka)

Tato funkce vrací delka znaků z řetězce retezec, počínaje pozicí start.

```
<script type="text/vbscript">
Dim cas
cas = "ráno večer odpoledne"
MsgBox "Dobrý " &  Mid(cas, 6, 5)
</script>
```

Zobrazí se "Dobrý večer". Poznamenejme, že u VBScript se argumenty funkce nemusí uzavírat do závorek. Stačí oddělení mezerou, jak jsme učinili při volání MsgBox. Ovšem pokud kombinujeme volání dvou funkcí (jako v našem případě), pak vnitřní funkce musí používat závorky. Moje rada zní: pokud jste na pochybách, používejte závorky.

VBScript: Date

Tato funkce vrací aktuální systémové datum.

```
<script type="text/vbscript">
MsgBox Date
</script>
```

Asi už se k tomu nedá víc dodat. Snad jen, že existuje celá skupina funkcí pro práci s datem, které umožňují získat informaci o dni, týdnu, hodině atd.

JavaScript: startString.replace(searchString, newString)

Vrací nový řetězec, který vznikne z řetězce startString nahrazením podřetězce searchString novým řetězcemnewString.

```
<script type="text/javascript">
var r, s = "Dlouhá a vítězná cesta.";
document.write("Originál = " + s + "<BR>");
r = s.replace("Dlouhá", "Krátká");
document.write("Výsledek = " + r);
</script>
```

Poznámka: V jazyce JavaScript se téměř všude setkáváme se speciálním typem funkce, které říkáme *metoda*. Metodou rozumíme funkci, která je spojena s *objektem* (viz dříve probírané téma Data, datové typy a proměnnéa viz více detailů dále). Za poznámku zde stojí především to, že uvedená funkce je *připojena* k řetězci stečkovým operátorem. Chápe se to tak, že s je řetězcem, nad kterým se provádí náhrada.

Není to nic nového. Už od začátku této učebnice používáme pro zobrazení výstupu v našich javascriptových programech metodu write() objektu document (zapisujeme document.write()). Zatím jsem se pouze nezmiňoval o důvodech, které jsou s touto formou zápisu spojeny.

Python: pow(x,y)

Funkce pow() umocňuje základ x na y.

```
>>> x = 2              # použijeme 2 jako hodnotu základu
>>> for y in range(0, 11):
...     print pow(x, y)  # umocní 2 na y, tj. na 0 až 10
```

V tomto příkladu generujeme hodnoty y od 0 do 10 a voláme zabudovanou funkci pow() s předáváme jí dva argumenty: x a y. Aktuální hodnoty x a y jsou do volání funkce pow() dosazeny při každé obrátce cyklu a výsledek je pokaždé vytištěn.

Poznámka: Python definuje také operátor pro umocňování **, který je ekvivalentem funkce pow().

Python: dir(m)

Další užitečnou zabudovanou funkcí jazyka Python je funkce dir(). Když jí předáme jméno modulu, vrátí nám seznam všech jmen, která modul exportuje, včetně všech proměnných a funkcí, které můžeme použít. O modulech jsme se až do tohoto okamžiku nezmiňovali, ale Python se dodává spolu s velkým množstvím modulů. Vypišme si funkcí dir() seznam jmen zabudovaných funkcí:

```
print dir(__builtins__)
```

Poznámka 1: builtins je jedním z oněch *magických* slov systému Python. Takže je opět obklopeno dvojicemi znaků podtržení — dvě podtržení na každé straně.

Poznámka 2: Pokud budete chtít funkci dir() použít pro libovolný jiný modul, musíte jej nejdříve zpřístupnit příkazem import. V opačném případě si Python bude stěžovat, že jméno modulu nezná.

```
>>> import sys
>>> dir(sys)
```

Vzpomeňte si, že s modulem sys jsme se setkali již v našich prvních tématech, věnovaných posloupnostem. Ve výstupním seznamu posledního volání funkce dir() byste si měli všimnout jména naší staré známé funkce exit().

Dříve, než se pustíme do dalších věcí, měli bychom si o modulech v jazyce Python říci více podrobností.

Používání modulů

Python je extrémně rozšiřovatelný jazyk v tom smyslu, že můžete přidávat nové možnosti tím, že provedete import potřebných modulů. Zakrátko si ukážeme, jak můžeme moduly vytvářet. Ale nejdříve si trochu pohrajeme z některými standardními moduly, se kterými se Python již dodává.

sys

S modulem sys jsme se již potkali v okamžiku, kdy jsme z kódu v jazyce Python volali funkci exit za účelem ukončení programu. Modul obsahuje celou řadu dalších užitečných funkcí — jak jsme si mohli povšimnout ve výstupu funkce dir ve výše uvedeném příkladu. Abychom k nim získali přístup, musíme provést příkazimport sys:

```
import sys          # zpřístupní vnitřní funkce
print sys.path()    # ukaž, kde Python hledá moduly
sys.exit()          # uvádíme předponu 'sys'
```

Pokud víme, že budeme funkce modulu používat velmi často a pokud nemají stejná jména jako funkce, které jsme již dříve importovali nebo vytvořili, pak můžeme psát:

```
from sys import *  # import všech jmen z modulu sys
print path()       # nyní nemusíme uvádět předponu 'sys'
exit()
```

Pokud chceme z modulu používat jen pár funkcí, pak bývá bezpečnější použít následující obrat:

```
from sys import path, exit # importujeme jen potřebné
print path()               # používáme bez předpony 'sys'
exit()
```

Povšimněte si, že za jmény vybraných funkcí neuvádíme závorky. Pokud bychom tak učinili, bylo by to chápáno jako pokus o spuštění funkcí. Uvádíme pouze jméno funkce.

Poznámka překladatele: příkaz from...import... je obecný a dá se použít pro zveřejnění různých jmen definovaných v modulu, nejen jmen funkcí. Vynechávání závorek je proto logické i z tohoto pohledu.

Na závěr bych vám rád ukázal trik, který vám může ušetřit psaní. Pokud používáte modul s velmi dlouhým jménem, můžete jej při importování přejmenovat. Příklad:

```
import SimpleXMLRPCServer as m
```

```
m.SimpleXMLRPCRequestHandler()
```

Pythonu jsme zde řekli, aby považoval m za zkratku pro modul SimpleXMLRPCServer. Pokud nyní chceme použít nějakou funkci z tohoto modulu, budeme před ni psát pouze m., což je mnohem kratší.

Další moduly jazyka Python a jejich obsah

Uvedeným způsobem můžeme importovat libovolné pythonovské moduly. Týká se to i modulů, které si sami vytvoříte. Za okamžik si ukážeme, jak na to. Ale nejdříve si uděláme krátkou přehlídku některých standardních modulů jazyka Python a představíme si něco z toho, co nabízejí:

Jméno modulu	Popis
sys	Umožňuje interakci se systémem Python: • exit() — ukončení běhu programu • argv — seznam argumentů z příkazového řádku • path — seznam cest prohledávaných při práci s moduly • ps1 — mění vyzývací řetězec '>>>' příkazového řádku interpretu jazyka Python
os	Umožňuje interakci s operačním systémem: • name — zkratka charakterizující používaný operační systém; užitečná při psaní přenositelných programů • system — provedení příkazu systému • mkdir — vytvoření adresáře • getcwd — zjistí současný pracovní adresář (z anglického **get c**urrent **w**orking **d**irectory)
re	Umožňuje manipulaci s řetězci předepsanou regulárními výrazy, jaké se používají v systému Unix: • search — hledej vzorek kdekoliv v řetězci • match — hledej pouze od začátku řetězce • findall — nalezne všechny výskyty vzorku v řetězci • split — rozděl na podřetězce, které jsou odděleny zadaným vzorkem • sub, subn — náhrada řetězců
math	Zpřístupňuje řadu matematických funkcí: • sin, cos, atd. — trigonometrické funkce • log, log10 — přirozený a dekadický logaritmus • ceil, floor — zaokrouhlení na celé číslo nahoru a dolů • pi, e — konstanty
time	Funkce pro práci s časem a datem: • time — vrací současný čas (vyjádřený v sekundách) • gmtime — převod času v sekundách na UTC (tj. na čas v univerzálních časových souřadnicích — známější pod zkratkou GMT z anglického Greenwich Mean Time, tedy greenwichský [grinidžský] čas) • localtime — převod do lokálního času (tj. posunutého vůči UTC o celé hodiny) • mktime — opačná operace k localtime • sleep — pozastaví běh programu na zadaný počet sekund
random	Generátory náhodných čísel — užitečné nejen pro programování her! • randint — generování náhodného čísla mezi dvěmi hranicemi (včetně) • sample — generování náhodného podseznamu z jiného seznamu • seed — počáteční nastavení klíče pro generování čísel

Uvedené funkce představují pouze špičku ledovce. V distribuci systému Python se nacházejí doslova tucty modulů a mnoho dalších si můžete stáhnout z Internetu (jejich dobrým zdrojem jsou stránky Vaults of Parnassus). Nahlédněte do dokumentace a najdete informace o funcích pro přístup na Internet, pro grafiku, pro tvorbu databází, atd.

Je důležité si uvědomit, že u většiny programovacích jazyků jsou tyto funkce buď zabudované nebo jsou součástí jejich standardní knihovny. Než začnete psát nějakou funkci, vždycky si v dokumentaci ověřte, zda již náhodou neexistuje. Tím jsme se pěkně dostali k tématu…

Definice našich vlastních funkcí

Nyní již víme, jak používat existující funkce a moduly. Ale jak můžeme vytvořit novou funkci? Jednoduše ji *definujeme*. To znamená, že napíšeme příkaz, který interpretu říká, že definujeme blok kódu, který může být na požádání proveden na jiném místě našeho programu.

Nejdříve VBScript

Vytvořme funkci, která nám vytiskne tabulku násobků libovolného čísla, které zadáme jako argument. V jazyce VBScript by to vypadalo takto:

```
<script type="text/vbscript">
Sub Nasobky(N)
    Dim I
    For I = 1 To 12
MsgBox I & " x " & N & " = " & I * N
    Next
End Sub
</script>
```

Za značkou uvádějící blok v jazyce VBScript jsme použili klíčové slovo Sub (jako *Subroutine*, čili podprogram) a celou definice jsme ukončili klíčovými slovy End Sub. V kulatých závorkách jsme uvedli seznam formálních *parametrů*. Uvnitř se nachází běžný kód v jazyce VBScript až na to, že se parametry chápou jako kdyby to byly již definované lokální proměnné.

Nově definovanou funkci můžeme od tohoto okamžiku volat třeba takto:

```
<script type="text/vbscript">
MsgBox "Následují hodnoty násobků čísla 7..."
Nasobky 7
</script>
```

Poznámka 1: V uvedené funkci jsme definovali *parametr* označený N a při jejím volání jsme předali *argumento* hodnotě 7. Když je funkce zavolána, pak její lokální proměnná N získá hodnotu 7. U funkce můžeme definovat tolik parametrů, kolik potřebujeme. Při jejím volání ale musí být každému parametru přiřazena hodnota. Některé programovací jazyky vám umožňují definovat *přednastavené hodnoty* parametrů (default value), takže pokud není jejich hodnota při volání určena, použije se přednastavená hodnota. (V odborné terminologii se používá pojem *implicitní* hodnota parametru, která se použije, pokud hodnota parametru není *explicitně* určena.)Za chvíli si ukážeme, jak se to dělá v jazyce Python.

Poznámka 2: V definici funkce je parametr N uzavřen v kulatých závorkách, ale — jak je u VBScript zvykem — při volání funkce jsme kulaté závorky nepoužili.

Uvedená funkce nevrací žádnou hodnotu a je ve skutečnosti tím, čemu se říká *procedura*. Je to jednoduše funkce, která nevrací hodnotu. Jazyk VBScript procedury a funkce rozlišuje. Syntakticky tak činí používáním jiného klíčového slova v definici. Podívejme se na zápis skutečné funkce zapsané v jazyke VBScript, která vrací tabulku násobku jako jeden dlouhý řetězec:

```
<script type="text/vbscript">
Function TabulkaNasobku(N)
Dim I, S
S = "Tabulka násobků čísla " & N & ":" & vbNewLine
For I = 1 to 12
S = S & I & " x " & N & " = " & I * N & vbNewLine
Next
TabulkaNasobku = S
End Function

Dim Nasobitel
Nasobitel = InputBox("Zadejte násobitele, pro který má být tabulka generována:?")
MsgBox TabulkaNasobku(Nasobitel)
</script>
```

Zápis funkce (používá klíčové slovo Function) je téměř stejný, jako zápis procedury (klíčové slovo Sub). Ale všimněte si, že uvnitř definice funkce musíme navíc výsledek přiřadit jménu funkce. Jakmile funkce skončí, vrátí hodnotu, která byla jejímu jménu přiřazena:

```
...
TabulkaNasobku = S
End Function
```

Pokud neprovedeme přiřazení konkrétní hodnoty, funkce vrátí implicitní hodnotu (default), což je obvykle nula nebo prázdný řetězec.

Povšimněte si také, že jsme na řádku MsgBox museli argument uzavřít do kulatých závorek. Kdybychom tak neučinili, funkce MsgBox by neuměla rozlišit, zda se má Nasobitel vytisknout, nebo zda se má předat funkci TabulkaNasobku jako první argument.

A teď v Pythonu

V jazyce Python by funkce pro tisk tabulky násobků vypadala takto:

```
def nasobky(n):
for i in range(1, 13):
print "%d x %d = %d" % (i, n, i*n)
```

A volali bychom ji takto:

```
print "Tady je tabulka násobků číslem 9 ..."
nasobky(9)
```

Poznámka překladatele: Místo funkce range() by se měla přednostně používat funkce efektivnější funkce xrange(). Funkce range() nejdříve vygeneruje celý seznam čísel a teprve ten se prochází v cyklu for. Funkce xrange() existenci seznamu při použití v cyklu for simuluje (vrací vždy další hodnotu), ale seznam nevytváří. Důležité je to zejména při generování velkého rozsahu indexů. Ve vzdálené budoucnosti se implementace range() změní na něco podobného xrange(). Do té doby používejte přednostně xrange(). V jednoduchých případech funguje stejně jako range(), v extrémních případech je výkonnější a paměťově nenáročná.

Povšimněte si, že v jazyce Python se mezi procedurami a funkcemi nedělá rozdíl. Při jejich definici se používá stejné klíčové slovo def. Jediný rozdíl spočívá v tom, že se k návratu funkční hodnoty používá příkaz return:

```
def tabulkaNasobku(n):
s = ""
for i in range(1, 13):
s = s + "%d x %d = %d\n" % (i, n, i*n)
return s
```

Jak vidíte, je to velmi jednoduché. Výsledek prostě vrátíme příkazem return. Výsledek funkce si můžeme vytisknout:

```
print tabulkaNasobku(7)
```

Implicitní hodnoty

Možná si vzpomenete, že jsem se již zmínil o použití implicitních hodnot (default). U funkcí můžeme předepsat implicitní hodnotu parametru, který nebyl při volání funkce zadán přímo (explicitně). Smysluplnou ukázku tohoto rysu může představovat funkce, která vrací den v týdnu. Pokud ji zavoláme bez parametru, míníme tím *dnes*. V ostatních případech předáváme požadovaný den číslem:

```
# -*- coding: cp1250 -*-
import time

# Hodnota dne -1 znamená 'dnes'.
def denTydne(CisloDne = -1):
# Výčet dnů musí být uveden v pořadí, v jakém je používá Python.
dny = [u'pondělí', u'úterý',
```

```
    u'středa', u'čtvrtek',
    u'pátek', u'sobota', u'neděle']

    # Zkontrolujeme, zda nejde o přednastavenou hodnotu.
    if CisloDne == -1:
        # Použij funkce modulu time k získání současného času
        # -- viz oficiální dokumentace modulu.
        cas = time.localtime(time.time())
        CisloDne = cas[6] # získáme číslo dne
    return dny[CisloDne]
```

Poznámka: Modul time potřebujeme použít jen v případě, kdy se uplatní přednastavená (implicitní) hodnota parametru. Operaci import bychom proto mohli odložit až na dobu, kdy modul skutečně potřebujeme. V případech, kdy přednastavená hodnota není nikdy použita, by pak náš program mohl být o něco výkonnější. Ale úspora výkonu by byla velmi malá a navíc bychom porušili konvenci, která říká, že import máme uvádět na začátku souboru. V podobných případech raději dáme přednost vyšší přehlednosti zdrojového textu před zanedbatelně vyšším výkonem při běhu.

Poznámka překladatele: Úvodní řádek `# -*- coding: cp1250 -*-` říká, že zdrojový text v souboru byl pořízen v kódování 1250 (přesněji windows-1250, ale Python vyžaduje uvedený tvar zápisu). Pokud jej neuvedeme, bude si Python při překladu stěžovat, že nalezl znaky s kódem větším než 128 (tj. mimo ASCII).

Nyní můžeme funkci zavolat:

```
print "Dnes je %s." % denTydne()
# Zapamatujte si, že v počítačové řeči začínáme nulou.
# Nula odpovídá prvnímu dni -- pondělí.
print "Třetím dnem je %s." % denTydne(2)
```

Počítání slov

Jako další příklad funkce, která vrací hodnotu, si uvedeme funkci počítající slova v zadaném řetězci. Mohli bychom ji použít i pro zjištění počtu slov v souboru tak, že bychom sečetli počty slov za jednotlivé řádky. Kód by mohl vypadat nějak takto:

```
def pocetSlov(s):
    s = s.strip()      # odstraníme mezery na začátku a na konci
    seznam = s.split() # seznam, kde prvkem je vždy slovo
    return len(seznam) # vrátíme počet prvků seznamu
```

Tím jsme nadefinovali funkci, která využívá některé metody zabudovaného typu řetězec. Zmínili jsme se o něm v kapitole Data, datové typy a proměnné. Teď bychom funkci rádi použili nějak takto:

```
for radek in soubor:
    celkem = celkem + pocetSlov(radek) # sečti počty za každý řádek
print "Soubor má %d slov." % celkem
```

Pokud byste uvedený kód vyzkoušeli, zjistili byste, že nefunguje. To, co jsme si právě předvedli, je běžná technika návrhu programu. Spočívá v načrtnutí našich představ o tom, jak by kód měl vypadat, ale netrápíme se s tím, aby byl kód absolutně správný. Takovému zápisu se někdy říká *pseudo kód* nebo — při dodržení formálnějšího stylu — zápis v *jazyce pro popis programu* (z anglického *Program Description Language* nebo zkráceně *PDL*).

Až se blíže podíváme na práci se soubory a s řetězci (v další části učebnice), vrátíme se k tomuto příkladu a přepíšeme jej do funkční podoby.

Funkce v JavaScript

V jazyce JavaScript můžeme vytvářet funkce samozřejmě také. Používáme k tomu příkaz function:

```
<script type="text/javascript">
    var i, hodnoty;

    function nasobky(m) {
    var vysledky = new Array();
    for (i = 1; i <= 12; ++i) {
        vysledky[i] = i * m;
        }
        return vysledky;
        }
    // Použití funkce.
    hodnoty = nasobky(8);

    for (i = 1; i <= 12; ++i) {
    document.write(hodnoty[i] + "<br>");
    }
    </script>
```

V tomto případě není uvedená funkce příliš užitečná. Přesto doufám, že jste si všimli podobnosti její základní struktury s definicemi funkcí v jazycích Python a VBScript. Se složitějšími funkcemi v jazyce JavaScript se setkáme v dalších částech učebnice. Zmiňme se například o tom, že v JavaScript se funkce používají jednak jako funkce a jednak k definici objektů. Zní to zmateně a ke zmatkům zde také může docházet.

Než se pustíme do dalších věcí, podívejme se zpět na příklad v JavaScript, který jsme si uvedli v kapitole Konverzace s uživatelem. Právě teď k tomu nastal ten správný čas. JavaScript jsme v něm použili pro čtení vstupu z webového formuláře. Kód vypadal takto:

```
<script type="text/javascript">
    function mujProgram(){
alert("Získali jsme hodnotu " + document.formular.pole.value);
    }
</script>

<form name='formular'>
<p>Zadej hodnotu a potom klikni myší mimo vstupní pole</p>
<input type='text' name='pole' onChange='mujProgram()'>
```

```
</form>
```

Když se na to podíváme s našimi nově nabytými znalostmi, vidíme, že jsme nadefinovali funkci nazvanou mujProgram. Ve formuláři jsme předepsali, že se tato funkce má zavolat, když ze změní obsah vstupního pole. Další vysvětlení si uvedeme v tématu Událostmi řízené programování.

Malé varování

Možnost definovat funkce představuje mocný nástroj, protože nám to umožňuje rozšiřovat schopnosti jazyka. Můžeme dokonce *změnit* chování jazyka tím, že pro předdefinované funkce uvedeme novou definici, s novým významem (některé jazyky to neumožňují). Obvykle se to nepovažuje za příliš dobrý nápad. Každopádně musíme velmi pečlivě kontrolovat dopady podobného činu (za chvíli si ukážeme, jak na to). Předefinováním chování standardních funkcí daného jazyka můžeme velmi ztížit srozumitelnost zdrojových textů. Čtenář očekává, že se taková funkce chová určitým způsobem, ale vy jste toto chování změnili. V praxi se proto změna základního chování zabudovaných funkcí považuje za nevhodnou. Jedna z možností, jak dodržet zásadu *neměnit chování zabudovaných funkcí* a přitom použít jméno funkce pro naše vlastní účely, spočívá ve vložení nové definice do objektu nebo do modulu, které platnost takové funkce definují ve svém vlastním kontextu. K objektové variantě se dostaneme později — v tématu Objektově orientované programování. Nyní se podívejme na možnost vytvoření našich vlastních modulů.

Vytváření našich vlastních modulů

Zatím jsme si ukázali, jak můžeme vytvářet naše vlastní funkce a volat je z jiných částí našeho programu. Vytváření vlastních funkcí nám ušetří hodně psaní a — což je mnohem důležitější — učiní naše programy srozumitelnějšími. Je to dáno tím, že do vytvořených funkcí ukryjeme některé detaily, na které pak při jejich použití nemusíme myslet. (Tento princip obalování složitých úseků programu funkcemi se z docela zřejmých důvodů nazývá *ukrývání informací* [*information hiding*].) Ale jak můžeme tyto funkce používat v jiných programech? Odpověď zní — vytvoříme *modul*.

Moduly v jazyce Python

V jazyce Python není modul ničím zvláštním. Je to prostý textový soubor s příkazy programu v jazyce Python. Obvykle jsou to definice funkcí. Takže, když napíšeme…

```
import sys
```

tak tím interpretu jazyka Python říkáme, že má načíst obsah tohoto modulu, provést inicializaci jeho kódu a zpřístupnit jeho vygenerovaná jména pro použití v *našem* souboru. Je to téměř jako kdybychom okopírovali obsah souboru sys.py do našeho souboru například přes schránku (clipboard) operacemi Kopírovat a Vložit (copy/paste). (No, ve skutečnosti to takhle není, ale koncepčně to můžeme tak chápat.) Překladače některých jazyků (z význačných jmenujme C a C++) kopírují na základě požadavků obsah souborů s moduly do aktuálního programu doslova[1].

Takže si to zrekapitulujme. Modul vznikne vytvořením pythonovského souboru (.py), který obsahuje funkce, které chceme používat v jiných programech. Potom jednoduše provedeme import našeho modulu přesně stejným způsobem, jako to děláme se standardními moduly. Snadné, co? Tak pojďme na to.

Okopírujte si níže uvedenou funkci a uložte ji do svého souboru se jménem nasobky.py. Můžeme to provést prostřednictvím editoru z IDLE, editoru Notepad (Poznámkový blok) nebo jiného editoru, který ukládá prosté textové soubory. Nepoužívejte aplikace typu textový procesor (nepoužívejte tedy například Microsoft Word nebo Microsoft WordPad), protože ty navíc do souboru ukládají všelijaké formátovací značky, kterým by Python nerozuměl.

```
def tisk_tabulky(nasobitel):
print "--- Tisk tabulky násobků číslem %d ---" % nasobitel
for n in range(1, 13):
print "%d x %d = %d" % (n, nasobitel, n * nasobitel)
```

Nyní na příkazový řádek systému Python napište:

```
>>> import nasobky
>>> nasobky.tisk_tabulky(12)
```

No vida! Právě jste vytvořili modul a použili jste ho.

Důležitá poznámka: Pokud jste Python nespustili ze stejného adresáře, ve kterém je uložen váš soubor nasobky.py, pak jej Python možná nenašel a zahlásil chybu. Pokud tomu tak skutečně je, můžete vytvořit proměnnou prostředí nazvanou PYTHONPATH, která obsahuje seznam adresářů, ve kterých se budou hledat moduly (tedy ty, které nejsou dodávány jako standardní spolu se sytémem Python).

Způsob vytváření proměnných prostředí je závislý na platformě. Předpokládám, že příslušné operace buď znáte nebo si je umíte zjistit. Například uživatelé Windows XP mohou použít tlačítko *Start* a najít si informace o proměnných prostředí. Dozví se, jak si je mohou vytvářet. **Poznámka překladatele:**Konkrétně lze po stisku *Start* vybrat položky *Nastavení — Ovládací panely*, poklepat na ikonu *Systém*, přepnout se na záložku *Upřesnit* a stisknout tlačítko *Proměnné prostředí*. Další podrobnosti viz dokumentace.

Moduly ve VBScript a v JavaScript

A jak je na tom VBScript? Tady to bude složitější. V samotném jazyce VBScript a v dalších starších variantách jazyka BASIC koncepce modulu vůbec neexistuje. VBScript místo toho staví na opakovaném použití kódu mezi objekty, který je založen na vytváření objektů. Dostaneme se k tomu později. Prozatím budeme muset potřebný kód okopírovat z dřívějších projektů do aktuálního projektu pomocí funkcí textového editoru.

Poznámka: Větší bratr jazyka VBScript — jazyk Visual Basic — ovšem koncept modulu podporuje. Modul můžeme načíst příkazem menu File|Open Module... v *integrovaném vývojovém prostředí (IDE)*. Pro moduly jazyka Visual Basic existují určitá omezení v tom smyslu, že do něj nemůžeme umístit cokoliv. Do detailů se zde pouštět nebudeme, protože se zde tímto jazykem nezabýváme. (Pokud chcete experimentovat, existuje (nebo alespoň tomu tak bylo) volně dostupná verze jazyka Visual Basic, která je známa jako *Visual Basic 5.0 Control Creation Edition*, čili *VBCCE*. Dala se zdarma stáhnout ze stránek společnosti Microsoft. Pokud máte chuť na pokusy, pak více podrobností naleznete na této stránce.

JavaScript (podobně jako VBScript) nenabízí žádný mechanismus pro používání modulů v podobě souborů s kódem. Ale existují zde určité výjimky a to ve specializovaných prostředích, kde se JavaScript nepoužívá jen uvnitř webových stránek. Jde například o *Windows Script Host* — viz dále.

Windows Script Host

Prozatím jsme se na jazyky VBScript a JavaScript dívali jako na jazyky pro programování uvnitř webového prohlížeče. Ten si však vynucuje určitá omezení, včetně nemožnosti používat moduly. V prostředí Microsoft Windows však existuje i jiný způsob používání VBScript (a JavaScript), konkrétně *Windows Script Host*, zkráceně *WSH*. Jde o technologii firmy Microsoft, která uživatelům umožňuje programovat své osobní počítače stejným způsobem, jakým programátoři v systému DOS používali dávkové soubory (s příponou .bat). WSH poskytuje mechanismy pro čtení ze souborů, ze systémového registru, umožňuje přístup k počítačům v síti, k tiskárnám, atd.

WSH verze 2 navíc umožňuje používat (include) jiný WSH soubor, takže zavádí možnost používání modulů. Ukažme si, jak to funguje. Nejdříve si vytvoříme soubor modulu nazvaný NejakyModul.vbs, který obsahuje:

```
Function OdectiDvojku(N)
OdectiDvojku = N - 2
End Function
```

Teď vytvoříme WSH skript, který nazveme například testModulu.wsf:

```
<?xml version="1.0" encoding="UTF-8" ?>

<job>
<script language="VBScript" src="NejakyModul.vbs" />
<script language="VBScript">
Dim hodnota, vysledek
WScript.Echo "Vložte číslo"
hodnota = WScript.StdIn.ReadLine
vysledek = OdectiDvojku(CInt(hodnota))

WScript.Echo "Výsledek je " &amp; CStr(vysledek)
</script>
</job>
```

Poznámka překladatele 1: V hlavičce XML dokumentu můžeme uvést, v jakém kódování je soubor uložen. Pokud bychom tak neučinili, pak se předpokládá, že je použito kódování UTF-8 nebo UTF-16. V případě, že se v dokumentu vyskytují pouze ASCII znaky, je obsah dokumentu v kódování UTF-8 totožný s obsahem užívajícím kódování ASCII. U českých dokumentů se ale objeví znaky s kódem větším než 127 a binární podoba souboru v kódování UTF-8 se přestane podobat jiným způsobům kódování, které používají 8bitové znaky. V takovém případě je nutné dokument skutečně uložit v kódování UTF-8 (musí to umět editor), protože některé znaky budou uloženy speciálním způsobem. Alternativně můžete v hlavičce XML dokumentu uvést jiné kódování (windows-1250, ISO-8859-2 a podobně). Pak musíte dokument v tomto kódování skutečně uložit. U českých dokumentů se dá obecně říci, že je vhodnější v hlavičce způsob kódování dokumentu uvádět.

Poznámka překladatele 2: Narozdíl od HTML, které pro předpis jazyka v elementu <script ...> předepisuje atribut type="text/*jazyk*", v XML předpisu pro WSH se uvádí forma language="*jazyk*", která se kdysi používala i v HTML.

Teď to můžeme spustit v DOSovém okně:

```
C:\> cscript testModulu.wsf
```

Soubor .wsf na sebe bere podobu XML souboru. Program se nachází mezi značkami <job> a </job>. První značka <script ... /> odkazuje na soubor modulu s názem NejakyModul.vbs. Druhá značka <script ...>obsahuje náš program, který volá funkci OdectiDvojku(), definovanou v modulu NejakyModul.vbs. (První ze značek script je nepárová — její zápis končí />. Veškeré informace jsou zapsány uvnitř této značky. Naopak druhý element script je vymezen dvěmi značkami, které obklopují příslušný obsah.) Soubor s příponou .vbsobsahuje běžný kód v jazyce VBScript bez jakýchkoliv XML či HTML značek.

Povšimněte si příkazu WScript.Echo, ve kterém spojujeme řetězce. Znak ampersand jsme museli zapsat zvláštním způsobem (escape), a sice &, protože příkaz je součástí XML souboru. Povšimněte si také, že ke čtení vstupu od uživatele používáme WScript.Stdin. Vzpomínáte si na poznámku o *stdin* a stdout v tématu Konverzace s uživatelem?

Poznámka překladatele: Pozor! Vytvořený soubor musíme skutečně spustit přes cscript na příkazovém řádku. Určitou činnost sice můžeme pozorovat i při prostém poklepání na ikonu .wsf souboru, ale v tomto případě by došlo ke spuštění v okénkovém režimu. Příkaz WScript.Echo by se projevil v podobě okna se zprávou a tlačítkem OK (Message Box). Nemáme ale vytvořenou vazbu na WScript.Stdin a nepodaří se nám zadat vstupní hodnotu. Po stisku tlačítka OK se objeví okno s chybovým hlášením a program skončí.

Stejnou techniku můžeme použít i pro JavaScript, přesněji řečeno pro verzi jazyka JavaScript firmy Microsoft, která se nazývá *JScript*. Stačí změnit atribut značky script, který určuje jazyk. Při použití WSH můžeme dokonce jazyky míchat. Můžeme importovat moduly v JavaScript a používat je v kódu VBScript a naopak. Jako důkaz si uvedeme odpovídající WSH skript, který přistupuje k VBScriptovému modulu z JavaScriptu:

```
<?xml version="1.0" encoding="UTF-8" ?>

<job>
<script language="VBScript" src="NejakyModul.vbs" />
<script language="JScript">
var hodnota, vysledek;
WScript.Echo("Vložte číslo");
hodnota = WScript.StdIn.ReadLine();
vysledek = OdectiDvojku(parseInt(hodnota));

WScript.Echo("Výsledek je " + vysledek);
</script>
</job>
```

Vidíte, jak moc jsou si obě verze podobné? Většina chytrých věcí se vlastně dělá přes objekty WScript a až na těch pár závorek a středníků jsou skripty v podstatě stejné.

V této učebnici nebudeme WSH používat příliš často. Přesto se k němu občas uchýlíme a sice v situacích, kdy nám více omezující prostředí webového prohlížeče neumožní demonstrovat některé vlastnosti. Tak například v dalším tématu s pomocí WSH ukážeme jak lze v jazycích VBScript a JavaScript pracovat se soubory. Pokud máte zájem, pak o WSH bylo napsáno pár knih. Je mu věnována rozsáhlá sekce webového serveru firmy Microsoft. Naleznete tam i ukázky programů, vývojové nástroje, atd. Vše se nachází zde: http://msdn.microsoft.com/scripting/.

V další části se podíváme na soubory a na zpracování textu. A potom, jak jsme si slíbili, se znovu podíváme na problém počítání slov v souboru. Ve skutečnosti si nakonec pro naše potřeby vytvoříme modul s funkcemi pro zpracování textu.

Zapamatujte si

- Funkce jsou formou modulu.
- Funkce vracejí hodnoty, procedury ne.

- V jazyce Python jsou moduly obvykle tvořeny definicemi funkcí, uloženými v souboru.
- Nové funkce se v jazyce Python uvádějí klíčovým slovem def.
- V jazyce VBScript se pro stejný účel používají klíčová slova Sub nebo Function, v jazyce JavaScript se používá function.

Práce se soubory

O čem si budeme povídat?

- Jak se otvírá soubor.
- Jak se čte z otevřeného souboru a jak se do něj zapisuje.
- Jak se soubor zavírá.
- Adresy — program realizující záznamník na adresy.
- Práce s binárními soubory.

Zpracování souborů začátečníky často přivádí do úzkých, ačkoliv důvody jsou pro mne tak trochu záhadou. Z pohledu programátora se soubory opravdu nijak neliší od souborů, které používáme při práci s textovým editorem nebo s jinou aplikací: musíme je *otevřít*, provedeme nějaké operace s obsahem a zase je *zavřeme*.

Největší rozdíl spočívá v tom, že v programu se k souboru přistupuje *sekvenčně*. To znamená, že čteme od jeho začátku, postupně po jednom řádku. Textový editor často dělá totéž, jenže si obsah celého souboru nejdříve načte do paměti, kde jej upravujete, a v okamžiku ukončení práce se souborem obsah paměti zapíše zpět do souboru a uzavře jej. (Proto se vám může zdát, že textový editor nepoužívá postupné čtení obsahu souboru.)Další rozdíl spočívá v tom, že z programu obvykle soubor otvíráme jen pro čtení nebo jen pro zápis. Při zápisu můžeme vytvořit zcela nový soubor (nebo můžeme přepisovat obsah již existujícího souboru) nebo obsah *připojujeme na konec* (append) existujícího souboru.

Další operace, kterou můžeme při zpracování souboru použít, je skok zpět na začátek.

Soubory — vstup a výstup

Podívejme se na to v praxi. Budeme předpokládat, že existuje soubor zvaný menu.txt a že obsahuje seznam jídel:

```
spam & vajíčka
spam & opékané brambory
spam & spam
```

Poznámka překladatele: Pojem *spam* jsme si již vysvětlovali. Pokud nečtete texty učebnice postupně, naleznete vysvětlení zde. V jedné z epizod série *Monty Python's Flying Circus* se slovo *spam* objevovalo velmi hojně. S překvapivě podobnou frekvencí se v českých titulcích objevovalo slovo *prejt*. Není to totéž. Je to jen podobně stručné, což je pro titulky (a pro dabing) asi důležité.

Nyní napíšeme program, který obsah souboru přečte a zobrazí jej na výstupu — podobně jako to v Unixu dělá příkaz cat nebo v DOSu příkaz type.

```
# Nejdříve soubor otevřeme ke čtení (r jako read).
vstup = file("menu.txt", "r")

# Soubor načteme do seznamu řádků a pak
# každou položku seznamu (řádek) vytiskneme.
for radek in vstup.readlines():
    print radek
# A nyní soubor zase zavřeme.
vstup.close()
```

Poznámka 1: Operace file() vyžaduje dva argumenty. Prvním z nich je jméno souboru. Můžeme je předat prostřednictvím proměnné nebo je můžeme zapsat přímo jako řetězec, jako jsme to učinili zde. (Takovému zápisu řetězce se říká literál. Jde vlastně o přímo zapsanou řetězcovou konstantu.) Druhý argument určuje *režim*. Ten říká, zda soubor otvíráme pro čtení (r jako read) nebo pro zápis (w jako write). Můžeme též určit, zda se jedná o ASCII text nebo o binární data — přidáním 'b' za 'r' nebo za 'w' takto: open(jm_soub, "rb").

Poznámka 2: K otevření souboru jsme použili funkci file(). Starší verze jazyka Python místo ní používaly funkci open(). Parametry obou funkcí jsou naprosto shodné. Používání open() se stále dává přednost, takže v dalších ukázkách budeme obvykle používat open(). Ale pokud se vám zdá používání file() logičtější, používejte file().

Poznámka překladatele: V dokumentaci se dočteme, že open() je alias pro file() a dočteme se zde také:
Záměrem je, aby byla funkci open() i nadále dávána přednost, pokud ji používáme jako *factory function*[1], která vrací nový objekt typu soubor. Zápis file se lépe hodí pro testování typu (například při volání isinstance(f, file)).

Z objektového pohledu (viz dále) ale můžeme v zápisu file() vidět také vytváření objektu voláním jeho konstruktoru. Záleží tedy na tom, jak se na věc chcete dívat. Funkčně jsou obě volání naprosto shodná.

Poznámka 3: Ze souboru jsme četli a uzavírali jsme jej voláním funkcí, před které jsme připsali souborovou proměnnou. Tomuto zápisu se říká *volání metody* a je to naše další setkání s *objektovou orientací*. Teď si s tím nelamte hlavu. Jenom si všimněte, že to má svým způsobem vztah k modulům. O použité souborové proměnné můžete uvažovat, jako kdyby to byla reference na modul, který obsahuje funkce pro práci se soubory a který se jakoby automaticky importuje pokaždé, když vytvoříme souborovou proměnnou.

Poznámka 4: Na konci soubor uzavíráme voláním metody close(). Python (a nejen Python) sice všechny otevřené soubory na konci programu uzavírá, ale mezi dobré zvyky patří předepisování uzavírání souborů přímo. Proč? No, operační systém může odkládat zápis dat do souboru až do doby, kdy je soubor uzavírán (kvůli zvýšení výkonu systému). Pokud náhodou váš program skončí neočekávaným způsobem, riskujete, že vaše drahocenná data nebudou do souboru zapsána vůbec. Takže poučení zní: Jakmile ukončíte zápis do souboru, zavřete jej.

A teď uvažme, jak bychom se mohli vypořádat s dlouhými soubory. V prvé řadě bychom museli soubor číst řádek po řádku. (V jazyce Python bychom místo použití readlines() a cyklu for museli použít readline() a cyklus while.) V takové situaci bychom mohli bychom použít proměnnou poc_radku, kterou bychom zvyšovali při načtení každého řádku a testovali bychom, zda dosáhla hodnoty 25 (počet řádku na obrazovce). Pokud by tato situace nastala, požádáme uživatele, aby stiskl nějaké tlačítko (dejme tomu Enter). Potom bychom poc_radku nastavili na nulu a pokračovali bychom dál. Můžete si to vyzkoušet jako cvičení….

Od verze 2.2 se Python k souborovému objektu umí chovat, jako kdyby to byl seznam řádků. To znamená, že v cyklu for nemusíme používat readlines(), ale jednoduše procházíme všemi řádky souboru. S využitím této vlastnosti můžeme předchozí příklad přepsat takto:

```
# Nejdříve soubor otevřeme ke čtení (r jako read).
vstup = file("menu.txt", "r")
```

```
# Procházíme souborem a tiskneme každou položku (řádek).
for radek in vstup:
    print radek
# A nyní soubor zase zavřeme.
vstup.close()
```

Výhoda tohoto stylu spočívá v tom, že nenarazíme na žádná omezení daná velikostí paměti, jako v případě použití readlines(). Takže se vlastně kombinují výhody cyklu for a výše zmíněného řešení využívajícího while/readline(). Ukázali jsme si skutečně všechno, co pro zpracování souboru potřebujeme. Otevřeme soubor, čteme z něj a manipulujeme s načtenými daty jak potřebujeme. Jakmile skončíme, soubor uzavřeme. V předchozím příkladu jste si mohli všimnout jednoho malého zádrhele. Načtené řádky již na konci obsahují znak konce řádku, takže když je vytisknete příkazem print, který přidá navíc své konce řádků, bude výstup proložen prázdnými řádky. Abychom se tomu vyhnuli, můžeme použít metodu zabudovaného řetězcového typu rstrip(), která z konce řetězce odstraní všechny *bílé znaky* — říká se jim také *netisknutelné znaky*. (Existují i příbuzné metody lstrip() a strip(), které odstraňují bílé znaky zleva, respektive z obou konců řetězce.) Pokud tedy část výše uvedeného příkladu upravíme do podoby...

```
for radek in vstup:
    print radek.rstrip()
```

... mělo by to dopadnout podle očekávání.

Když budeme v jazyce Python chtít zapsat program pro příkaz copy, jednoduše otevřeme nový soubor pro zápis a místo tisku načtených řádků na displej je budeme zapisovat do tohoto souboru:

```
# Vytvoříme obdobu příkazu: COPY MENU.TXT MENU.BAK

# Nejdříve otevřeme soubory pro čtení (r) a pro zápis (w).
vstup = open("menu.txt", "r")
vystup = open("menu.bak", "w")

# Řádky vstupního souboru kopírujeme do nového souboru.
for radek in vstup:
    vystup.write(radek)

print "1 soubor okopírován..."

# Nyní soubory zavřeme.
vstup.close()
vystup.close()
```

Všimli jste si, že jsem na konci použil příkaz print k tomu, aby uživatel poznal, že se něco stalo? Podobná *zpětná vazba pro uživatele* je obvykle vhodná.

Protože jsme v tomto případě zapisovali stejný řádek, který jsme před tím načetli, nenastanou žádné problémy z konci řádků. (Metoda write() nepřidává další konec řádku.) Pokud bychom ale chtěli zapisovat řetězce, které jsme si sami vygenerovali nebo které jsme před tím zbavili pravostranných bílých znaků metodou rstrip(), pak bychom museli na konec výstupního řetězce znak nového řádku přidat. Udělali bychom to takto:

```
vystup.write(radek + '\n')  # \n reprezentuje přechod na nový řádek
```

Podívejme se, jak metodu write() využijeme v našem kopírovacím programu. Abychom soubor jen nekopírovali, přidáme na začátek souboru dnešní datum. Tím z jednoduše upravovatelného textového souboru s nabídkou jídel vygenerujeme denní menu. Stačí, když před vlastním kopírováním obsahu souboru menu.txtpřipíšeme na začátek nového souboru pár řádků:

```
# -*- coding: cp1250 -*-
# Vytvoříme denní menu podle obsahu souboru menu.txt.

import time

# Nejdříve otevřeme soubory pro čtení (r) a pro zápis (w).
vstup = open("menu.txt", "r")
vystup = open("menu.prn", "w")

# Připravíme si řetězec s dnešním datem.
dnes = time.localtime(time.time())
datum = time.strftime(u"%A %d. %B", dnes)

# Přidáme řádek s nadpisem a prázdný řádek.
vystup.write(u"Denní nabídka pro %s\n\n" % datum)

# Řádky vstupního souboru kopírujeme do nového souboru.
for radek in vstup:
    vystup.write(radek)

print u"Menu pro %s bylo vytvořeno..." % datum

# Nyní soubory zavřeme.
vstup.close()
vystup.close()
```

Povšimněte si, že jsme použili modul time k získání aktuálního data a času (time.time()) a k převodu na n-tici souvisejících hodnot (time.localtime()), které jsou zase použity funkcí time.strftime() pro zformátování řetězcové podoby data. Ten je přes další formátovací řetězec vložen do nadpisu. Výsledný soubor pak vypadá nějak takto:

```
Denní nabídka pro Sunday 07. August
```

```
spam & vajíčka
spam & opékané brambory
spam & spam
```

Ačkoliv jsme na konec formátovacího řetězce nadpisu vložili dva znaky '\n', objevil se jen jeden prázdný řádek. Je to tím, že první znak způsobil ukončení řádku s nadpisem a teprve ten druhý způsobil vygenerování prázdného řádku. Správné vytváření a odstraňování znaků pro nový řádek patří při zpracování textových souborů k jedné z těch otravnějších věcí.

Poznámka překladatele: Pokud se vám v nadpisu a ve vypisovaném upozornění objevily anglické názvy dne a měsíce — jak je naznačeno v příkladu výstupu výše —, je to tím, že jste Pythonu neoznámili, jaké místní jazykové zvyklosti (locale) se mají používat. Zkuste za řádek import time přidat následující dva řádky:

```
import locale
locale.setlocale(locale.LC_ALL, 'cz')
```

Výstupní soubor pak nabude o něco lepší české podoby:

```
Denní nabídka pro neděle 07. srpen

spam & vajíčka
spam & opékané brambory
spam & spam
```

Musíme se však smířit, že Python neumí skloňovat česky. I kdyby to nějakým zázrakem uměl, při formátování data jsme stejně nijak neuvedli, v jakém pádu se má výsledek objevit, takže jména dne a měsíce budou uvedena v prvním pádu. Povšimněte si, že jsme na prvním řádku programu (formou speciálního komentáře) uvedli kódování, ve kterém je uložen zdrojový text (zde pro Microsoft Windows, podle potřeby je upravte). Povšimněte si také, že řetězce, které obsahují nebo mohou obsahovat české znaky, uchováváme v kódování Unicode (před úvodní uvozovkou píšeme písmeno **u**).

Konce řádků v různých operačních systémech

Celé téma *konce řádků v textových souborech* patří k temným stránkám nestandardizované implementace v různých operačních systémech. Rozdíly mají své kořeny v dávných dnech úsvitu datových komunikací, v magii ovládání mechanických dálnopisů. Nové řádky se indikují v podstatě třemi různými způsoby:

1. Znak návratu vozíku (Carriage Return — CR; '\r').
2. Znak posunu o řádek (Line Feed — LF; '\n').
3. Dvojice CR/LF ('\r\n').

V různých operačních systémech se používají všechny tři techniky. V MS DOS (a odtud i v MS Windows) se používá třetí způsob. Unix (včetně Linuxu) používá druhou metodu. Apple ve svém původním systému MacOS používá první metodu, ale v současnosti používá metodu druhou. Je to dáno tím, že *MacOS X* je ve skutečnosti variantou systému Unix.

Takže jak se má chudák programátor s takovou rozdílností zakončování řádků vyrovnávat? V mnoha jazycích musí prostě více testovat a provádět jiné akce v závislosti na konkrétním operačním systému. V modernějších jazycích, včetně Pythonu, máme k dispozici prostředky, které nám umožní tento zmatek zvládnout. V případě jazyka Python tato pomoc přichází v podobě modulu os. Ten definuje proměnnou zvanou linesep, která obsahuje posloupnost konce řádku pro daný operační systém. Takže přidávání nových řádků není složité. Pokud je chceme naopak odstranit, použijeme rstrip(), který při odstraňování konce řádku zohlední vlastnosti operačního systému. Takže pokud si chceme při zpracování konců řádků zachovat příčetnost: k odstraňování konců řádků používejte vždy rstrip() a před zápisem do souboru zakončujte řádky přidáním os.linesep.

Stále zde zůstává nepříjemná situace, kdy byl soubor vytvořen v jednom operačním systému a zpracovává se naj jiném, neslučitelném. Bohužel s tím nic moc nenaděláme. Můžeme jen porovnat konce řádků s os.linesep a zjistit, v čem se liší.

Poznámka překladatele: Podle mého názoru mají výše vyjádřená skepse a uvedené rady smysl pouze v situaci, kdy obsah textového souboru načítáme v binárním režimu. V textovém režimu se v systému MS Windows posloupnost konců řádků převádí při čtení na \n automaticky (a to nejen v Pythonu), při zápisu se zase automaticky provádí převod na dvojznakovou posloupnost. Osobně jsem se nikdy nemusel zabývat popisovanou situací a nikdy jsem nemusel přímo používat os.linesep.

V systému Unix se s tímto problémem můžeme setkat jen v případě, kdy načítáme soubor vytvořený v jiném operačním systému. Aniž bych si to nějak ověřoval, předpokládám, že jde o starý problém, který se při práci se soubory v textovém režimu řeší už dávno. Se systémem MacOS ovšem nemám žádné zkušenosti.

Při zpracování souboru byste mohli ještě chtít, aby se načtená data přidávala na konec existujícího souboru. Jednou z možností by bylo otevřít výstupní soubor pro čtení, načíst jeho obsah do seznamu, připojit k seznamu data ze vstupního souboru a nakonec celý seznam zapsat jako novou verzi původního výstupního souboru. Pokud by byly soubory krátké, pak to nezpůsobí žádné problémy. Ale pokud je výstupní soubor velmi velký, třeba větší než 100MB, pak vám prostě při vytváření seznamu řádků dojde paměť. (I kdybyste měli dostatečně velkou paměť, takový postup by byl časově náročný.) Naštěstí můžeme operaci open() určit další režim "a"(jako append), který zajistí připojení dat na konec souboru — do souboru prostě zapisujeme. Je to dokonce ještě vylepšené tím, že pokud soubor neexistuje, bude vytvořen nový soubor — jako kdybyste použili režim "w".

Uveďme si příklad, kdy používáme takzvaný log[2] soubor, do kterého zapisujeme chybová hlášení. Přitom ale nechceme smazat předchozí záznamy, takže nové záznamy připisujeme na konec souboru (error = chyba; msg = message [mesidž] = zpráva):

```
def logError(msg):
err = open("Errors.log", "a")
err.write(msg)
err.close()
```

V reálném světě bychom ovšem rádi nějakým způsobem omezili velikost souboru. Běžně se používá technika, kdy se jméno souboru odvodí z aktuálního data. Takže když se datum změní, vytvoří se automaticky nový soubor. Správce systému pak může snadno najít chyby, které se staly v určitý den. Může snadno rozhodnout, které soubory jsou staré, archivovat je a odstranit v případě, kdy už nebudou potřebné. (Připomeňme si, že aktuální datum můžeme zjistit pomocí funkcí modulu time — stejně jako ve výše uvedeném příkladu generování denní nabídky.)

Oprášený příklad záznamníku s adresami

Pamatujete si na příklad záznamníku s adresami, který jsme poprvé nakousli v tématu Data, datové typy a proměnné a poté vylepšili v kapitole Konverzace s uživatelem? Teď z něj udělám něco opravdu užitečného tím, že obsah záznamníku budeme ukládat do souboru. Při startu programu jej samozřejmě budeme také načítat. Pro tyto účely si napíšeme pár funkcí. V tomto příkladu tedy spojíme několik prvků a dovedností, kterými jsme se zabývali v předešlých tématech.

Náš základní návrh bude vyžadovat funkci, která při startu přečte obsah souboru, a další funkci, která jej při ukončování programu opět do souboru zapíše. Prostřednictvím další funkce uživateli nabídneme možnost volby ze zobrazeného menu. A každou volbu položky z menu budou obsluhovat další funkce. Menu bude uživateli umožňovat:

- Přidání položky do záznamníku s adresami.
- Odstranění položky ze záznamníku.
- Nalezení a zobrazení existující položky.
- Ukončení programu.

Načtení obsahu záznamníku

```
jmeno_souboru = 'adresy.dat'

def nactiObsah(zaznamnik):
    import os
    if os.path.exists(jmeno_souboru):
        soubor = open(jmeno_souboru, 'r')
        for radek in soubor:
            jmeno = radek.rstrip()
            polozka = soubor.next().rstrip()
            zaznamnik[jmeno] = polozka
    else:
        soubor = open(jmeno_souboru, 'w') # vytvoř nový prázdný soubor
    soubor.close()
```

Povšimněte si, že znaky konců řádků odstraňujeme voláním rstrip(). Povšimněte si také, že k získání dalšího řádku souboru uvnitř těla cyklu využíváme operace next(). A všimněte si také toho, že jsme jméno souboru uložili do proměnné, která je definována na úrovni modulu. To znamená, že proměnnou jmeno_souboru můžeme využít jak při načítání, tak při ukládání dat.

Poznámka překladatele k příkladu: Osobně nejsem příznivcem řešení, kdy se v jednom cyklu for načítají dva řádky souboru najednou. První problém spočívá v tom, že by druhý řádek již nemusel být v souboru přítomen (například díky chybě při implementaci zápisu do souboru). V takovém případě vznikne při volání next() výjimka, kterou zde neošetřujeme. To by ale nebylo nejhorší — chyba by se rychle ukázala.

Za závažnější prohřešek považuji to, že *zneužíváme* znalosti vnitřní implementace cyklu for a spoléháme se, že funguje právě tak, jak momentálně funguje. Tuto znalost v kódu zveřejňujeme voláním metody next() a mlčky předpokládáme, že je vše v pořádku. Jinými slovy, ve zdrojovém textu tím vyjadřujeme přímou souvislost fungování cyklu for a metody next(). Úvahy podobného typu se nám mohou v budoucnu vymstít, protože by se například chování cyklu for mohlo změnit. V tomto případě to není pravděpodobné. Berte to jako teoretickou možnost.

V tomto případě bych se pravděpodobně uchýlil k řešení, které by se tomuto problému vyhýbalo. Jeden záznam s celou adresou bych ukládal na jeden řádek. Jeho části bych vhodným způsobem na řádku při zápisu odděloval a při načítání bych je (z jednoho řádku) odpovídajícím způsobem získal.

Za zbytečnou považuji celou větev else. Vytváření souboru ani po logické stránce neodpovídá operaci, při které bychom čekali pouze čtení ze souboru.

Uložení obsahu záznamníku

```
def ulozObsah(zaznamnik):
    soubor = open(jmeno_souboru, 'w')
    for jmeno, polozka in zaznamnik.items():
        soubor.write(jmeno + '\n')
        soubor.write(polozka + '\n')
    soubor.close()
```

Povšimněte si, že při zápisu dat musíme přidávat znak konce řádku ('\n').

Načtení uživatelského vstupu

```
def nactiVolbu(menu):
    print menu
    volba = int(raw_input(u'Zvolte možnost (1-4): '))
    return volba
```

Poznámka překladatele: Python verze 2.4.1 a pravděpodobně i předchozí verze obsahují chybu v implementaci zabudované funkce raw_input(). Pokud použijeme parametr v kódování Unicode a program spustíme pod MS Windows v konzolovém okně, vypíše se výzva v neočekávaném kódování. Je to dáno tím, že MS Windows z historických důvodů používají v konzolovém (DOSovém) okně jiné kódování, než v oknech grafického uživatelského rozhraní. Detailní popis chyby můžete nalézt (anglicky) na SourceForge u projektu Python pod číslem 1099364.

Jeden z vývojářů Pythonu navrhuje dočasné řešení, kdy řetězec s výzvou převedeme do kódování, které používá stdout (souborový objekt pro standardní výstup) takto:

```
import sys
vyzva.encode(sys.stdout.encoding)
```

Nyní máme dvě možnosti. Buď si definujeme vlastní funkci, která co do funkčnosti nahradí raw_input() a přidělíme jí vlastní jméno, nebo upravíme funkčnost původní raw_input() jejím předefinováním. Vybral jsem druhou možnost, protože ji lze využít pro opravu stávajících programů. Vytvoříme novou stejnojmennou funkci, která bude uvnitř volat její zabudovanou variantu (viz předpona __builtins__):

```
def raw_input(vyzva):
    import sys
    return __builtins__.raw_input(vyzva.encode(sys.stdout.encoding))
```

Obecnou nevýhodou tohoto i alternativního přístupu je to, že předefinovaná funkce má omezenou oblast platnosti. Pokud se chcete zeptat na podrobnosti, učiňte tak odkazem na konci této stránky.

Přidání položky

```
def pridejPolozku(zaznamnik):
    jmeno = raw_input(u'Vložte jméno: ')
    polozka = raw_input(u'Vložte ulici, město a telefonní číslo: ')
    zaznamnik[jmeno] = polozka
```

Odstranění položky

```
def odstranPolozku(zaznamnik):
jmeno = raw_input(u'Vložte jméno: ')
del(zaznamnik[jmeno])
```

Nalezení položky

```
def najdiPolozku(zaznamnik):
jmeno = raw_input(u'Vložte jméno: ')
if jmeno in zaznamnik: # v originále je zaznamnik.keys(), ale je to zbytečné
print jmeno, zaznamnik[jmeno]
else:
print u"Lituji. Pro '%s' nebyla nalezena žádná položka." % jmeno
```

Ukončení programu

Pro ukončení programu nebudeme psát nějakou zvláštní funkci. Místo toho budeme volbu na ukončení testovat v podmínce cyklu while. Takže hlavní program bude vypadat takto:

```
def main():
menu = u'''
1) Přidej položku
2) Odstraň položku
3) Najdi položku
4) Uložit a konec
'''
zaznamnik = {}
nactiObsah(zaznamnik)
volba = nactiVolbu(menu)
while volba != 4:
if volba == 1:
pridejPolozku(zaznamnik)
elif volba == 2:
odstranPolozku(zaznamnik)
elif volba == 3:
najdiPolozku(zaznamnik)
else:
print u'Neočekávaná volba. Zkuste to znovu.'
volba = nactiVolbu(menu)
ulozObsah(zaznamnik)
```

Teď už zbývá jen zavolat při spuštění programu funkci main(). Zajistíme to při použití trošky pythonovské magie:

```
if __name__ == '__main__':
main()
```

Tento záhadný úsek kódu nám umožní spouštět pythonovský soubor buď jako modul tím, že ho importujeme, nebo jako program tim, že ho spustíme. Rozdíl při uvedených dvou použitích pythonovského souboru spočívá v tom, že při importování modulu je vnitřní proměnné __name__ (dva znaky podtržení na začátku jména a dva na konci) přiřazeno jméno modulu. Pokud je soubor spuštěn přímo (tj. použit jako samostatný program), je proměnná __name__ nastavena na hodnotu '__main__'. Tajemné, že?

Pokud nyní vložíte všechny kousky kódu do nového textového souboru a uložíte jej jako adresy.py, mělo by to jít spustit z příkazového řádku operačního systému tím, že napíšete:

```
C:\PROJEKTY> python adresy.py
```

Nebo prostě v Průzkumníku (Explorer, MS Windows) poklepete na ikonu. Mělo by se spustit nové DOSové okno a po ukončení programu by mělo zas zmizet.

V Linuxu by to vypadalo podobně:

```
$ python adresy.py
```

Prostudujte si uvedený kód, zkuste v něm najít chyby (nechal jsem tam přinejmenším dvě, ale může jich tam být i více) a zkuste je opravit. Výsledný cca 70 řádkový program je typickým představitelem programů, které byste mohli začít psát pro svou vlastní potřebu. Pár věcí se v něm dá vylepšit — dostaneme se k tomu v další části —, ale i v této podobě jde o rozumně užitečný, malý nástroj.

VBScript a JavaScript

Ani jeden z jazyků VBScript a JavaScript nepodporuje práci se soubory. Jde o rys související s bezpečností. Zajišťuje, že nikdo nebude schopen číst vaše soubory v situaci, kdy jste si nevinně stáhli nějakou webovou stránku. Na druhou stranu se tím však omezuje obecná použitelnost obou jazyků. V části zabývající se znovupoužitelností modulů jsme se ale dozvěděli, že si v tomto směru můžeme pomoci, když použijeme Windows Script Host. WSH nám dává k dispozici objekt FileSystem, který jakémukoliv WSH jazyku umožní čtení souborů. Nejdříve se podíváme na příklad v JavaScript a potom si jej srovnáme s řešením v jazyce VBScript. Ale znovu uvidíme, jako v předchozím případě, že klíčovými prvky řešení budou volání WScript objektů.

Než se dostaneme k podrobnostem, měli bychom se zmínit o *objektovém modelu* FileSystem. Objektovým modelem rozumíme sadu vzájemně souvisejících objektů (tříd), které může programátor přímo využívat. V rámci WSH se objektový model FileSystem skládá z objektů FSO a z řady objektů typu File, včetně objektu TextFile, který budeme používat. Najdeme zde také pomocné objekty. Pro naše účely z nich budeme využívat objekt TextStream. V podstatě budeme postupovat tak, že vytvoříme instanci objektu třídy FSO, tu použijeme pro vytvoření objektů třídy TextFile a z nich vytvoříme objekty TextStream. Do nich budeme zapisovat nebo z nich budeme číst. **The TextStream objects themselves are what we actually read/write from the files.**

Následující kód uložte do souboru nazvaného zkusSoubory.js a spusťe jej pomocí cscript způsobem, který jsme použili v úvodu k WSH (tedy cscript zkusSoubory.js).

Otevření souboru

Abychom ve WSH mohli otevřít soubor, musíme si vytvořit objekt typu FSO a poté jeho prostřednictvím vytvořit objekt TextFile.

```
var jmenoSouboru, fso, inSoubor, outSoubor, radek;
```

```
// Získáme jméno souboru.
fso = new ActiveXObject("Scripting.FileSystemObject");
WScript.Echo("Jak se bude soubor jmenovat? ");
jmenoSouboru = WScript.StdIn.Readline();

// Otevřeme inSoubor pro čtení a outSoubor pro zápis.
inSoubor = fso.OpenTextFile(jmenoSouboru, 1); // režim 1 = čtení
jmenoSouboru = jmenoSouboru + ".BAK"
outSoubor = fso.CreateTextFile(jmenoSouboru);
```

Čtení a zápis

```
// Cyklus přes vstupní soubor, dokud nenarazíme na konec.
while ( ! inSoubor.AtEndOfStream){
radek = inSoubor.ReadLine();
WScript.Echo(radek);
outSoubor.WriteLine(radek);
}
```

Uzavření souborů

```
inSoubor.close();
outSoubor.close();
```

Teď v jazyce VBScript

```
<?xml version="1.0" encoding="UTF-8" ?>

<job>
<script language="VBScript">
Dim fso, inSoubor, outSoubor, inJmenoSouboru, outJmenoSouboru
Set fso = CreateObject("Scripting.FileSystemObject")

WScript.Echo "Zadejte jméno souboru pro zálohu."
inJmenoSouboru = WScript.StdIn.ReadLine
outJmenoSouboru = inJmenoSouboru &amp;amp; ".BAK"

' Otevřeme soubory.
Set inSoubor = fso.OpenTextFile(inJmenoSouboru, 1)
Set outSoubor = fso.CreateTextFile(outJmenoSouboru)

' Čteme soubor a vytváříme záložní kopii.
While not inSoubor.AtEndOfStream
line = inSoubor.ReadLine
outSoubor.WriteLine(line)
Wend

' Uzavřeme oba soubory.
inSoubor.Close
outSoubor.Close

WScript.Echo inJmenoSouboru &amp;amp; " zálohován do " &amp;amp; outJmenoSouboru
</script>
</job>
```

Poznámka překladatele: Soubor uložíme do souboru zkusSoubory.wsf a spustíme:

```
C:\PROJEKTY> cscript zkusSoubory.wsf
```

Práce s netextovými soubory

Zpracování textu patří k jednomu z nejběžnějších programátorských úkolů. Občas ale potřebujeme zpracovávat i binární data. V jazycích VBScript nebo JavaScript se s tímto problémem setkáme velmi zřídka — už jen proto, že nemají přímou podporu práce se soubory —, takže se budeme zabývat jen tím, jak se s tím vypořádáme v jazyce Python.

Otvírání a uzavírání binárních souborů

Klíčový rozdíl mezi textovými a binárními soubory spočívá v tom, že textové soubory jsou složené z *oktetů*(nebo bajtů) s binárními daty, kde každý bajt reprezentuje znak. Konec souboru je označen speciálním bajtem, kterému se anglicky obecně říká *end of file* [end of fajl] nebo *eof* (čili konec souboru). Binární soubor obsahuje libovolná binární data, takže pro identifikaci konce souboru nemůže být použita žádná speciální hodnota. Pro čtení takových souborů se proto musí použít jiný režim. Pokud tedy v Pythonu (nebo v jiném programovacím jazyce) otvíráme binární soubor, musíme jej otevřít v binárním režimu. V opačném případě riskujeme, že soubor bude ukončen v místě prvního výskytu znaku eof, který Python nalezne mezi binárními daty. Binárního otevření souboru v jazyce Python dosáhneme tak, že k parameru režimu přidáme 'b':

```
binarniSoubor = file('binSoubor.bin', 'rb')
```

Jedinou odlišnost od otvírání textového souboru představuje hodnota režimu 'rb'. Písmeno 'b' můžeme přidat i k ostatním režimům: 'wb' pro zápis, 'ab' pro připojování za konec souboru.

Uzavírání binárního souboru se provádí stejně, jako u textového souboru. Jednoduše zavoláme metodu otevřeného souborového objektu close():

```
binarniSoubor.close()
```

Protože soubor byl otevřen v binárním režimu, nemusíme Pythonu poskytovat nějakou další zvláštní informaci — Python ví, jak má soubor korektně uzavřít.

Poznámka překladatele: Můj osobní pohled na rozdíl v binárních a textových souborech je trochu jiný. Hlavní odlišnost spatřuji v tom, zda existuje obecně přijímaná interpretace uložených dat, či nikoliv. Jinými slovy se dá říci, že odlišnost spočívá v tom, zda existuje obecně přijímaný *abstraktní pohled* na uložená data. Obecně se přijímá to, že v textovém souboru jsou uloženy znaky a že se textový soubor člení na řádky. Jde o technický pohled. Z jazykového hlediska bychom

mohli text členit na odstavce, věty, slova, znaky uvnitř slov a znaky, které nejsou součástí slov. Zůstaňme u technického pohledu.

Textový soubor lze v současnosti považovat za pouhou variantu binárních souborů. Dohodnutá interpretace si vynucuje, aby obsahoval jen určitou podmnožinu všech možných binárních kombinací. Pokud víme, že se na soubor máme dívat jako na textový, jsme například schopni vytisknout nebo jinak zobrazit jeho lidsky čitelnou podobu.

Pokud bychom chtěli být přísní, měl by textový soubor obsahovat pouze písmena, další tisknutelné znaky, mezeru a sekvence pro oddělování řádků (CR a LF — viz poznámka výše). Oddělovače řádků patří mezi takzvané *řídicí znaky*. Tento pojem pochází z doby dálnopisů — tyto znaky řídily činnost dálnopisu, pokud zrovna neměl něco tisknout. Historické a technické souvislosti způsobily, že se za součást textových souborů považují i další řídicí znaky, jako je znak tabulační ('\t'), znak zpětného posuvu ('\b' z anglického back), znak*alarm* ('\a', dálnopis cinknul), a další. Patří sem i znak, který ukončuje konec souboru, ale v současných textových souborech se nepovažuje za povinný.

Binární soubor žádnou dohodnutou interpretaci nemá. To znamená, že bez dalších znalostí nevíme, co data znamenají, jak s nimi máme zacházet, kolik bajtů nebo bitů tvoří jeden informační celek, zda soubor obsahuje posloupnost informačních jednotek o stejné velikosti, či nikoliv, atd.

Stupně abstrakce se tedy při práci se soubory v binárním a v textovém režimu liší. Práci se soubory v binárním režimu musíme z hlediska zpracování systémem považovat za práci na nižší úrovni abstrakce. S vyšší úrovní abstrakce textových souborů souvisí existence některých operací, jako je například načtení jednoho řádku.

Reprezentace dat a jejich ukládání

Než si řekneme, jak můžeme přistupovat k datům v binárním souboru, měli bychom se dozvědět něco o tom, jakým způsobem jsou data reprezentována a ukládána v počítači. Veškerá data jsou ukládána jako posloupnosti binárních číslic (*bi*nary digit), bitů. Bity se sdružují do skupin po 8 nebo po 16 a nazývají se bajty (bytes), respektive slova (words). (Skupiny po 4 bitech se někdy nazývají nibble.) Bajt může obsahovat jeden z 256 různých vzorků, kterým jsou přiřazeny hodnoty 0–255.

Veškeré informace, se kterými v našich programech manipulujeme — řetězce, čísla a další —, musí být převedeny na posloupnosti bajtů. To znamená, že pro znaky, které jsme použili v řetězci, musíme vyhradit odpovídající vzorek bajtů. V minulosti se používalo několik způsobů *kódování*, ale nejpoužívanějším se stalo takzvané *ASCII* kódování (**A**merican **S**tandard **C**oding for **I**nformation **I**nterchange). Čisté ASCII je naneštěstí definováno jen pro 128 hodnot, což nestačí pro použití v neanglických jazycích. Později byl navržen nový kódovací standard, známý jako *Unicode*. Ten pro ukládání datové reprezentace znaků používá místo bajtů slova, což umožňuje kódovat přes 65000 znaků. Pokud použijeme kódovací formát UTF-8, pak původní soubory v ASCII kódování představují korektní reprezentaci Unicode textu. Python standardně podporuje kódování ASCII. Pokud před zápis řetězce uvedeme písmeno **u**, bude se řetězec považovat za řetězec v kódování Unicode.

Poznámka překladatele: O věcech souvisejících se standardem Unicode se můžeme podrobněji dočíst na stránkách http://www.unicode.org/. Technický úvod ke standardu Unicode naleznete na stránce http://www.unicode.org/standard/principles.html (principy, formáty).

Výše uvedená informace o počtu bajtů na znak a o počtu kódovaných znaků je nepřesná. Standard Unicode verze 4.0 definuje kódy pro 96447 znaků. Unicode verze 4.1.0 přidává dalších 1273 znaků. Standard definuje jednoznačné kódy pro každý znak. Pro ukládání do souboru se používají kódovací formáty UTF-8, UTF-16 a UTF-32. Určují způsob, jakým se jednoznačné číslo znaku převede do binární podoby pro uložení v souboru. Pokud použijeme formát UTF-32, pak je každý znak kódován na 4 bajtech. Pokud použijeme UTF-16, pak je většina znaků kódována na 2 bajtech a některé na 4 bajtech. Pokud použijeme kódování UTF-8, pak jsou ASCII znaky kódovány na jednom bajtu, ale některé znaky vyžadují až 4 bajty. Pokud chceme pracovat s Unicode řetězci v neanglických jazycích, musíme na začátku zdrojového textu pythonovského programu uvést speciální komentář, který říká, v jakém kódování je zdrojový text uložen. Při překladu zápisu řetězce pak může dojít ke korektnímu převodu zápisu do Unicode.

Do binárního kódování musíme převádět i čísla. Pro malá celá čísla stačí přímo využít hodnoty jednoho bajtu. Ale pro čísla větší, než 255 (nebo pro záporná čísla, nebo pro racionální čísla) musíme učinit něco navíc. Během doby se objevila celá řada standardů pro kódování numerických dat. Využívá je většina programovacích jazyků a operačních systémů. Řadu způsobů kódování čísel s plovoucí řádovou čárkou vydal například americký *Institute of Electrical and Electronic Engineering* (IEEE).

Pointa spočívá v tom, že při čtení binárního souboru musíme v našem programu zajistit převod surových bitových vzorků na hodnotu správného *datového typu*. Sérii bajtů, kterou jsme původně zapsali jako řetězec znaků, můžeme klidně načítat jako sérii čísel v plovoucí řádové čárce. Původní význam se tím samozřejmě ztratí. Chci jen naznačit, že stejný bitový vzorek může reprezentovat oba případy. Pokud tedy načítáme binární data, je velmi důležité, abychom je převedli na správný datový typ.

Modul struct

Pro kódování a dekódování binárních dat můžeme v Pythonu využít modulu struct (zkratka pro *structure*, tedy struktura). Tento modul pracuje podobně, jako když jsme používali formátovací řetězec pro tisk dat různého typu. Zadáváme řetězec, který reprezentuje typ načítaných dat, a ten je použit pro proud bajtů, který se pokoušíme interpretovat.

Modul struct můžeme použít také pro převod dat na proud bajtů určených k zápisu do binárního souboru (nebo dokonce do komunikační linky).

Modul definuje řadu kódů pro převod formátů, ale my zde použijeme jen kódy pro celá čísla a pro řetězce. (Ostatní kódy si můžete vyhledat v dokumentaci k modulu struct, který je součástí distribuce Pythonu. Kódy pro celé číslo a řetězec jsou i respektive s. Formátovací řetězec modulu struct se skládá z posloupnosti kódů, kterým jsou předřazena čísla, určující kolik prvků příslušného typu chceme získat. Tak například zápis 4sznamená, že chceme řetězec o délce 4 znaky.

Poznámka překladatele: Číslo před řetězcovou značkou se chápe jinak, než čísla před značkami pro jiné typy. U řetězce udává délku získávaného řetězce, u značek ostatních typů jde skutečně o počet hodnot daného typu. Tak například značka 10s vyjadřuje *jeden* řetězec o délce 10 znaků, zatímco značka 10c vyjadřuje *deset*jednoznakových řetězců (Python nezná typ *znak*).

Dejme tomu, že bychom chtěli detaily adresy ve výše zmíněném záznamníku adres zapisovat jako binární data, kde by číslo domu bylo uloženo jako celé číslo a zbytek by byl uložen jako řetězec. (Z praktického hlediska to zase není tak dobrý nápad, protože *čísla domů* někdy obsahují i písmena.) Formátovací řetězec by pak vypadal nějak takto:

```
'i34s'  # Předpokládáme, že na adresu je vyhrazeno 34 znaků.
```

Pokud bychom potřebovali pracovat s různou délkou adresy, mohli bychom si napsat funkci, která vytvoří její binární podobu takto:

```
def formatujAdresu(adresa):
# split rozdělí řetězec na seznam 'slov'.
```

```
    slova = adresa.split()
    cislo = int(slova[0])
    zbytek = ' '.join(slova[1:])
    format = "i%ds" % len(zbytek)  # vytvoř formátovací řetězec
    return struct.pack(format, cislo, zbytek)
```

Takže adresu jsme rozsekali na kousky metodou split() zabudovaného typu řetězec. První slovo jsme převedli na číslo a ostatní slova jsme opět spojili mezerami do jednoho řetězce. Jeho délku potřebujeme pro vygenerování formátovacího řetězce pro metodu modulu struct.

Funkce formatujAdresu() vrací posloupnost bajtů, které zachycují binární vyjádření zadané adresy. Když už tedy máme potřebná binární data, podívejme se, jak je můžeme do binárního souboru zapsat a zase je zpět přečíst.

Čtení a zápis s využitím modulu struct

Vytvoříme si binární soubor, který bude obsahovat jediný řádek adresy převedený do binární podoby výše nadefinovanou funkcí formatujAdresu(). Soubor musíme otevřít pro zápis v binárním režimu ('wb'), zakódujeme data, zapíšeme je do souboru a ten následně uzavřeme. Vyzkoušejme si to:

```
import struct

f = file('adresa.bin','wb')
data = "10 Ulice, Město, 0171 234 8765"
bindata = formatujAdresu(data)
f.write(bindata)
f.close()
```

Otevřením souboru adresa.bin v Poznámkovém bloku (notepad, případně v jiném editoru) si můžete ověřit, že data byla skutečně zapsána v binárním tvaru. Znaky adresy sice budou čitelné, ale neuvidíme zde žádné číslo 10.

Abychom adresu ze souboru opět přečetli, musíme jej otevřít v režimu 'rb', načíst data jako posloupnost bajtů, uzavřít soubor a nakonec data rozbalit metodou unpack() modulu struct. K tomu opět potřebujeme formátovací řetězec. Otázka zní, jak by měl vypadat? V našem případě víme, že musí být stejný jako ten, který jsme si připravili uvnitř funkce formatujAdresu() — konkrétně iNs, kde N musíme nahradit konkrétním číslem. Ale jak hodnotu N zjistíme?

V modulu struct najdeme také pomocné funkce, které vracejí velikost každého datového typu. Když si spustíme Python v interaktivním režimu, pak po pár pokusech zjistíme, kolik bajtů zabírají hodnoty různého datového typu:

```
>>> import struct
>>> print struct.calcsize('i')
4
>>> print struct.calcsize('s')
1
```

Takže teď už víme, že číslo zabere 4 bajty a každý znak řetězce zabere jeden bajt. To znamená, že N spočítáme jako délku dat mínus 4. Vyzkoušejme si načíst obsah našeho souboru:

```
import struct

f = file('adresa.bin','rb')
data = f.read()
f.close()

format = "i%ds" % (len(data) - 4)
cislo, zbytek = struct.unpack(format, data)
adresa = str(cislo) + ' ' + zbytek
print adresa
```

Co se týká binárních souborů je to vše, k čemu jsem se chtěl vyjádřit. Jistě jste si všimli, že používání binárních dat vede ke komplikacím. Pokud k tomu nemáte velmi dobrý důvod, pak uvedený přístup rozhodně nedoporučuji. Pokud ovšem skutečně potřebujete číst binární soubor, je to možné. V takovém případě ovšem musíte vědět, co data reprezentují.

Poznámka překladatele: V uvedeném příkladu předpokládáme, že pracujeme s řetězci, kde je každý znak uložen na jednom bajtu. Pokud bychom navíc potřebovali pracovat s Unicode řetězci, pak se při použitém kódování UTF-8 může počet bajtů pro uložení znaku měnit. Novější verze jazyka Python navíc podporují i celočíselný typ, kde hodnota tohoto typu může být větší, než jakou můžeme zachytit na 4 bajtech. Pořadí ukládaných bajtů čísla se navíc řídí pravidly konkrétního výpočetního prostředí (little/big endian). Pokud tedy chceme zařídit přenositelnost takto vygenerovaných binárních souborů do na jiné systémy, musíme si pomoci explicitním uvedením dalších formátovacích značek, které předepíší konkrétní pořadí ukládání bajtů. Věci mohou být mnohem komplikovanější, než se na první pohled zdá.

Zapamatujte si

- Před použitím souborů je musíte otevřít.
- Ze souborů můžeme obvykle jen číst nebo do nich můžeme jen zapisovat, ale ne obojí současně.
- Funkce readlines() jazyka Python přečte všechny řádky souboru najednou, zatímco funkce readline()přečte jen jeden řádek. Může nám to pomoci šetřit pamětí.
- Po použití soubor uzavřete.
- Při práci se soubory v binárním režimu musíme navíc uvést příznak 'b'.

Zpracování textu

O čem si budeme povídat?

- Jak rozdělit řádky textu na skupiny znaků.
- Jak vyhledávat textové řetězce uvnitř jiných řetězců.
- Jak nahradit text uvnitř řetězce.
- Jak měnit velikost písmen (malá za velká a naopak).

Zpracování textu patří mezi nejběžnější programátorské činnosti. Díky tomu nám většina programovacích jazyků nabízí řadu specifických nástrojů, které mají zpracování textu usnadnit. V této části se podíváme na některé z nich a na to, jak je můžeme využít při realizaci typických programátorských úloh.

Mezi nejběžnější akce při práci s textem patří:

- Rozdělování řádků textu na skupiny znaků,

- hledání podřetězců v řetězcích,
- náhrada textu uvnitř řetězce,
- změna velikosti písmen (malá na velká a naopak).

Podíváme se, jak se uvedené úkoly řeší v jazyce Python. Poté se stručně seznámíme s tím, jaké možnosti zpracování textu poskytují jazyky VBScript a JavaScript.

Od verze 2.3 Python při zpracování textu postupuje trošku nejednoznačně. Příčina spočívá v tom, že starší verze jazyka Python prováděly veškeré manipulace prostřednictvím modulu, který byl napěchovaný funkcemi a užitečnými konstantami. Python verze 2.0 zavedl pro typ řetězec metody, které nahrazují funkce ve zmíněném modulu, ale konstanty byly dostupné stále jen prostřednictvím modulu. Tato situace trvala až do verze 2.3. Další vývoj směřuje k tomu, aby potřeba používat starý modul string byla zcela odstraněna. V tomto tématu se zaměříme výlučně na nový, objektově orientovaný přístup k manipulaci s řetězci. Pokud si chcete vyzkoušet práci s původním modulem, hledejte potřebné informace v jeho dokumentaci.

Rozdělování řetězců

Nejdříve se budeme zabývat tím, jak můžeme řetězec rozdělit na části, z kterých se skládá. Při zpracování textových souborů se s takovým požadavkem setkáváme často. Obsah souboru se obvykle snažíme číst po řádcích, ale požadovaná data mohou být uložena v určitých částech řádků. Jako příklad si můžeme uvést náš záznamník s adresami. Místo tisku celé adresy bychom mohli chtít přistupovat například jen k částem adresy.

V Pythonu pro tento účel použijeme metodu split() (rozdělit na části, rozštípnout) následujícím způsobem:

```
>>> retezec = 'Toto je (kratky) retezec.'
>>> print retezec.split()
['Toto', 'je', '(kratky)', 'retezec.']
```

Povšimněte si, že se nám vrací seznam, který obsahuje slova z řetězcové proměnné retezec. Všechny mezery byly odstraněny. Předdefinovaným oddělovačem metody split() jsou totiž *bílé znaky* (tj. tabulátory, znaky přechodu na nový řádek a mezery). Zkusme to znovu, ale tentokrát si jako oddělovač vybereme otvírací závorku:

```
>>> print retezec.split('(')
['Toto je ', 'kratky) retezec.']
```

Vidíte ten rozdíl? Seznam obsahuje tentokrát jen dva prvky a otvírací závorka na začátku 'kratky)' byla odstraněna. Důležitou vlastností metody split() je to, že odstraňuje oddělovací znaky. Většinou takové chování požadujeme, ale občas bychom si přáli, aby tomu tak nebylo.

K dispozici máme i metodu join(), která přebírá seznam (nebo také jinou podobu posloupnosti) řetězců a spojuje je dohromady. Jednou z matoucích vlastností metody join() je to, že řetězec, jehož metodu join()voláme, je použit v roli spojovacích znaků. Následující příklad ukazuje, co mám na mysli:

```
>>> lst = ['Tohle', 'je', 'seznam', 'slov.']
>>> print '-+-'.join(lst)
Tohle-+-je-+-seznam-+-slov.
>>> print ' '.join(lst)
Tohle je seznam slov.
```

Když o tom začnete přemýšlet, dává to smysl. Ale na první pohled to vypadá divně.

Poznámka překladatele: Na první pohled by se zdálo přirozenější, kdyby metoda join() byla metodou typu *seznam* a jako parametr by přebírala řetězec, který se použije pro spojení prvků seznamu:

```
>>> lst.join('-+-')    # Tohle NEFUNGUJE!
```

Ale argumentem metody join() nemusí být přímo *seznam* řetězců. Může to být libovolný iterovatelný objekt, tj. objekt který podporuje průchod jednotlivými řetězcovými položkami. (Typicky to bývá kontejner, ale může to být například i generátor s omezeným počtem generovaných řetězcových hodnot.) Pokud si to uvědomíme, nevypadá už rozhodnutí o příslušnosti metody join() k řetězcovému typu tak nezvykle. V opačném případě by metoda join() musela být implementována pro všechny typy kontejnerů, jejichž obsah bychom chtěli spojovat. Následující příklad uvádí převod seznamu na n-tici a na množinu a spojení jejich obsahu mezerou:

```
>>> lst = ['Tohle', 'je', 'seznam', 'slov.']
>>> t = tuple(lst)
>>> se = set(lst)
>>> print ' '.join(t)
Tohle je seznam slov.
>>> print ' '.join(se)
je seznam Tohle slov.
```

Povšimněte si, že v případě množiny není dodrženo pořadí, ve kterém se prvky množiny vkládaly (z množinového hlediska není pořadí položek důležité). Z prvků množiny ovšem můžeme nějakou funkcí vytvořit uspořádanou posloupnost:

```
>>> print ' '.join(sorted(se))
Tohle je seznam slov.
```

Zdánlivě správné pořadí slov je dáno pouze náhodou, protože uvedená slova ve větě náhodou odpovídají svému lexikografickému pořadí. Dokažme si to na jiném příkladě:

```
>>> lst = ['tohle', 'je', 'neusporadany', 'salat', 'z', 'ceskych',
...        'slov', 'bez', 'diakritickych', 'znamenek']
>>> se = set(lst)
>>> print ' '.join(se)
ceskych slov neusporadany diakritickych znamenek bez salat je z tohle
>>> print ' '.join(sorted(se))
bez ceskych diakritickych je neusporadany salat slov tohle z znamenek
```

Počítání slov

Nyní se znovu podívejme na program pro počítání slov, o kterém jsem se zmínil v předchozí podkapitole o funkcích.

Připomeňme si, že *pseudo kód* vypadal takto:

```
def pocetSlov(s):
seznam = split(s)  # seznam, kde prvkem je vždy slovo
return len(seznam) # vrátíme počet prvků seznamu

for radek in soubor:
```

```
celkem = celkem + pocetSlov(radek) # sečti počty za každý řádek
print "Soubor má %d slov." % celkem
```

Nyní již víme, jak lze načítat řádky souboru. Podívejme se blíže na tělo funkce pocetSlov(). Nejdříve chceme z daného řádku vytvořit seznam slov. Stačí, když na řádek aplikujeme standardní metodu split(). Nahlédnutím do dokumentace zjistíme, že zabudovaná funkce len() vrací pro seznam počet v něm umístěných prvků. V našem případě to bude počet slov v řetězci, což je přesně to, co potřebujeme.

Takže konečná podoba zdrojového kódu vypadá takto:

```
# -*- coding: cp1250 -*-
import string
def pocetSlov(s):
seznam = s.split() # split() je metodou řetězcového objektu s
return len(seznam) # vrátíme počet prvků seznamu

vstup = open('menu.txt', 'r')
celkem = 0 # vytvoříme proměnnou a nastavíme jí počáteční hodnotu nula

for radek in vstup:
celkem = celkem + pocetSlov(radek) # sečti počty za každý řádek
print u'Soubor má %d slov.' % celkem

vstup.close()
```

Tento program není tak docela správný, protože například započítá samostatně stojící znak '&' (ampersand) jako slovo (ačkoliv si vlastně můžete myslet, že je to správné). Program navíc zpracovává jediný soubor (menu.txt). Ale není příliš obtížné upravit jej tak, aby jméno souboru četl z příkazového řádku (argv[1]) nebo prostřednictvím raw_input(), jak jsme si ukázali v podkapitole Konverzace s uživatelem. Řešení ponechávám za domácí úkol.

Vyhledávání textu

Další běžnou operací, na kterou se teď podíváme, je vyhledávání podřetězce v delším řetězci. V Pythonu pro ni opět nalezneme podporu v podobě metody zabudovaného řetězcového typu — tentokrát se jmenuje find(). Její základní způsob použití je velmi jednoduchý. Předáte jí vyhledávaný podřetězec, a pokud jej Python v hlavním řetězci najde, vrátí index prvního znaku, na kterém podřetězec začíná. Pokud podřetězec nalezen není, vrací se hodnota -1:

```
>>> retezec = 'Toto je dlouhy retezec, ve kterem se nachazi podretezec.'
>>> print retezec.find('dlouhy')
8
>>> print retezec.find('moc')
-1
>>> print retezec.find('retezec')
15
```

První dva příklady použití jsou přímočaré. První z nich vrací index začátku podřetězce 'dlouhy'. Druhý příklad použití vrací hodnotu -1, protože podřetězec 'moc' se v řetězci nevyskytuje. U třetího příkladu použití narazíme na jeden detail — vyhledal se *první* výskyt zadaného podřetězce. Ale co vlastně můžeme dělat v případech, kdy se hledaný vzorek vyskytuje v původním řetězci více než jednou?

Jedna z možností spočívá v použití indexu prvního výskytu a rozseknutí původního řetězce na dvě části. Poté můžeme hledat znovu. Takto pokračujeme až do doby, kdy se nám vrátí výsledek -1:

```
# -*- coding: cp1250 -*-
retezec = u"'Haf, ňaf', říká pes. Kolik 'af' je v řetězci?"
s = retezec           # Na začátku se odkazujeme na stejný řetězec.
pocet = 0
index = s.find('af')
while index != -1:
pocet += 1
s = s[index + 1:]  # Budeme pracovat s druhou částí řetězce (slicing).
index = s.find('af')
print u"V řetězci \"%s\" jsme našli %d výskyty vzorku 'af'." % (retezec, pocet)
```

V příkladu jsme pouze výskyty vzorku pouze počítali. Stejně dobře bychom ale mohli nalezené indexy sbírat do seznamu, který bychom využili při dalším zpracování.

Pokud použijeme nepovinné parametry metody find(), můžeme proces urychlit. Těmito nepovinnými parametry je určení počáteční a koncové pozice v původním řetězci:

```
# -*- coding: cp1250 -*-
retezec = u"'Haf, ňaf', říká pes. Kolik 'af' je v řetězci?"
pocet = 0
index = retezec.find('af')
while index != -1:
pocet += 1
start = index + 1  # Připravíme nový začátek.
index = retezec.find('af', start)
print u"V řetězci \"%s\" jsme našli %d výskyty vzorku 'af'." % (retezec, pocet)
```

U tohoto řešení nemusíme pokaždé vytvářet nový řetězec, což může být v případě dlouhých řetězců časově náročné. Pokud chceme vyhledávat výskyt podřetězce jen v několika prvních znacích (a další případné výskyty nás nezajímají), můžeme určit počáteční a koncové pozice prohledávané části řetězce takto:

```
# -*- coding: cp1250 -*-
retezec = u"'Haf, ňaf', říká pes. Kolik 'af' je v řetězci?"
print retezec.find('af', 0, 20)
```

V souvislosti s vyhledáváním Python nabízí navíc několik pěkných, speciálních metod, které se nám hodí v nejčastějších případech — jde zejména o metody startswith() a endswith() (dalo by se přeložit jako *začíná tímhle* a *končí tímhle*). Samotná jména napovídají, so metody asi dělají. Vracejí hodnoty True nebo False v závislosti na to, zda původní řetězec (tj. objekt jehož metodu voláme) začíná nebo končí vyhledávaným podřetězcem. Příklad:

```
>>> print 'Python jede!'.startswith('Perl')
False
>>> print 'Python jede!'.startswith('Python')
True
>>> print 'Python jede!'.endswith('je nanic!')
False
>>> print 'Python jede!'.endswith('de!')
True
```

Povšimněte si, že výsledek je boolovského typu. Povšimněte si také, že vyhledávací řetězec nemusí mít podobu celého slova, stačí prostě podřetězec. Pokud chcete testovat na výskyt podřetězce jen v učité části řetězce, můžeme upřesnit počáteční a koncovou pozici části, která se má prohledávat — stejně jako u metody find(). V praxi se ale tenhle rys moc nepoužívá.

Na závěr si uveďme, že pro prosté otestování, zda se podřetězec nachází kdekoliv v řetězci můžeme použít pythonovský operátor in takto:

```
>>> if 'foo' in 'foobar': print 'True'
True
>>> if 'baz' in 'foobar': print 'True'
>>> if 'bar' in 'foobar': print 'True'
True
```

Poznámka překladatele: Podle mého názoru uvedený příklad zbytečně kombinuje přímo nesouvisející věci a svádí tak začátečníky na scestí.

1. Pro demonstraci není vůbec nutné používat konstrukci if.
2. Tisknout True, pokud je výsledek výrazu True navozuje představu, že jinak to není možné ukázat.

Za didaktičtější pokládám přímé zobrazení výsledku výrazu:

```
>>> print 'foo' in 'foobar'
True
>>> print 'baz' in 'foobar'
False
>>> print 'bar' in 'foobar'
True
```

Liší se to sice tím, že se v druhém případě tiskne False, zatímco v původním příkladu se netiskne nic, ale asi to není na závadu. V interaktivním režimu můžeme dokonce vynechat příkaz print:

```
>>> 'foo' in 'foobar'
True
>>> 'baz' in 'foobar'
False
>>> 'bar' in 'foobar'
True
```

K vyhledávání je to zatím vše. Teď se podívejme, jak můžeme provádět náhrady textu.

Náhrada textu

Když nalezneme hledaný podřetězec, často jej chceme změnit na něco jiného. Řešení se nám nabízí v podobě metody replace(), kterou nalezneme mezi metodami pro pythonovský typ řetězec. Vyžaduje zadání dvou argumentů: vyhledávaný podřetězec a řetězec, kterým bude nahrazen. Metoda vrací nový řetězec, který je výsledkem náhrady.

```
>>> retezec = 'Jedna, dvě / Honza jde / nese pytel s brouky'
>>> print retezec.replace('s brouky', 'mouky')
Jedna, dvě / Honza jde / nese pytel mouky
```

Zajímavé je, že metoda replace() standardně nahrazuje *všechny výskyty* vyhledaného řetězce a ne jen první výskyt, narozdíl od metody find(). (Není to zase tak překvapující, pokud si uvědomíme, že find() by při vyhledávání všech výskytů musela vracet seznam pozic. Bráno z opačného konce, u metody replace() — pokud by nahrazovala standardně jen první výskyt — bychom zase nějakým způsobem museli umět předepsat náhradu všech výskytů.) Maximální počet náhrad můžeme omezit zadáním nepovinného argumentu count (tj. počet):

```
>>> retezec = 'Haf, ňaf, blaf, řekl pejsek.'
>>> print retezec.replace('af', 'afiky')
Hafiky, ňafiky, blafiky, řekl pejsek.
>>> print retezec.replace('af', 'afiky', 1) # jen první výskyt
Hafiky, ňaf, blaf, řekl pejsek.
```

Mnohem důmyslnější operace vyhledávání a náhrad můžeme provádět pomocí nástroje zvaného *regulární výrazy*. Ale práce regulárními výrazy je mnohem složitější, a proto si zaslouží vlastní téma v části *Témata pro pokročilé*.

Změna velikosti písmen

Poslední věc, kterou se v této části budeme zabývat, je změna malých písmen na velá a naopak. Není to tak úplně běžná operace, ale Python nám pro tento účel nabízí pár metod:

```
>>> print u'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.lower()
smíšený případ: ěščřžýáíéúůó ěščřžýáíéúů
>>> print u'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.upper()
SMÍŠENÝ PŘÍPAD: ĚŠČŘŽÝÁÍÉÚŮÓ ĚŠČŘŽÝÁÍÉÚŮ
>>> print u'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.swapcase()
smíŠENý přÍPAD: ĚŠČŘŽÝÁÍÉÚŮÓ ěščřžýáíéúů
>>> print u'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.capitalize()
Smíšený případ: ěščřžýáíéúůó ěščřžýáíéúů
>>> print u'TEST'.isupper()
True
>>> print u'TEST'.islower()
False
```

Povšimněte si, že metoda capitalize() (z anglického *capital letter*, čili velké písmeno) provádí zvětšení prvního písmene pro řetězec jako celek, nikoliv pro každé slovo zvlášť. Všimněte si také dvou funcí pro otestování řetězce na určitou

vlastnost (neboli *predikátů*). K dalším užitečným testům patří isalpha() a isspace(). Poslední zmíněná funkce kontroluje, zda se v řetězci vyskytují jen *bílé znaky*, tedy nejen mezery (tj. také tabulátory, konce řádků...).
V dalších částech kurzu budeme mnohé z uvedených řetězcových metod používat. Zejména Případová studie gramatického počítadla jich používá několik najednou.

Poznámka překladatele: Povšimněte si, že ve výše uvedené ukázce řetězcových metod jsou všechny literály (řetězcové konstanty) zapsány jako Unicode řetězce — na začátku, před prvním apostrofem mají uvedeno písmeno **u**. V originálním anglickém textu tomu tak není. Ale čeština používá znaky, které nepatří do kódu ASCII, a proto význam kódu znaku závisí na tom, jaké kódování používáme.
Pythonovský string, který není uložen v kódování Unicode, s sebou nenese žádnou informaci o použitém kódování. Python takový řetězec chápe jako posloupnost bajtů. Nemá odkud vzít informaci o tom, že velké písmeno k znaku s kódem 249 má kód 217. Například v kódování windows-1250 jde o znaky ůŮ. V jiných kódováních ovšem uvedené kódy mohou patřit zcela jiným znakům, nemusí vůbec tvořit pár malé/velké písmeno a dokonce vůbec nemusí patřit mezi písmena.
ASCII znaky — tj. ty, které mají kód v rozsahu 0 až 127 — představují výjimku v tom smyslu, že mají stejný kód ve všech odvozených kódováních (kódy ASCII znaků jsou zachovány i v Unicode). Metody pro změnu velikosti písmen tedy mohou bez problémů nad těmito znaky fungovat i bez znalosti používaného kódování pro písmena s větším kódem. Podívejme se, co Python provede, když bychom v příkladu neuvedli řetězce v Unicode:

```
>>> print 'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.lower()
smíšenÝ pŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ
>>> print 'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.upper()
SMÍšENÝ PŘíPAD: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ
>>> print 'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.swapcase()
smÍšENÝ pŘíPAD: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ
>>> print 'SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ'.capitalize()
SmÍšenÝ pŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ
>>> print 'ĚŠČŘŽÝÁÍÉ'.islower()
False
>>> print 'ĚŠČŘŽÝÁÍÉ'.isupper()
False
```

Všechna písmena v rozsahu ASCII jsou správně převedena, ale všechna ostatní písmena byla ponechána beze změny. Metoda neví, co s nimi, proto je ponechá v původním tvaru. Uvědomte si, že je to přirozené chování metod v případech, kdy zpracovávají znaky, která nejsou písmenem. Existují také písmena, která nemají definováno odpovídající malé nebo velké písmeno (například německé ostré s). Všimněte si, že predikátyislower() a isupper() zde nepovažují testované řetěze ani za malá ani za velká písmena.
Unicode řetězce jsou ovšem speciálnější, informačně bohatší. Písmena jsou určena jednoznačně a existuje i jednoznačný vztah mezi malým a velkým písmenem (pokud to u znaku dává smysl). Metody pro převody písmen z malých na velká a naopak nemusí zohledňovat žádné nejednoznačnosti. Při použití Unicode se tedy i u ostatních lidských jazyků dostáváme do stejně pohodové situace, v jaké o podobných řetězcových operacích píší anglicky mluvící autoři ;-)

Zpracování textu v jazyce VBScript

VBScript vychází z jazyka BASIC. Díky tomu disponuje celou řadou zabudovaných funkcí pro práci s řetězci. V referenční příručce jsem jich napočítal nejméně 20 a to jsem nepočítal ty, které se vztahují k zpracování znaků v Unicode.
To znamená, že v jazyce VBScript můžeme s řetězci dělat v podstatě vše, co jsme si ukazovali v jazyce Python. V rychlosti si ukážeme jeho možnosti:

Rozdělování textu

Začneme funkcí Split:

```
<script type="text/vbscript">
Dim retezec
Dim seznam
retezec = "Toto je seznam slov."
seznam = Split(retezec) ' vrací pole
MsgBox seznam(1)
</script>
```

Pokud nám rozdělování řetězce v místech s bílými znaky (zde mezerami) nevyhovuje, můžeme stejně jako v Pythonu předepsat hodnotu oddělovače.
Pro opačný postup zde máme funkci Join — stejně jako v jazyce Python.

Vyhledávání a náhrada v textu

K vyhledávání podřetězce slouží funkce InStr, což je zjevně zkratka z anglického *In String*, tedy *v řetězci*.

```
<script type="text/vbscript">
Dim s, n
s = "Toto je dlouhý textový řetězec."
n = InStr(s, "dlouhý")
MsgBox "Slovo 'dlouhý' bylo nalezeno na pozici: " & CStr(n)
</script>
```

Návratová hodnota obvykle udává pozici v původním řetězci, na které začíná vyhledávaný podřetězec. Pokud není nalezen, vrací se nula. (V jazyce VBScript to nepůsobí žádný problém, protože se v něm indexuje od jedničky. To znamená, že hodnota nula nereprezentuje platnou hodnotu indexu.) Pokud má řetězec hodnotu Null, vrací se hodnota Null. Tím se poněkud komplikuje ověřování, zda nedošlo k chybě.
Stejně jako v Pythonu můžeme i v jazyce VBScript vymezit část původního řetězce, která se má prohledávat. Můžeme uvést index, od kterého se má začít vyhledávat:

```
<script type="text/vbscript">
Dim s, n
s = "Toto je dlouhý textový řetězec."
n = InStr(6, s, "dlouhý")  ' Hledej od pozice 6.
MsgBox "Slovo 'dlouhý' bylo nalezeno na pozici: " & CStr(n)
</script>
```

Narozdíl od Pythonu můžeme v jazyce VBScript předepsat také to, zda se mají při vyhledávání rozlišovat velká a malá písmena, či nikoliv. Pokud to neurčíme, velká a malá písmena se rozlišují.

Náhrady textu se provádějí prostřednictvím funkce Replace:

```
<script type="text/vbscript">
Dim s
s = "Příšerně zeleňoučký kůň úpěl ďábelské ódy."
MsgBox Replace(s, "zeleňoučký", "žluťoučký")
</script>
```

Poslední nepovinný argument určuje, kolik výskytů vyhledávaného vrorku se má nahradit. Pokud neurčíme jinak, nahrazují se všechny výskyty. Můžeme určit i počáteční pozici pro prohledávání a náhradu, jako u výše zmíněné funkce InStr.

Změna velikosti písmen

Změnu velikosti písmen provádíme v jazyce VBScript funkcemi UCase a LCase. Ekvivalent pythonovské metodycapitalize() tady nenajdeme.

```
<script type="text/vbscript">
Dim s
s = "SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ"
MsgBox LCase(s)
MsgBox UCase(s)
</script>
```

To je vše, čím se budeme v této učebnici zabývat. Pokud se chcete dozvědět více, projděte si seznam funkcí v nápovědě k VBScript.

Poznámka překladatele: V tomto případě (narozdíl od stejného řetězce v pythonovském příkladu, kdy řetězec nebyl uveden v Unicode) dojde ke korektnímu převodu na malá i velká písmena i u znaků s diakritikou. Je to dáno tím, že VBScript s řetězci jednoznačně spojuje určité kódování. Pokud je skript vložen do HTML souboru, pak bývá kódování předepsáno v jeho hlavičce. Stejně je tomu i u XML souborů. Pokud kódování není předepsáno vůbec, pak VBScript předpokládá, že se používá kódování definované v systému.

Zpracování textu v JavaScript

JavaScript je z našich tří jazyků pro zpracování textu vybaven nejhůře. I přesto jsou základní operace do určité míry podporovány. Ve srovnání s jazyky Python a VBScript jazyk JavaScript trpí nedostatky, které tkví spíše jen v množství "cingrlátek". JavaScript tato omezení kompenzuje silnou podporou *regulárních výrazů*. (Budeme se jimi zabývat v jednom z dalších témat.) Regulární výrazy výrazně rozšiřují možnosti, které poskytují výše zmíněné primitivní funkce, ale platíme za to zvýšením složitosti.

JavaScript, stejně jako Python, využívá k manipulaci s řetězci objektově orientovaný přístup. Všechny operace se provádějí prostřednictvím metod třídy String.

Rozdělování textu

Rozdělování textu se provádí metodou split():

```
<script type="text/javascript">
var seznam, retezec = "Toto je krátký řetězec";
seznam = retezec.split(" ");
document.write(seznam[1]);
</script>
```

Povšimněte si, že JavaScript vyžaduje zadání oddělovacího znaku. Neexistuje zde žádná předdefinovaná hodnota. Oddělovač může být definován i regulárním výrazem, takže operace rozdělení může být velmi důmyslná.

Vyhledávání textu

K vyhledávání textu se v jazyce JavaScript používá metoda search():

```
<script type="text/javascript">
var retezec = "Na Nilu ibisi kvílili bílí...";
document.write("'ibisi' se nachází na pozici: " + retezec.search(/ibisi/));
</script>
```

zmatky řetězec/vzorek

Znovu platí, že vyhledávaný řetězcový vzorek je ve skutečnosti regulárním výrazem. To znamená, že vyhledávání může mít velmi důmyslná pravidla. Na druhou stranu neexistuje možnost vymezit část původního řetězce, která se má prohledávat, zadáním počáteční pozice (ačkoliv i tuto možnost můžeme simulovat prostřednictvím regulárního výrazu). JavaScript podporuje i jinou vyhledávací operaci s mírně odlišným chováním. Nazývá ze match() ([meč], zde ve smyslu *odpovídat*, *pasovat*). V této části se metodou match() nebudeme zabývat.

Náhrada textu

zmatky řetězec/vzorek

K nahrazování textu se používá metoda replace().

```
<script type="text/javascript">
var retezec = "Kočka leze dírou, pes dveřmi...";
document.write(retezec.replace(/dveřmi/, "oknem"));
</script>
```

A znovu připomeňme, že vyhledávaný vzorek může být zadán regulárním výrazem. Myslím, že už vidíte ten obecný vzor. Operace náhrady nahrazuje všechny výskyty vyhledávaného vzorku a, pokud mohu říci, neexistuje žádný způsob, jak dosáhnout náhrady pouze jednoho výskytu, aniž byste řetězec nejdříve rozdělili a pak jej opět pospojovali dohromady.

zmatky řetězec/vzorek, počet náhrad

Změna velikosti písmen

Pro změnu velikosti písmen máme k dispozici dvě funkce: toLowerCase() a toUpperCase().

```
<script type="text/javascript">
var retezec = "SMÍšenÝ PŘípad: ěščřžýáíéúůó ĚŠČŘŽÝÁÍÉÚŮ";
document.write(retezec.toLowerCase()+ "<br>");
document.write(retezec.toUpperCase()+ "<br>");
</script>
```

K těmto funkcím můžeme stěží co dodat. Jednoduchým způsobem provádějí jednoduchou činnost. JavaScript — narozdíl od ostatních jazyků, o kterých se bavíme — nabízí řadu speciálních textových funkcí pro zpracování HTML. Odhaluje tím své kořeny jazyka určeného pro programování ve webovém prostředí. Nebudeme se jimi zabývat. Popis naleznete ve standardní dokumentaci.

Tím uzavíráme pohled do světa zpracování textu. Doufám, že jste tím získali nástroje, které potřebujete ke zpracování textu ve vašich projektech. Závěrečná rada zní: Při zpracování textu si vždy pročtěte dokumentaci zvoleného jazyka.Řešení nejzákladnějších programátorských úloh je často podpořeno mocnými nástroji.

Zapamatujte si

- Soubory musíme před použitím otevřít.
- Ze souborů můžeme obvykle jen číst nebo do nich můžeme jen zapisovat, ale ne obojí současně.
- Funkce readlines() jazyka Python přečte všechny řádky souboru najednou, zatímco funkce readline()přečte jen jeden řádek. Může nám to pomoci šetřit pamětí.
- Po použití soubor uzavřete.

Ošetření chyb

O čem si budeme povídat?

- Krátce z historie práce s chybami.
- Dvě techniky práce s chybami.
- Jak v našem kódu definovat a signalizovat chybu, která má být ošetřena jinde.

Krátce z historie práce s chybami

Z hlediska způsobu zpracování chyb je z našich tří jazyků VBScript ten nejbizarnější. Je to dáno tím, že staví na základech jazyka BASIC, který patří k jedněm z prvních programovacích jazyků (kolem roku 1963). Způsob zpracování chyb v jazyce VBScript patří k těm místům, ze kterých je zmíněné dědictví jasně vidět. Pro naše účely to není špatné, protože mi to dává příležitost k vysvětlení, proč VBScript používá právě takový přístup. Vysvětlíme si historii způsobu zpracování chyb od jazyka BASIC, přes Visual Basic, až k VBScript. Poté se podíváme na mnohem modernější přístup, který ukázkovým způsobem využívají JavaScript a Python.

V tradiční verzi jazyka BASIC byly se na všech řádcích programu psala čísla řádků. Přesun řízení se prováděl skokem na určitý řádek použitím příkazu GOTO. (Příklad jsme si ukázali v rámci tématu Větvení.) Byl to v podstatě jediný možný způsob řízení. V prostředí s takovými vlastnostmi byl běžný způsob ošetřování chyb založen na deklaraci proměnné ERRORCODE, ve které se ukládala číselná hodnota. Když se v programu vyskytla chyba, nastavil se obsah proměnné ERRORCODE na odpovídající hodnotu — nepodařilo se otevřít soubor, neslučitelnost typů, přetečení operátoru, a podobně.

Uvedený přístup vedl k psaní kódu, který vypadal podobně, jako následující úsek fiktivního programu:

```
1010 LET DATA = INPUT FILE
1020 CALL FUNKCE_PRO_ZPRACOVANI_DAT
1030 IF NOT ERRORCODE = 0 GOTO 5000
1040 CALL JINA_FUNKCE
1050 IF NOT ERRORCODE = 0 GOTO 5000
1060 REM POKRACUJ VE ZPRACOVANI TAKTO...
...
5000 IF ERRORCODE = 1 GOTO 5100
5010 IF ERRORCODE = 2 GOTO 5200
5020 REM DALSI PRIKAZY IF
...
5100 REM ZDE ZPRACUJ CHYBOVY KOD 1.
...
5200 REM ZDE ZPRACUJ CHYBOVY KOD 2.
```

Vidíme, že skoro polovina hlavního programu se zabývá zjišťováním, zda nenastala chyba. Časem byl zaveden o něco elegantnější mechanismus, ve kterém se detekce chyba a jejich ošetření částečně přesunulo do interpretu jazyka. Vypadal nějak takto:

```
1010 LET DATA = INPUTFILE
1020 ON ERROR GOTO 5000
1030 CALL FUNKCE_PRO_ZPRACOVANI_DAT
1040 CALL JINA_FUNKCE
...
5000 IF ERRORCODE = 1 GOTO 5100
5010 IF ERRORCODE = 2 GOTO 5200
```

K popisu umístění kódu pro ošetření chyby zde stačí jeden řádek. Pokud funkce narazila na chybu, musela i nadále nastavovat hodnotu proměnné ERRORCODE, ale velmi se zjednodušil zápis (a čtení!) kódu.

No a co to má společného s námi? Je to docela prosté. Tento způsob zpracování chyb i nadále používá Visual Basic, ačkoliv čísla řádků byla nahrazena uživatelsky přívětivějšími návěštími. VBScript — jako potomek jazyka Visual Basic — používá výrazně ořezanou verzi téhož mechanismu. To ve svém důsledku znamená, že nám VBScript dává na vybranou: buď budeme ošeřovat chyby lokálně, nebo je budeme zcela ignorovat.

Pokud se rozhodneme pro ignorování chyb, zapíšeme to takto:

```
On Error Goto 0  ' 0 říká "nikam neskákej"
NejakaFunkce()
NejakaJinaFunkce()
...
```

Pokud se rozhodneme pro lokální ošetření chyb, použijeme zápis:

```
On Error Resume Next
NejakaFunkce()
If Err.Number = 42 Then
' tady ošetři chybu
NejakaJinaFunkce()
...
```

Takový zápis vypadá trochu zpátečnicky. Ve skutečnosti prostě odráží výše popsaný historický proces.

V případě výskytu chyby se interpret standardně zachová tak, že uživateli zobrazí zprávu a zastaví provádění programu. Právě toto se stane, pokud při určování způsobu zpracování chyby použijeme zápis Goto 0. To znamená, že zápisem Goto 0 vypínáme lokální řízení a říkáme interpretu, aby se zachoval obvyklým způsobem.

Předpis pro zpracování chyby obsahující Resume Next nám buď dovolí předstírat, že chyba vůbec nenastala, nebo si můžeme otestovat objekt pro popis chyby (je pojmenován Err). Zajímá nás zejména jeho číselná složka Number. (Jde o naprosto stejnou techniku, jako výše ukázané testování ERRORCODE.) Objekt Err nese ještě další informační položky, které nám mohou pomoci vyrovnat se s nastalou situací méně katastrofickým způsobem, než je zastavení programu. Můžeme například zjistit, v jakém místě k chybě došlo (v jakém objektu, v jaké funkci a podobně). Můžeme také získat textový popis chyby, který můžeme uživateli vypsat jako součást hlášení o chybě, nebo jej můžeme zapsat do log souboru. Typ chyby můžeme v objektu Err změnit použitím jeho metody Raise. Tuto metodu můžeme použít i při generování našich vlastních chyb, které nastaly v našich vlastních funkcích.

Jako příklad mechanismu ošetřování chyby v jazyce VBScript is uveďme případ, kdy dochází k dělení nulou:

```
<script type="text/vbscript">
Dim x, y, vysledek
x = CInt(InputBox("Zadejte dělence: "))
y = CInt(InputBox("Zadejte dělitele: "))
On Error Resume Next
vysledek = x/y
If Err.Number = 11 Then ' Dělení nulou
vysledek = Null
End If
On Error GoTo 0 ' ošetřování chyb opět vypneme
If VarType(vysledek) = vbNull Then
MsgBox "CHYBA: Operace nemohla být provedena."
Else
MsgBox CStr(x) & " děleno " & CStr(y) & " je rovno " & CStr(vysledek)
End If
</script>
```

Upřímně řečeno, uvedený přístup moc pěkný není. A zatímco obdivování dávné historie může být balzámem pro duši, moderní programovací jazyky, včetně jazyků Python a JavaScript, nabízejí mnohem elegantnější způsoby ošetřování chyb. Podívejme se, o jaké mechanismy jde.

Ošetřování chyb v jazyce Python

Zpracování výjimek

V modernějších programovacích prostředích se vyvinul alternativní způsob práce s chybami. Je znám pod pojmem *ošetření výjimek* a je založen na tom, že funkce *vrhají* (*throw*) nebo, jinými slovy, *vyvolávají* (*raise*) výjimky (exception [iksepšn]). Systém si pak zajistí vyskočení z aktuálního bloku kódu na nejbližší blok pro ošetření výjimky. V systému se nachází také blok kódu, který *zachytí* (*catch*) všechny výjimky, které dosud nebyly zpracovány někde jinde. Poté obvykle zobrazí chybové hlášení a ukončí běh aplikace.

K velkým výhodám tohoto stylu ošetřování chyb patří mnohem lepší čitelnost hlavní funkčnosti programu. Je to dáno tím, že nedochází k mísení s kódem pro ošetřování chyb. Jinými slovy, v bloku kódu si můžeme číst aniž bychom byli jakkoliv nuceni číst kód pro ošetřování chyb.

Podívejme se, jak se uvedený styl programování používá v praxi.

Try/Catch

Blok kódu pro ošetření výjimek se trochu podobá bloku if...then...else:

```
try:
# Zde se umístí hlavní část kódu programu.
except TypVyjimky:
# Zde se bude zpracovávat jmenovaná výjimka.
except JinyTypVyjimky:
# Zde se ošetřuje jiná výjimka.
else:
# Zde umístíme úklidový kód, který se provede
# v případě, že nenastane žádná výjimka.
```

Python se pokouší provádět příkazy mezi příkazy try a prvním except. Pokud dojde k chybě, provádění příkazů v bloku za try se zastaví a skočí se dolů k příkazu except. Postupně se procházejí jednotlivé příkazy except, dokud se nenajde ten, který odpovídá typu chyby (neboli *výjimky*). Pokud se nalezne, provede se blok bezprostředně následujícího kódu. Pokud se nenalezne žádný odpovídající příkaz except, předává se nalezená chyba dál, do dalších (vyšších) úrovní programu. Pokud není nalezen odpovídající except, dostane se chyba až na úroveň interpretu jazyka Python, který chybu zachytí, zobrazí chybové hlášení a zastaví provádění programu. V našich programech jsme zatím pozorovali právě takový projev, protože jsme zatím neuměli chybu zachytit a zpracovat sami.

Pokud v bloku try k žádným chybám nedojde, pak se provede blok za else. Ale v praxi se této možnosti využívá velmi zřídka. Poznamenejme, že příkaz except, u kterého není uveden žádný typ chyby, zachytí chyby *všech* typů, které dosud nebyly zpracovány. Využívání této formy příkazu se ale obecně nepovažuje za dobrý nápad. Výjimku ovšem představuje jeho používání v nejvyšší úrovni programu, kdy chceme zabránit tomu, aby Python uživateli zobrazil velmi technickou podobu chybového hlášení. Obecná podoba příkazu except nám v tomto případě umožní odchytit všechny dosud neošetřené chyby a zobrazit přívětivější hlášení o *ukončování programu*.

Za zmínku stojí, že součástí instalace Pythonu je i modul traceback, který nám umožní extrahovat různé doplňkové informace o zdroji chyby. To se nám může hodit při vytváření log souboru a pro podobné akce. Modulem traceback se zde zabývat nebudeme. V případě potřeby naleznete popis jeho vlastností a použití ve standardní dokumentaci modulů.

A jak to vše skutečně funguje si ukážeme na příkladu:

```
hodnota = raw_input("Zadej delitele: ")
try:
hodnota = int(hodnota)
print "42 / %d = %d" % (hodnota, 42 / hodnota)

except ValueError:
print "Hodnotu nelze prevest na cele cislo."

except ZeroDivisionError:
```

```
print "Neni povolena nulova hodnota."

except:
print "Stalo se neco neocekavaneho."

else:
print "Program skoncil uspesne."
```

Pokud program spustíme a místo čísla zadáme nějaký řetězec, zobrazí se zpráva vypisovaná ve větvi ValueError(chyba hodnoty). Pokud zadáme nulu, zobrazí se zpráva pro ZeroDivisionError (chyba při dělení nulou). Pokud zadáme platné číslo, zobrazí se výsledek a zpráva o úspěšném ukončení programu.

Try/Finally

Existuje ještě jeden typ bloku, který souvisí s výjimkami. Umožňuje zapsat kód pro úklid prováděný i poté, co nastala chyba. Nazývá se try...finally (try [tray] = zkus, pokus se vykonat; finally [fajnly] = nakonec) a typicky se používá pro zavírání souborů, vyprazdňování vyrovnávacích pamětí (buffer) na disk a podobně. Blok finally je proveden vždy jako poslední nezávisle na tom, co se stane v sekci try. Pokud nenastane výjimka, prostě se provede. Pokud nastane výjimka, zapamatuje se její objekt, kód bloku finally se provede a zapamatovaná výjimka se znovu vyvolá.

```
try:
# Kód, související s účelem programu.
finally:
# V této části provádíme úklidové akce nezávisle
# na tom, zda v bloku try nastala chyba či nikoliv.
```

Síla této konstrukce se projeví při kombinaci s blokem try/except. Konkrétní volba pořadí zanoření těchto bloků nepřináší významné výhody. Pořadí zpracování příkazů zůstává v obou případech stejné. Osobně používám blok try/finally obvykle jako vnější, protože si tím připomínám, že se blok větve finally provede jako poslední. Ale z pohledu Pythonu je to jedno.

Příklad:

```
print 'Start programu.'
try:
try:
data = file('data.dat')
hodnota = int(data.readline().split()[2])
print 'Hodnota je %d.' % (hodnota/(42-hodnota))
except ZeroDivisionError:
print 'Hodnota byla 42.'
finally:
data.close()
print 'Konec programu.'
```

V tomto případě dojde k uzavření souboru vždy, nezávisle na tom, zda v bloku try/except vznikne výjimka. Všimněte si odlišnosti chování ve vztahu k větvi else v konstrukci try/except/else, protože blok v else se zavolá jen v případě, kdy nedojde k *žádné* výjimce. To by znamenalo, že by nedošlo k uzavření souboru. Pokud bychom zase kód pro uzavření souboru umístili jednoduše mimo konstrukci try/except, pak by se soubor neuzavřel v případě, kdy by nastala jiná výjimka, než ZeroDivisionError. Takže pouze konstrukce try/finallyzajistí, že k uzavření souboru dojde *vždy*.

Poznámka překladatele: V souboru data.dat se očekává alespoň jeden řádek, který obsahuje alespoň tři slova (řetězce nerozdělené mezerami) a třetí slovo má charakter čísla. Příklad obsahu:

```
xxx yyy 40
```

Zadáním hodnoty 42 (do souboru) vyvoláme výjimku ZeroDivisionError. Pokud místo čísla zadáme například řetězec ccc, vznikne jiná výjimka (ValueError) související s tím, že se řetězec nedaří převést na číslo. Pokud místo čísla neuvedeme vůbec nic, vznikne při následném volání metody split() k vygenerování kratšího seznamu, takže nebude existovat položka s indexem 2. V takovém případě vznikne výjimka IndexError.

Osobně se mi nelíbí otvírání souboru na jiné úrovni (v zanořeném try), než na jaké se provádí uzavírání — i když to funguje. Podle mého názoru by se soubor měl zavírat na stejné úrovni v kódu, na které byl otevřen.

Platí to obecně, ale nejvýrazněji je to vidět v případě, kdy jedna z akcí (otevření/zavření souboru) je umístěna uvnitř funkce a druhá vně. Například funkce, které dostává jako argument otevřený soubor, by jej neměla zavírat. Ono to sice může fungovat naprosto bez problémů, ale může dojít ke zmatkům v naší hlavě, když si čteme zdrojový text. Důvod spočívá v tom, že při zápisu používání funkce je tato akce skryta našemu zraku a nemusíme si uvědomit, co se vevnitř děje. Výjimku představují situace, kdy funkce svým jménem napovídá, že se předaný soubor uvnitř otevře, respektive uzavře.

Pokud tedy toto pravidlo *selského rozumu* napasujeme na výše uvedený příklad, pak osobně dávám přednost následujícímu zápisu příkladu (navíc přidán příkaz print za uzavření souboru, abychom zviditelnili provedení bloku kódu):

```
print 'Start programu.'
try:
data = file('data.dat')
try:
hodnota = int(data.readline().split()[2])
print 'Hodnota je %d.' % (hodnota/(42-hodnota))
except ZeroDivisionError:
print 'Hodnota byla 42.'
finally:
data.close()
print 'Soubor uzavřen.'
print 'Konec programu.'
```

Je tady ale ještě jeden zádrhel. Pokud by navíc selhalo i otvírání zouboru data.dat, dojde k výjimce proměnná data nebude naplněna (případně se váže na předchozí hodnotu). To znamená, že v sekci finally, ve kterém se budeme pokoušet o volání metody close() objektu, který neexistuje. Proto je lepší zapsat:

```
print 'Start programu.'
data = file('data.dat')
try:
```

```
try:
    hodnota = int(data.readline().split()[2])
    print 'Hodnota je %d.' % (hodnota/(42-hodnota))
except ZeroDivisionError:
    print 'Hodnota byla 42.'
finally:
    data.close()
    print 'Soubor uzavřen.'
print 'Konec programu.'
```

V takovém případě je ovšem sporné už samotné použití try/finally, protože v případě, kdy se soubor nepovede otevřít, nedostaneme se vůbec do následující konstrukce. Zavírání souboru stejně v takovém případě není možné (viz předchozí odstavec). Pokud se soubor povede otevřít, nepotřebujeme uzavírání souboru vkládat do bloku finally. Osobně bych se proto přiklonil k naprosto jednoduchému řešení bez vnější konstrukce try/finally.

```
print 'Start programu.'
data = file('data.dat')

try:
    hodnota = int(data.readline().split()[2])
    print 'Hodnota je %d.' % (hodnota/(42-hodnota))
except ZeroDivisionError:
    print 'Hodnota byla 42.'
except:
    print u'Nastala jiná výjimka.'
    data.close()
print 'Konec programu.'
```

Závěr: Původní příklad není tak jednoduchý, jak vypadá.

Generování chyb

Jakým způsobem můžeme generovat výjimky — dejme tomu uvnitř modulu —, které má zachytit někdo jiný? V jazyce Python je pro tento případ vyhrazeno klíčové slovo raise:

```
delenec = 42
delitel = input('Jakou hodnotou chcete dělit číslo 42? ')
if delitel == 0:
    raise ZeroDivisionError()
```

Tento kód vygeneruje výjimku ZeroDivisionError, která může být zachycena v bloku try/except. Zbytku programu se to jeví naprosto stejně, jako kdyby chybu vygeneroval přímo Python. Klíčovým slovem raise se předepisuje také předávání chyby zevnitř bloku except do vyšších úrovní programu. Při výskytu chyby můžeme například chtít provést nějakou lokální akci, dejme tomu zapsat záznam do log souboru, ale poté chceme, aby se o dalších akcích rozhodlo na vyšších úrovních programu. Použití může vypadat takto:

```
logsoubor = file('errorlog.txt', 'w')

def f(hodnota):
    try:
        return 127 / (42-hodnota)
    except ZeroDivisionError:
        logsoubor.write('Hodnota byla 42.')
        raise

try:
    f(42)
except ZeroDivisionError:
    print 'Nastala chyba. Zkuste znovu.'
```

Povšimněte si, jak funkce f() zachytává chybu, zapisuje zprávu do souboru se záznamem o chybě a poté předává zachycenou výjimku ke zpracování v bloku kódu, který se nachází ve vnější, obalující konstrukcitry/except.

Poznámka překladatele: S *log soubory*, tedy se soubory určenými pro záznam (protokolování) chyb a zvláštních stavů, se většinou zachází tak, že se záznamy neustále připisují na konec. Prakticky to znamená, že by se log soubor měl otvírat pro zápis za konec souboru, tedy v režimu *append* (druhý parametr s hodnotou 'a').

Další zásada vyplývá ze skutečnosti, že u otevřeného souboru není zaručeno, že je veškerý zapisovaný obsah skutečně fyzicky uložen na disku. Pokud toho chceme dosáhnout, pak můžeme ve vhodných okamžicích provádět takzvané vyprázdnění vyrovnávací paměti (flush).

Při zápisu do log souboru s chybami, kdy se předpokládá nízký počet zápisů, bývá praktičtější log soubor otevřít před každým zápisem a poté ho hned zavřít. Pokud aplikace havaruje, máme jistotu, že se neztratilo několik posledních zápisů. Ve výše uvedeném příkladě je log soubor otevřen na začátku a dokonce jsme jej zapoměli zavřít. Při výskytu chyby se původní obsah přepisuje, takže budeme mít zapsán jen poslední záznam. V mnoha jednoduchých případech to stačí, ale proč bychom to nemohli udělat pořádněji, když to není o moc složitější:

```
def f(hodnota):
    try:
        return 127 / (42-hodnota)
    except ZeroDivisionError:
        logsoubor = file('errorlog.txt', 'a') # Otevřeme pro připsání na konec,
        logsoubor.write('Hodnota byla 42.\n') # zapíšeme hodnotu
        logsoubor.close()                     # a soubor uzavřeme.
        raise

try:
    f(42)
except ZeroDivisionError:
```

```
print u'Nastalo dělení nulou. Zkuste znovu.'
```

V praktických případech často používáme jediný log soubor a chceme do něj zapisovat na více místech v programu. Museli bychom tedy na více místech opakovat výše uvedené okomentované řádky. Problém by nastal v situaci, kdy se rozhodneme například změnit jméno log souboru. V takovém případě by nám mělo v hlavě varovně zasvítit pravidlo **DRY** z anglického *Do not Repeat Yourself*, čili volně *Neopakujte se*. Věc jednoduše vyřešíme tím, že si pro zápis na konec log souboru vytvoříme vlastní funkci log(). S jejím využitím pak výsledek bude vypadat nějak takto:

```
def log(zprava):
    logsoubor = file('errorlog.txt', 'a') # Otevřeme pro připsání na konec,
    logsoubor.write(zprava)               # zapíšeme hodnotu
    logsoubor.close()                     # a soubor uzavřeme.

def f(hodnota):
    try:
        return 127 / (42-hodnota)
    except ZeroDivisionError:
        log('Hodnota byla 42.\n')  # Zápis zprávy do log souboru.
        raise

try:
    f(42)
except ZeroDivisionError:
    print u'Nastalo dělení nulou. Zkuste znovu.'
```

V tomto okamžiku byste si mohli říci. *Proč bych za každou zprávu nepřidával automaticky přechod na nový řádek přímo ve funkci log()?* Nedoporučuji to. Uvědomte si, že v takovém případě byste se zbavili možnosti zapisovat postupně několika voláními funkce více hodnot na jeden řádek log souboru. Místo toho je vhodnější vytvořit další, specializovanější funkci, která využívá výše definované, obecnější funkce log(). Speciální funkce může například do log souboru zapisovat i datum, čas a jmého přihlášeného uživatele:

```
def log(zprava):
    logsoubor = file('errorlog.txt', 'a') # Otevřeme pro připsání na konec,
    logsoubor.write(zprava)               # zapíšeme hodnotu
    logsoubor.close()                     # a soubor uzavřeme.

def logErr(text):
    import time                           # Importujeme potřebné moduly.
    import getpass
    tim = time.strftime('%c')             # Získáme časovou značku.
    usr = getpass.getuser()               # Získáme jméno uživatele.
    log('%s %s: %s\n' % (tim, usr, text)) # Zapíšeme zformátovaný řádek.

def f(hodnota):
    try:
        return 127 / (42-hodnota)
    except ZeroDivisionError:
        logErr('Hodnota byla 42.')  # Zápis zprávy do log souboru.
        raise

try:
    f(42)
except ZeroDivisionError:
    print u'Nastalo dělení nulou. Zkuste znovu.'
```

Za účelem zjemnění řízení našeho programu (nebo diagnostiky chyb) můžeme definovat své vlastní typy výjimek. Činíme tak prostřednictvím nových tříd výjimek. (S definicemi tříd jsme se krátce seznámili rámci tématu Data, datové typy a proměnné a podrobněji se s nimi setkáme ještě později v kapitole věnované objektově orientovanému programování.) Pro tento účel obvykle definujeme třídu velmi prostou, která nedefinuje žádný další obsah a která je pouze odvozena od standardní bázové třídy Exception. Používá se jako *chytré návěští*, které se dá rozpoznávat v příkazech except. Spokojme se s následujícím stručným příkladem:

```
class BrokenError(Exception): pass

try:
    raise BrokenError
except BrokenError:
    print u'Narazili jsme na chybu při zpracování.'
```

Poznámka překladatele: Vzhledem k následující autorově poznámce jsem ponechal původní anglický identifikátor. Připojuji se k výzvě dodržovat níže uvedenou konvenci.

Povšimněte si, že jsme při tvorbě jména použili konvenci, kdy se na konec jména třídy dává přípona "Error" (čili chyba). Povšimněte si, že *dědíme* chování obecné třídy Exception ([iksepšn], výjimka) tím, že její jméno uvedeme do závorek za jménem definované třídy. K dědičnosti se podrobněji dostaneme v kapitole věnované objektově orientovanému programování.

Ještě poslední poznámka k části týkající se generování chyb. Prozatím jsme pro předčasné ukončování našich programů prováděli importováním modulu sys a voláním jeho funkce exit(). Jiný způsob, kterým dosáhneme naprosto stejného výsledku, spočívá ve vyvolání výjimky SystemExit:

```
>>> raise SystemExit
```

Hlavní výhoda tohoto přístupu spočívá v tom, že nemusíme nejdříve provést import sys.

Poznámka překladatele: Osobně se k tomuto postupu moc nepřikláním. Volání funkce exit() je známé i z jiných jazyků a jiným čtenářům vašeho zdrojového textu může volání sys.exit() připadat přirozenější.

JavaScript

V jazyce JavaScript se zpracování chyb provádí velmi podobně, jako v jazyce Python. Jen místo pythonovských klíčových slov try, except a raise se používají klíčová slova try, catch ([keč], chytit) a throw (vrhnout, hodit).

Ukážeme si pár příkladů, ale principy zůstávají naprosto stejné jako v jazyce Python. V jazyce JavaScript ale nemáme konstrukci try/finally.

Zachytávání chyb

Zachytávání chyb v bloku kódu se předepisuje klíčovým slovem try a sadou příkazů catch téměř stejně, jako v Pythonu:

```
<script type="text/javascript">
try {
var x = NeexistujiciFunkce();
document.write(x);
}
catch(err) {
document.write("Došlo k chybě.");
}
</script>
```

Vyvolání chyby

V jazyce Python jsme k vyvolání chyby používali klíčové slovo raise. V jazyce JavaScript používáme podobným způsobem throw(). Také v jazyce JavaScript si můžeme vytvořit vlastní typy chyb, jako v Pythonu. Ale mnohem jednodušší způsob spočívá v použití řetězce.

```
<script type="text/javascript">
try {
throw("Nová chyba");
}
catch(e) {
if (e == "Nová chyba")
document.write("Zachytili jsme novou chybu.");
else
document.write("Nenastala nová chyba.");
}
</script>
```

Poznámka překladatele: Řetězce se pro výjimky používaly dříve i v jazyce Python. Z důvodů zpětné kompatibility jsou dosud podporovány, ale u nových programů se jejich používání nedoporučuje. Do budoucna se plánuje odstranění této možnosti.

To je vše, co si o zpracování chyb řekneme. V tématech pro pokročilé uvidíte použití mechanismu pro zpracování chyb v praxi, spolu s použitím dalších základních konceptů, jako jsou posloupnosti, cykly a větvení. V tomto okamžiku již máte k dispozici všechny podstatné nástroje, které potřebujete pro vytváření mocných programů. Možná byste si teď měli zkusit nějaké vlastní programy vytvořit. Stačí pár, jen abyste dostali popisované mechanismy do hlavy před tím, než se pustíme do dalších témat. Pár námětů:

- Jednoduchá hra, jako je OXO nebo šibenice.
- Jednoduchá databáze — mohla by vycházet z našeho příkladu záznamníku s adresami — pro ukládání detailů vaší sbírky videozáznamů, DVD nebo CD.
- Diář, který vám umožní ukládat důležité události a schůzky. A pokud se cítíte být děsný frúd, nechť se automaticky objevuje připomínka.

Abyste se s výše uvedenými úkoly vyrovnali, budete muset použít všechny rysy jazyka, se kterými jsme se dosud seznámili, a možná i pár dodávaných modulů. Nezapomeňte občas nahlédnout do dokumentace. Pravděpodobně v ní najdete pár věcí, které vám ulehčí práci. Nezapomínejte taky na užitečnost interaktivního režimu (>>>). Zkoušejte si v něm nové věci, dokud nepochopíte, jak fungují. Teprve poté přeneste získané znalosti do vašeho programu. Tímto způsobem pracují i profesionálové. A co je taky důležité, dobře se bavte!

Nashledanou v části pro pokročilé :-)

Zapamatujte si

- V jazyce VBScript testujeme chybové kódy příkazem if.
- Výjimky se zachytávají v Pythonu zachytávají pomocí except a v JavaScript pomocí catch.
- Vygenerování výjimky můžeme v Pythonu předepsat klíčovým slovem raise, v JavaScript pomocí throw.
- Typy chyb jsou v jazyce Python vyjadřovány pomocí tříd, v JavaScript pomocí řetězců.

Prostory jmen

Úvod

Už slyším, jak se ptáte… Co to je ten *prostor jmen* (namespace)? No, dá se to těžko vysvětlit. Ne proto, že by to bylo nějak zvlášť komplikované, ale spíš proto, že se k tomu každý jazyk staví trochu jinak. Samotný koncept je docela přímočarý.

Prostor jmen je prostor nebo oblast uvnitř programu, kde je jméno (proměnné, třídy, atd.) platné.

Dřívější programovací jazyky (jako třeba BASIC) pracovaly pouze s *globálními proměnnými*, to znamená s takovými proměnnými, které byly vidět z celého programu — dokonce uvnitř funkcí. To činilo udržovatelnost programů velmi obtížnou, protože pro každý kousek programu bylo velmi snadné změnit nějakou proměnnou, aniž by se to ostatní části programu nějak dozvěděly. Tomuto jevu se říká *vedlejší efekt*. Novější jazyky (včetně moderních verzí jazyka BASIC) tento problém obcházejí zavedením prostorů jmen. (Jazyk C++ jde v tomto směru do extrému tím, že umožňuje programátorovi vytvořit svůj vlastní prostor jmen kdekoliv uvnitř programu. To ocení zvláště tvůrci knihoven, kteří chtějí dosáhnout jednoznačnosti jmen svých funkcí i v případě, kdy se současně použijí knihovny jiných tvůrců.)

Jak to řeší Python?

V systému Python vytváří každý modul svůj vlastní prostor jmen. Pokud chceme používat jména jeho částí, musíme jim buď předřadit jméno modulu, nebo musíme explicitně importovat požadovaná jména dovnitř prostoru jmen našeho modulu. Není to pro nás nic nového. Už jsme to dělali při práci s moduly sys a string. V určitém smyslu vytváří svůj prostor jmen i definice třídy. Takže pokud chceme zpřístupnit metodu nebo vlastnost třídy, musíme nejdříve použít jméno instance nebo třídy.

V Pythonu existují jen 3 prostory jmen (nebo *rozsahy platnosti* — scopes):

1. Lokální rozsah — jména definovaná uvnitř funkce nebo metody.

2. Rozsah v rámci modulu — jména definovaná uvnitř souboru modulu.
3. Zabudovaná jména — jména definovaná uvnitř samotného systému Python, která jsou přístupná vždy.

No dobrá. Ale jak to vše dáme dohromady, když proměnné v různých prostorech jmen mají stejné jméno? A nebo, jak se odkazujeme na jméno, které se nenachází v aktuálním prostoru jmen? Podívejme se nejdříve na první případ: Pokud se funkce odvolává na proměnnou nazvanou X a uvnitř funkce existuje nějaká proměnná X(tj. uvnitř prostoru s lokálními proměnnými), pak to bude právě tato lokální proměnná, kterou Python uvidí a použije. Je věcí programátora, aby se vyhnul střetům jmen, kdy má nějaká lokální proměnná stejné jméno jako proměnná modulu a kdy bychom mohli chtít zpřístupnit obě najednou. Existence lokální proměnné v takovém případě maskuje existenci globální proměnné. Obecně bychom měli globální proměnné používat co nejméně. Obvykle bývá lepší, když běžnou, lokálníproměnnou předáváme jako parametr volané funkce a vrací se nám s modifikovaným obsahem.

Poznámka překladatele ke globálním proměnným: Z pohledu začátečníka se může zdát používání globálních proměnných velmi výhodné. Jednoduše přece uvedeme jméno proměnné, které je známé ve všech místech programu! V čem je problém? Postupně zjistíte, že těch problémů může být hned několik. Zdánlivá jednoduchost může později věci zkomplikovat:

- Při zvětšování programu používáme čím dál víc jmen a začnou se nám špatně vymýšlet. Pokud chceme jméno proměnné později změnit, musíme je měnit na mnoha různých místech. Pro vhodný textový editor to nemusí být velký problém, ale...
- Pokud jazyk umožňuje práci s lokálními proměnnými, pak obvykle existence lokální proměnné zamaskuje existenci stejnojmenné globální proměnné — platí to i pro Python. Při čtení zdrojového textu si vůbec nemusíme všimnout toho, že je stejnojmenná lokální proměnná již definována. A najednou se program začne chovat jinak, než bychom čekali.
- Použitím jména globální proměnné někde v kódu vytváříme obtížně kontrolovatelnou vazbu na zbytek systému. Nejde jen o to, že existuje vazba na globálně pojmenovanou proměnnou, ale jejím prostřednictvím se najednou ovlivňuje činnost všech ostatních částí kódu, které s globální proměnnou pracují. Jinými slovy, někdy je velmi obtížné zajistit, aby v globální proměnné byla uložena právě ta hodnota, která tam má být uložena. Zdánlivě nevinná změna obsahu, provedená v jednom místě, může neočekávaným způsobem ovlivnit činnost jiných částí aplikace.

Druhý případ, kdy se odkazujeme na jméno, které se nenachází mezi lokálními, se řeší následujícím způsobem: Funkce prohlédne svůj lokální prostor. Pokud zde požadované jméno nenalezne, hledá v prostoru modulu. A pokud není nalezeno ani zde, hledá se v prostoru zabudovaných jmen (builtin scope). Jediná nepříjemnost nastane v situaci, kdy chceme přiřadit hodnotu externí proměnné. Při normálním postupu by vznikla nová proměnná tohoto jména, ale tomu se chceme vyhnout. Takže aby se nevytvořila lokální proměnná daného jména, musíme určit, že se jedná o jméno globální.

Vše si ukážeme v akci na následujícím příkladu (jde o čistě ilustrační příklad):

```
# Proměnné na úrovni modulu.
W = 5
Y = 3

# Parametry se chovají jako proměnné náležející funkci. Takže X patří
# do lokálního prostoru.
def spam(X):

    # Funkci oznámíme, že má W hledat na úrovni modulu a nevytvářet svou
    # proměnnou W.
    global W

    Z = X*2 # Nová proměnná Z je vytvořena jako lokální.
    W = X+5 # Práce s W na úrovni modulu -- viz výše.

    if Z > W:
        # pow je jméno 'zabudované' funkce.
        print pow(Z, W)
        return Z
    else:
        return Y # Lokální Y neexistuje, takže se použije globální.
```

Pokud importujeme modul, jako je například sys, stane se jméno sys lokálně dostupným jménem. Poté můžeme jména uvnitř prostoru jmen modulu sys zpřístupnit použitím takzvaného *kvalifikovaného jména*, jak jsme si ukázali dříve. (Kvalifikované jméno se od holého liší tím, že holému jménu předřadíme takzvaný kvalifikátor, který má podobu dalšího jména, vhodně spojeného s původním holým jménem. V jazyce Python se obě části oddělují tečkou. Například v jazyce C++ se oddělují dvěma dvojtečkami.)

Pokud napíšeme

```
from sys import exit
```

pak v lokálním prostoru jmen zpřístupníme pouze funkci exit. Nemůžeme použít žádné jiné jméno z modulu sys a dokonce ani samotné jméno modulu sys.

Ještě v jazyce BASIC...

BASIC volí ve srovnání z jazykem Python opačný přístup. Všechny vytvořené proměnné se automaticky stávají globálními (aby byla zachována kompatibilita, tedy slučitelnost, s programy psanými pro starší verze jazyka BASIC), ale programátor může vytvářet i proměnné, které jsou lokální uvnitř funkcí, jejich označením klíčovým slovem — LOCAL.

Tcl

Asi si mohu dovolit tvrdit, že v Tcl neexistuje žádný mechanismus pro přístup k různým úrovním viditelnosti jmen. Důvodem je asi zvláštní způsob, jakým Tcl program provádí. Všechny proměnné se v každém případě jeví jako lokální vzhledem k jejich nejbližšímu okolí — proměnné na úrovni souboru jsou viditelné pouze pro příkazy uvnitř stejného souboru a proměnné procedur jsou viditelné pouze uvnitř procedur. Komunikaci mezi těmito prostory jmen můžeme zajistit pouze předáváním hodnot v podobě parametrů, při volání zmíněných procedur.

Objektově orientované programování

Co to vůbec je?

Nyní se pustíme do něčeho, co se do doby asi před pěti lety považovalo za náročné téma. V současnosti se již *objektově orientované programování* stalo normou. Jazyky, jako jsou Java a Python ztělesňují tento koncept do té míry, že se setkání s objekty nevyhnete již při programování jednoduchých věcí. Takže o čem vlastně objektové programování pojednává?

Podle mého názoru k nejlepším úvodům do problematiky patří:

- *Object Oriented Analysis* autorů Peter Coad & Ed Yourdon.
- *Object Oriented Analysis and Design with Applications* autora jménem Grady Booch (první vydání — pokud se vám je podaří sehnat).
- *Object Oriented Software Construction* autora jménem Bertrand Meyer (určitě chtějte druhé vydání).

Poznámka překladatele: Není mi známo, že by uvedená literatura byla přeložena do českého jazyka. Pokud je skutečnost jiná, dejte mi, prosím, vědět.

Knihy jsou uvedeny v pořadí rostoucí hloubky, velikosti a akademické exaktnosti. Většině neprofesionálních programátorů bude vyhovovat první kniha. Úvod, který je více zaměřen na programování, naleznete v *Object Oriented Programming* autora jménem Timothy Budd (druhé vydání). Osobně jsem tuto knihu nečetl, ale opěvují ji recenze lidí, jejichž názorů si vážím.

A konečně celou hromadu informací o všech možných tématech kolem objektově orientovaného programování (OOP) naleznete na webových stránkách http://www.cetus-links.org.

Protože předpokládám, že teď nemáte čas ani sklony k tomu, abyste zkoumali obsah všech uvedených knih a odkazů, předložím vám stručný přehled tohoto konceptu. (**Poznámka:** Některým lidem se koncept objektové orientace zdá těžce pochopitelný, jiným *sedne* hned. Pokud patříte k té první kategorii, netrapte se tím. Objekty můžete docela dobře používat i v případě, že vám jejich výhody nejsou zcela zřejmé.)

A ještě jedna poznámka na závěr. V této části budeme používat pouze Python, protože ani BASIC, ani Tcl objekty nepodporují. Při dodržování určitých konvencí zápisu kódu lze koncept objektově orientovaného návrhu využít i v jazycích, které nejsou objektově orientované, ale v takovém případě jde spíše jen o možné východisko z nouze než o doporučovanou strategii. Pokud je váš problém výhodně řešitelný technikami objektově orientovaného návrhu a programování, pak je vždy nejlepší, když použijete objektově orientovaný jazyk.

Dejme data a funkce dohromady

Objekty v sobě zahrnují nejen data ale i funkce, které nad uvedenými daty pracují. Data i funkce jsou svázány dohromady takovým způsobem, že objekt můžete předat z jedné části programu do druhé a obě části mohou přistupovat nejen k datovým *atributům*, ale přístupné jsou i *operace*.

Takže například objekt typu řetězec (string) poskytuje nejen prostor pro uložení znaků řetězce, ale poskytuje i *metody* pro provádění operací nad uloženým řetězcem — vyhledávání, změnu malých písmen na velká, určení délky řetězce a podobně.

V souvislosti s objekty se hovoří o komunikaci *zasíláním zpráv*. Jeden objekt zašle jinému objektu zprávu a přijímající objekt na ni zareaguje provedením jedné ze svých operací, takzvané *metody*. Takže říkáme, že metoda je vlastním objektem *vyvolána* při příjmu odpovídající zprávy. Způsob zápisu tohoto obratu bývá různý, ale nejběžnější z nich se snaží napodobit přístup ke složkám záznamu — používá tečkové notace. Takže pro třídu fiktivního prvku (widget[1]) můžeme psát:

```
w = Widget() # vytvoř novou instanci w třídy Widget()
w.paint()    # zašli mu zprávu 'paint' (tj. 'vykresli')
```

Tento zápis způsobí, že bude vyvolána metoda paint().

Poznámka překladatele: Připomeňme si znovu, že je to vnitřní funkce objektu. Zatímco v neobjektových jazycích (například Pascal, C) se tento zápis používal pouze pro zpřístupnění datových složek záznamu, u objektově orientovaných jazyků se používá jak pro zpřístupnění datových složek objektu (v tomto smyslu je objekt totéž, co v neobjektových jazycích záznam), tak pro zpřístupnění jeho vnitřních funkcí (říká se jim také členské funkce, protože jsou členy objektu nebo třídy). Ale nejpoužívanějším a obecně srozumitelným pojmem pro takové funkce je *metoda*. To, že se nejedná o datovou složku záznamu (nebo objektu) se v různých jazycích obvykle vyjadřuje tím, že zápis připomíná volání funkce. Typicky se za identifikátor metody zapisují kulaté závorky. V nich se mohou uvádět i požadované argumenty — jako u funkcí.

Definice tříd

Objekty mohou být různého typu ve stejném smyslu, jako mohou být i data různého typu. Množina objektů se shodnými charakteristikami je známa pod pojmem *třída*. Třídu můžeme nadefinovat a potom můžeme vytvářet *instance* této třídy, což jsou vlastně skutečné objekty. Odkazy (reference) na tyto objekty můžeme v našem programu ukládat do proměnných.

Podívejme se na konkrétní příklad a uvidíme, jestli se to podaří vysvětlit lépe. Vytvoříme třídu Zprava, která bude popisovat existenci datové složky typu řetězec — tj. textu zprávy — a metodu k vytištění (zobrazení) zprávy (vytisknout).

```
class Zprava:
def __init__(self, retezec):
self.text = retezec

def vytisknout(self):
print self.text
```

Poznámka 1: Jedna z metod této třídy se nazývá __init__. Jde o speciální metodu, které se říká *konstruktor*. Říká se jí tak, protože se volá v okamžiku vytváření (konstruování) nové instance objektu. Pokud uvnitř této metody nějaké proměnné něco přiřadíme (což v jazyce Python zajistí její vytvoření), pak bude tato proměnná patřit výhradně nové instanci. V jazyce Python existuje řada podobných metod. Téměř všechny podobné jsou od zbytku odlišeny tím, že používají speciální tvar jména __xxx__ (tedy dva znaky podtržení, slovo a opět dva znaky podtržení).

Poznámka 2: Obě uvedené metody používají první parametr self. (Při personifikaci objektu by se to dalo přeložit jako *já* nebo *já sám*.) Toto pojmenování je dáno pouze konvencí, ale vhodně vyjadřuje výskyt (existenci) objektu. Jak uvidíme později, tento parametr bude naplněn až za běhu, a to interpretem — nikoliv tedy programátorem. To jinými slovy znamená, že metoda vytiskni bude volána bez zadávání argumentů takto: m.vytiskni().

Poznámka překladatele: Teoreticky bychom tomuto parametru mohli přidělit jakékoliv jméno, ale řada pomocných nástrojů předpokládá dodržování této konvence. Navíc tomu všichni uživatelé jazyka Python rozumí na první pohled. Pokud nejde o pouhé pokusy, nepoužívejte jiné jméno prvního parametru. Dodržujte uvedenou konvenci i vy.

Poznámka 3: Uvedenou třídu jsme pojmenovali Zprava s velkým 'Z'. Jde opět pouze o konvenci, která se ovšem používá velmi často, a to nejen v jazyce Python, ale i v jiných objektově orientovaných jazycích. Daná konvence říká, že jména metod by měla začínat malým písmenem a další slova, ze kterých se jméno metody skládá, by měla začínat velkým písmenem. Takže například metody "vypočítej stav účtu" bychom zapsali:vypocitejStavUctu.

Poznámka překladatele: pojmy *třída* a *instance třídy* (tedy objekt) jsou velmi důležité. Bez jejich dokonalého pochopení budete v problematice objektově orientovaného programování jen tápat. Pokud se považujete za laiky, mohl by vám věc osvětlit výklad těchto pojmů napasovaný na Pohádku o Popelce[2].

V tomto okamžiku se možná budete chtít znovu podívat na kapitolu Data, datové typy a proměnné a zopakovat si *uživatelsky definované typy*. Příklad s adresou by měl být nyní o něco jasnější.

Třída je v jazyce Python v podstatě jediným uživatelsky definovaným typem. Třída, která má pouze atributy (datové složky), ale žádné metody (s výjimkou __init__), odpovídá záznamům v jazyce BASIC a v jiných neobjektových jazycích.

Používání tříd

Teď, když už máme definovánu třídu Zprava, můžeme vytvářet její instance a můžeme s nimi manipulovat:

```
z1 = Zprava("Ahoj lidi!")
z2 = Zprava("Sbohem. Bylo to krátké, ale sladké.")

poznamky = [z1, z2]  # vlož objekty do seznamu
for zpr in poznamky:
zpr.vytisknout() # každou zprávu vytiskni
```

Takže vidíme, že s třídou zacházíme, jako kdyby to byl standardní datový typ jazyka Python. A to byl vlastně cíl tohoto cvičení.

Když dva dělají totéž, není to totéž...

Zatím tedy umíme definovat své vlastní typy (třídy), umíme vytvářet jejich instance a umíme je přiřazovat do proměnných. Těmto objektům (instancím) pak můžeme zasílat zprávy, což způsobí provedení metod, které jsme definovali. Ale co se týká objektově orientovaného přísupu, je zde ještě jedna věc. Z mnoha pohledů jde o nejdůležitější vlastnost vůbec.

Mějme dva objekty různých tříd, které podporují stejnou množinu zpráv, ale definují své vlastní odpovídající metody. V takovém případě můžeme tyto objekty udržovat ve společné kolekci a v našem programu s nimi můžeme zacházet stejným způsobem. Objekty se však budou chovat každý jinak (po svém). Této schopnosti — chovat se jinak při zpracování stejné zprávy — se říká *polymorfismus* (doslova *mnohotvárnost*, ale nepřekládá se).

Typicky se toho využívá například při existenci několika různých grfických objektů, které se umí vykreslit, když obdrží zprávu 'paint'. Objekt kruhu vykreslí ve srovnání s objektem trojúhelníku velmi odlišný obrazec, ale pokud oba definují metodu paint, můžeme tento rozdíl jako programátoři ignorovat a můžeme o nich uvažovat jako o *tvarech*.

Podívejme se na příklad, kde místo vykreslování tvarů budeme vypočítávat jejich plochy. Nejdříve vytvoříme třídy Ctverec a Kruh:

```
class Ctverec:
def __init__(self, strana):
self.strana = strana
def vypocitejPlochu(self):
return self.strana**2

class Kruh:
def __init__(self, polomer):
self.polomer = polomer
def vypocitejPlochu(self):
import math
return math.pi*(self.polomer**2)
```

Nyní vytvoříme seznam tvarů (buď kruhů nebo čtverců) a poté vytiskneme jejich plochy:

```
seznam = [Kruh(5), Kruh(7), Ctverec(9), Kruh(3), Ctverec(12)]

for tvar in seznam:
print "Plocha je: ", tvar.vypocitejPlochu()
```

Pokud nyní zkombinujeme uvedené myšlenky s moduly, dostaneme velmi mocný mechanismus pro opakované použití kódu. Uložme definice tříd do modulu — řekněme tvary.py — a když potom budeme chtít manipulovat s tvary, jednoduše nejdříve provedeme import tohoto modulu. Přesně takto je do systému Python zařazena řada standardních modulů. To je důvod, proč se přístup k metodám objektu tolik podobá používání funkcí z modulu.

Dědičnost

Dědičnost se často používá jako mechanismus pro implementování polymorfismu. V mnoha objektově orientovaných jazycích je to ve skutečnosti jediný způsob pro implementování polymorfismu. Funguje to následovně.

Třída může z *rodičovské třídy* nebo také *nadtřídy* (super class) *dědit* jak atributy (datové prvky) tak operace. To znamená, že nová třída, která se ve většině věcí podobá jiné třídě, nemusí znovu implementovat všechny metody existující třídy. Místo toho může její schopnosti zdědit a *předefinovat* (override) jen to, co se má dělat jinak (například metodu pro vykreslování, o které jsme se zmínili ve výše uvedeném případě).

Nejlepší bude ukázat vše na příkladu. Vytvoříme *hierarchii tříd* pro bankovní účty, u kterých můžeme ukládat hotovost, zjišťovat stav účtu a realizovat výběr. Některé z účtů poskytují úroky (pro naše účely budeme předpokládat, že úrok je vypočítán při každém vkladu — zajímavá inovace pro bankovní svět) a u jiných se při výběru účtuje poplatek.

Třída BankovniUcet

Podívejme se, jak by to mohlo vypadat. Nejdříve uvažme atributy a operace bankovního účtu na nejobecnější (nebo *abstraktní*) úrovni.

Obvykle je nejlepší uvažovat nejdříve o operacích a teprve podle potřeby doplnit atributy tak, abychom mohli oprerace realizovat. Takže u bankovního účtu můžeme:

- Vkládat hotovost,
- vybírat hotovost,
- zjišťovat současný stav účtu a
- převádět peníze na jiný účet.

Pro tyto operace potřebujeme znát číslo bankovního účtu (pro operaci převodu) a současný stav účtu. Vytvořme odpovídající třídu:

```
class ChybaZustatku(Exception): pass

class BankovniUcet:
def __init__(self, pocatecniVklad):
self.stav = pocatecniVklad
print "Byl založen účet s počátečním stavem %5.2f korun." % self.stav

def vlozit(self, castka):
self.stav = self.stav + castka

def vybrat(self, castka):
if self.stav >= castka:
self.stav = self.stav - castka
else:
raise ChybaZustatku("Litujeme. Na vašem účtu je jen %6.2f korun."
% self.stav)

def zustatek(self):
return self.stav

def prevod(self, castka, ucet):
try:
self.vybrat(castka)
ucet.vlozit(castka)
except ChybaZustatku, e:
print str(e)
```

Poznámka 1: Před výběrem z účtu kontrolujeme stav účtu a pro ošetření chyb používáme výjimky. Chyba ChybaZustatku samozřejmě neexistuje, takže si ji musíme vytvořit. **Definujeme ji jako třídu, která je odvozena od zabudované třídy Exception. Ta je bázovou třídou všech zabudovaných výjimek systému Python a měla by se používat pro všechny uživatelsky definované výjimky. Při vytváření instance této třídy (příkazem raise) lze předat řetězec, který lze z objektu výjimky extrahovat zabudovanou funkcí str().**.

Poznámka překladatele: V novějších verzích jazyka Python se pro výjimky vždy doporučuje používat třídy, které odvodíme od bázové třídy Except. V roli instancí výjimek se již nedoporučuje používat řetězce.

Poznámka 2: Metoda prevod používá pro realizaci převodu *členské funkce* neboli metody vybrat a vlozit třídy BankovniUcet. Tento přístup je při objektově orientovaném programování velmi běžný a je znám jako *zasílání zpráv sobě samému* (self messaging). Znamená to, že *odvozené třídy* mohou implementovat své vlastní verze metod vlozit a vybrat, přičemž metoda prevod může u všech typů účtů zůstat stejná.

Třída UrocenyUcet

Teď za použití dědičnosti vytvoříme účet, na který budou připisována procenta (budeme předpokládat 3 %) při každém vkladu. Třída se bude shodovat s dříve uvedenou třídou BankovniUcet s výjimkou metody vlozit. Jednoduše ji předefinujeme (override):

```
class UrocenyUcet(BankovniUcet):
def vlozit(self, castka):
BankovniUcet.vlozit(self, castka)
self.stav = self.stav * 1.03
```

A je to! Začíná se ukazovat síla objektově orientovaného programování. Všechny ostatní metody byly zděděny z třídy BankovniUcet (tím, že jsme BankovniUcet uvedli do závorek za jméno nové třídy). Povšimněte si, že metoda vlozit místo kopírování kódu raději volá metodu vlozit své *nadtřídy* (superclass). Takže pokud nyní upravíme metodu vlozit třídy BankovniUcet například přidáním nějakých kontrol chyb, projeví se tyto změny automaticky i v *podtřídě*.

Poznámka překladatele: Místo pojmu *nadtřída* (super class) se často používá pojem *bázová třída* (base class) a místo pojmu *podtřída* (sub-class) se často používá pojem *odvozená třída* (derrived class).

Třída UcetSPoplatkem

Tento typ účtu se opět shoduje s třídou BankovniUcet pro standardní bankovní účet s tím rozdílem, že se tentokrát při každém výběru účtují tři koruny. Stejně jako v případě třídy UrocenyUcet můžeme novou třídu vytvořit děděním z třídy BankovniUcet a úpravou metody vybrat.

```
class UcetSPoplatkem(BankovniUcet):
def __init__(self, pocatecniVklad):
BankovniUcet.__init__(self, pocatecniVklad)
self.poplatek = 3

def vybrat(self, castka):
BankovniUcet.vybrat(self, castka + self.poplatek)
```

Poznámka 1: Velikost poplatku je uložena v *členské proměnné*, takže ji podle potřeby můžeme později měnit. Povšimněte si, že zděděnou metodu __init__ můžeme volat stejně jako každou jinou metodu.

Poznámka 2: Poplatek jednoduše přičítáme k požadované hodnotě výběru a provedení celé operace zajistíme voláním metody vybrat třídy BankovniUcet.

Poznámka 3: Vedlejším efektem tohoto postupu je to, že se poplatek uplatní i při převodu na jiný účet. Pravděpodobně to takto chceme, takže je to v pořádku.

Testujeme náš systém

Abychom si vyzkoušeli, že to všechno funguje, zkuste spustit následující kus kódu (buď z příkazového řádku systému Python nebo vytvořením souboru s těmito testy). Následující kód předpokládá, že jste výše uvedené definice tříd uložili do souboru bankovniucet.py. Pokud byste je uložili do jiného souboru, musíte změnit jméno modulu v prvním řádku souboru.

```
from bankovniucet import *
```

```
# Nejdříve vyzkoušíme standardní BankovniUcet.
a = BankovniUcet(500)
b = BankovniUcet(200)
a.vybrat(100)
# a.vybrat(1000)
a.prevod(100, b)
print "A = ", a.zustatek()
print "B = ", b.zustatek()

# Teď vyzkoušíme UrocenyUcet.
c = UrocenyUcet(1000)
c.vlozit(100)
print "C = ", c.zustatek()

# A ještě UcetSPoplatkem.
d = UcetSPoplatkem(300)
d.vlozit(200)
print "D = ", d.zustatek()
d.vybrat(50)
print "D = ", d.zustatek()
d.prevod(100, a)
print "A = ", a.zustatek()
print "D = ", d.zustatek()


# A nakonec provedeme převod z účtu s poplatky na úročený účet.
# Z účtu s poplatky by se měl odečíst i poplatek a na úročeném
# účtu by měl přibýt i úrok.
print "C = ", c.zustatek()
print "D = ", d.zustatek()
d.prevod(20, c)
print "C = ", c.zustatek()
print "D = ", d.zustatek()
```

Nyní odkomentujte řádek a.vybrat(1000) a uvidíte, jak zafunguje výjimka.

A je to. Jde o poměrně zjednodušený příklad, ale ukazuje, jak můžeme využít dědičnosti k rychlému rozšíření existující funkčnosti o nové rysy.

Ukázali jsme si, jak lze příklad vytvořit po etapách a jak lze vytvořit testovací program, kterým funkčnost řešení ověříme. Naše testy nebyly úplné v tom smyslu, že jsme nepokryli všechny možné případy. Mohli bychom přidat také další kontroly. Co kdyby byl například vytvořen účet se záporným zůstatkem…

Kolekce objektů

Jedna z otázek, která vás mohla napadnout, zní: *Jak bychom měli zacházet s větším množstvím objektů?* Nebo také: *Jak bychom měli zacházet s objekty, které vytvoříme za běhu programu?* Statické vytvoření objektů bankovních účtů — jak jsme to učnili výše — je velmi snadné:

```
ucet1 = BankovniUcet(...)
ucet2 = BankovniUcet(...)
ucet3 = BankovniUcet(...)
atd.
```

Ale u reálných řešení dopředu nevíme kolik účtů budeme chtít vytvořit. Jak si s tím poradíme? Uvažujeme nyní o problému trochu podrobněji.

Potřebujeme nějaký druh *databáze*, která by nám dovolila nalézt požadovaný bankovní účet podle jména vlastníka (nebo spíše podle čísla účtu, protože jedna osoba může mít více účtů a také více lidí může mít stejné jméno).

V kolekci potřebujeme něco vyhledat podle jednoznačného klíče… hmmm — to vypadá na použití slovníku! (Připomeňme si, že pro takto pojmenovanou strukturu jazyka Python lze použít i pojem *vyhledávací tabulka*.)Podívejme se, jak bychom použili strukturu typu slovník (dictionary) pro uložení dynamicky vytvořených objektů.

```
from bankovniucet import *
import time

# Funkce pro generování unikátních identifikačních čísel.
def ziskejDalsiID():
    ok = raw_input("Vytvořit účet [a/n]? ")
    if ok[0] in 'aA':       # v případě souhlasu...
        id = time.time()    # použij aktuální čas jako základ ID,
        id = int(id) % 10000 # převeď na max. 4místné celé číslo
        # Poznámka překladatele: time.time() vrací reálné číslo
        # odpovídající času v sekundách (s přesností obvykle
        # lepší než 1 sekunda).
    else: id = -1           # tato hodnota zastaví cyklus
    return id


tabulkaUctu = {}  # nový slovník
while 1:          # nekonečný cyklus
    id = ziskejDalsiID()
    if id == -1:
        break     # break násilně ukončí cyklus while
```

```
vklad = float(raw_input("Počáteční vklad? "))

# Identifikaci id použijeme pro vytvoření nové položky tabulky.
tabulkaUctu[id] = BankovniUcet(vklad)
print "Byl vytvořen nový účet číslo %04d s vkladem %0.2f" % (id, vklad)

# Zobrazíme si zůstatky na všech účtech.
for id in tabulkaUctu:
# Poznámka překladatele: od Python 2.0 není nutné
# v zápisu cyklu uvádět tabulkaUctu.keys(), jak tomu bylo
# u starších verzí.
print "%04d\t%0.2f" % (id, tabulkaUctu[id].zustatek())

# Nyní vyhledáme konkrétní účet. Pokud chcete ukončit
# program, vložte nečíselnou hodnotu.
while 1:
id = int(raw_input("Zadejte číslo účtu: "))
if id in tabulkaUctu:
print "Zůstatek = %0.2f" % tabulkaUctu[id].zustatek()
else: print "Chybné číslo účtu."
```

V roli klíče, který se používá pro vyhledávání ve slovníku, může být samozřejmě použito cokoliv, co jednoznačně identifikuje objekt. Může to být nějaký z jeho atributů, například jméno. Cokoliv, co je jednoznačné. Možná teď pro vás bude užitečné, když si znovu projdete kapitolu Data, datové typy a proměnné, a přečtete si konkrétně část, která se týká slovníků (vyhledávacích tabulek). Jsou to opravdu velmi užitečné kontejnery.

Ukládání vašich objektů

Výše uvedené řešení má jednu nevýhodu. Jakmile ukončíte program, všechny údaje budou ztraceny. Potřebujeme tedy nějaký způsob pro ukládání objektů. S vaším postupujícím programátorským růstem se později naučíte, jak pro tento účel používat databáze. Ale v tomto okamžiku nám bude pro ukládání a opětovné načítání objektů stačit textový soubor. Python definuje moduly (zvané Pickle a Shelve), které umí s objekty v tomto smyslu zacházet efektivněji. Ale ukažme si raději generický (tedy *obecně použitelný*) způsob, který by fungoval v libovolném programovacím jazyce. Technický termín pro schopnost uložení a opětovné obnovení stavu objektů se shodou okolností nazývá *persistence*. (Tento pojem se chápe jako termín a obvykle se nepřekládá. Z hlediska významu bychom jej ale mohli přeložit jako *schopnost přetrvat* — rozumí se uchovat svůj stav po dobu, kdy aplikace neběží.)

Generický (tedy *obecně použitelný*) způsob spočívá ve vytvoření metod save a restore v objektu nejvyšší úrovně (poznámka překladatele: autor má na mysli *bázovou třídu*) a předefinujeme je v každé odvozené třídě tak, že nejdříve zavolají zděděnou verzi a poté přidají své lokálně definované atributy. Poznámka překladatele: Tyto metody nebudeme překládat (save [sejv] = uložit; restore [ristór] = obnovit), protože se jim typicky dávají právě tato anglická jména.

```
class A:
def __init__(self, x, y):
self.x = x
self.y = y

def save(self, fn):
f = open(fn, "w")
# Poznámka překladatele: Od verze Python 2 by se
# měla dávat přednost zápisu f = file(fn, "w")
f.write(str(self.x) + '\n') # převeď na řetězec
f.write(str(self.y) + '\n')
return f  # do stejného souboru budou své hodnoty
# připisovat objekty odvozených tříd

def restore(self, fn):
f = open(fn)
self.x = int(f.readline()) # převeď zpět na původní typ
self.y = int(f.readline())
return f

class B(A):
def __init__(self, x, y, z):
A.__init__(self, x, y)
self.z = z

def save(self, fn):
f = A.save(self, fn)        # zavolej rodičovskou metodu
f.write(str(self.z) + '\n')  # přidej vlastní hodnoty
return f                    # pro případné další potomky

def restore(self, fn):
f = A.restore(self, fn)
self.z = int(f.readline())
return f

# Vytvoříme instance.
a = A(1, 2)
b = B(3, 4, 5)
```

```
# Uložíme instance.
a.save('a.txt').close() # nezapomeň uzavřít soubor
b.save('b.txt').close()

# Obnovíme instance.
newA = A(5, 6)
newA.restore('a.txt').close() # nezapomeň uzavřít soubor
newB = B(7, 8, 9)
newB.restore('b.txt').close()
print "A: ", newA.x, newA.y
print "B: ", newB.x, newB.y, newB.z
```

Poznámka: Vytisknou se hodnoty, které jsou obnoveny načtením ze souborů, nikoliv hodnoty, které jsme použili při vytváření instancí.

Klíčovým požadavkem je předefinování metod save a restore v každé třídě a také to, aby byla nejdříve volána rodičovská metoda (tj. metoda bázové třídy). V odvozené třídě se pak musíme postarat pouze o přidané atributy. Způsob, jakým atribut převedeme na řetězec a uložíme, závisí samozřejmě na nás, ale musí být umístěn na zvláštním řádku. Při obnovování jednoduše obrátíme postup, který jsme použili při ukládání.

Poznámka překladatele: V uvedeném příkladu se skrývá problém. Otevřený soubor by se měl vždy explicitně uzavřít. V tomto smyslu je uvedené řešení poměrně nedokonalé. Pokud se považujete za začátečníky, soustřeďte se především na hlavní myšlenku, kterou autor prezentuje. Nespoléhejte na správnost příkladu v detailech.

Při práci se soubory by se mělo používat nepsané pravidlo, které říká, že soubor by se měl uzavírat na stejné úrovni, kde se otevřel. Tím se obvykle vyhneme komplikacím s předáváním odpovědnosti za uzavření souboru do volaného kódu nebo naopak do volajícího kódu. Z tohoto pravidla vyplývá, že by se místo textového jména souboru měl metodám save() a restore() předávat již objekt otevřeného souboru.

Další problém spočívá v tom, že explicitně určujeme jméno souboru, do kterého se objekt ukládá. Pokud byste někdy takto vybudovanou třídu chtěli použít například v jiné aplikaci, mohli byste se setkat s komplikacemi.

V tomto místě pokládám za vhodné zmínit se o tom, že u odvozené třídy *musíme sami zajistit* uvnitř metody __init__() volání stejnojmenné metody bázové třídy, které jako první parametr předáváme self.

Doufám, že vám tato kapitola dala přičichnout k objektově orientovanému programování. Další informace a příklady můžete nalézt v nějaké dalším on-line učebnici nebo si můžete přečíst některou z knih, o kterých jsme se zmiňovali na začátku.

Prostory jmen

Úvod

Už slyším, jak se ptáte… Co to je ten *prostor jmen* (namespace)? No, dá se to těžko vysvětlit. Ne proto, že by to bylo nějak zvlášť komplikované, ale spíš proto, že se k tomu každý jazyk staví trochu jinak. Samotný koncept je docela přímočarý.

Prostor jmen je prostor nebo oblast uvnitř programu, kde je jméno (proměnné, třídy, atd.) platné.

Dřívější programovací jazyky (jako třeba BASIC) pracovaly pouze s *globálními proměnnými*, to znamená s takovými proměnnými, které byly vidět z celého programu — dokonce uvnitř funkcí. To činilo udržovatelnost programů velmi obtížnou, protože pro každý kousek programu bylo velmi snadné změnit nějakou proměnnou, aniž by se to ostatní části programu nějak dozvěděly. Tomuto jevu se říká *vedlejší efekt*. Novější jazyky (včetně moderních verzí jazyka BASIC) tento problém obcházejí zavedením prostorů jmen. (Jazyk C++ jde v tomto směru do extrému tím, že umožňuje programátorovi vytvořit svůj vlastní prostor jmen kdekoliv uvnitř programu. To ocení zvláště tvůrci knihoven, kteří chtějí dosáhnout jednoznačnosti jmen svých funkcí i v případě, kdy se současně použijí knihovny jiných tvůrců.)

Jak to řeší Python?

V systému Python vytváří každý modul svůj vlastní prostor jmen. Pokud chceme používat jména jeho částí, musíme jim buď předřadit jméno modulu, nebo musíme explicitně importovat požadovaná jména dovnitř prostoru jmen našeho modulu. Není to pro nás nic nového. Už jsme to dělali při práci s moduly sys a string. V určitém smyslu vytváří svůj prostor jmen i definice třídy. Takže pokud chceme zpřístupnit metodu nebo vlastnost třídy, musíme nejdříve použít jméno instance nebo třídy.

V Pythonu existují jen 3 prostory jmen (nebo *rozsahy platnosti* — scopes):

1. Lokální rozsah — jména definovaná uvnitř funkce nebo metody.
2. Rozsah v rámci modulu — jména definovaná uvnitř souboru modulu.
3. Zabudovaná jména — jména definovaná uvnitř samotného systému Python, která jsou přístupná vždy.

No dobrá. Ale jak to vše dáme dohromady, když proměnné v různých prostorech jmen mají stejné jméno? A nebo, jak se odkazujeme na jméno, které se nenachází v aktuálním prostoru jmen? Podívejme se nejdříve na první případ: Pokud se funkce odvolává na proměnnou nazvanou X a uvnitř funkce existuje nějaká proměnná X(tj. uvnitř prostoru s lokálními proměnnými), pak to bude právě tato lokální proměnná, kterou Python uvidí a použije. Je věcí programátora, aby se vyhnul střetům jmen, kdy má nějaká lokální proměnná stejné jméno jako proměnná modulu a kdy bychom mohli chtít zpřístupnit obě najednou. Existence lokální proměnné v takovém případě maskuje existenci globální proměnné.

Obecně bychom měli globální proměnné používat co nejméně. Obvykle bývá lepší, když běžnou, lokálníproměnnou předáváme jako parametr volané funkce a vrací se nám s modifikovaným obsahem.

Poznámka překladatele ke globálním proměnným: Z pohledu začátečníka se může zdát používání globálních proměnných velmi výhodné. Jednoduše přece uvedeme jméno proměnné, které je známé ve všech místech programu! V čem je problém? Postupně zjistíte, že těch problémů může být hned několik. Zdánlivá jednoduchost může později věci zkomplikovat:

- Při zvětšování programu používáme čím dál víc jmen a začnou se nám špatně vymýšlet. Pokud chceme jméno proměnné později změnit, musíme je měnit na mnoha různých místech. Pro vhodný textový editor to nemusí být velký problém, ale...
- Pokud jazyk umožňuje práci s lokálními proměnnými, pak obvykle existence lokální proměnné zamaskuje existenci stejnojmenné globální proměnné — platí to i pro Python. Při čtení zdrojového textu si vůbec nemusíme všimnout toho, že je stejnojmenná lokální proměnná již definována. A najednou se program začne chovat jinak, než bychom čekali.
- Použitím jména globální proměnné někde v kódu vytváříme obtížně kontrolovatelnou vazbu na zbytek systému. Nejde jen o to, že existuje vazba na globálně pojmenovanou proměnnou, ale jejím prostřednictvím se najednou ovlivňuje činnost všech ostatních částí kódu, které s globální proměnnou pracují. Jinými slovy, někdy je velmi obtížné zajistit, aby

v globální proměnné byla uložena právě ta hodnota, která tam má být uložena. Zdánlivě nevinná změna obsahu, provedená v jednom místě, může neočekávaným způsobem ovlivnit činnost jiných částí aplikace.

Druhý případ, kdy se odkazujeme na jméno, které se nenachází mezi lokálními, se řeší následujícím způsobem: Funkce prohlédne svůj lokální prostor. Pokud zde požadované jméno nenalezne, hledá v prostoru modulu. A pokud není nalezeno ani zde, hledá se v prostoru zabudovaných jmen (builtin scope). Jediná nepříjemnost nastane v situaci, kdy chceme přiřadit hodnotu externí proměnné. Při normálním postupu by vznikla nová proměnná tohoto jména, ale tomu se chceme vyhnout. Takže aby se nevytvořila lokální proměnná daného jména, musíme určit, že se jedná o jméno globální.

Vše si ukážeme v akci na následujícím příkladu (jde o čistě ilustrační příklad):

```
# Proměnné na úrovni modulu.
W = 5
Y = 3

# Parametry se chovají jako proměnné náležející funkci. Takže X patří
# do lokálního prostoru.
def spam(X):

# Funkci oznámíme, že má W hledat na úrovni modulu a nevytvářet svou
# proměnnou W.
global W

Z = X*2 # Nová proměnná Z je vytvořena jako lokální.
W = X+5 # Práce s W na úrovni modulu -- viz výše.

if Z > W:
# pow je jméno 'zabudované' funkce.
print pow(Z, W)
return Z
else:
return Y # Lokální Y neexistuje, takže se použije globální.
```

Pokud importujeme modul, jako je například sys, stane se jméno sys lokálně dostupným jménem. Poté můžeme jména uvnitř prostoru jmen modulu sys zpřístupnit použitím takzvaného *kvalifikovaného jména*, jak jsme si ukázali dříve. (Kvalifikované jméno se od holého liší tím, že holému jménu předřadíme takzvaný kvalifikátor, který má podobu dalšího jména, vhodně spojeného s původním holým jménem. V jazyce Python se obě části oddělují tečkou. Například v jazyce C++ se oddělují dvěma dvojtečkami.)

Pokud napíšeme

```
from sys import exit
```

pak v lokálním prostoru jmen zpřístupníme pouze funkci exit. Nemůžeme použít žádné jiné jméno z modulu sys a dokonce ani samotné jméno modulu sys.

Ještě v jazyce BASIC...

BASIC volí ve srovnání z jazykem Python opačný přístup. Všechny vytvořené proměnné se automaticky stávají globálními (aby byla zachována kompatibilita, tedy slučitelnost, s programy psanými pro starší verze jazyka BASIC), ale programátor může vytvářet i proměnné, které jsou lokální uvnitř funkcí, jejich označením klíčovým slovem — LOCAL.

Tcl

Asi si mohu dovolit tvrdit, že v Tcl neexistuje žádný mechanismus pro přístup k různým úrovním viditelnosti jmen. Důvodem je asi zvláštní způsob, jakým Tcl program provádí. Všechny proměnné se v každém případě jeví jako lokální vzhledem k jejich nejbližšímu okolí — proměnné na úrovni souboru jsou viditelné pouze pro příkazy uvnitř stejného souboru a proměnné procedur jsou viditelné pouze uvnitř procedur. Komunikaci mezi těmito prostory jmen můžeme zajistit pouze předáváním hodnot v podobě parametrů, při volání zmíněných procedur.

Objektově orientované programování

Co to vůbec je?

Nyní se pustíme do něčeho, co se do doby asi před pěti lety považovalo za náročné téma. V současnosti se již *objektově orientované programování* stalo normou. Jazyky, jako jsou Java a Python ztělesňují tento koncept do té míry, že se setkání s objekty nevyhnete již při programování jednoduchých věcí. Takže o čem vlastně objektové programování pojednává?

Podle mého názoru k nejlepším úvodům do problematiky patří:

- *Object Oriented Analysis* autorů Peter Coad & Ed Yourdon.
- *Object Oriented Analysis and Design with Applications* autora jménem Grady Booch (první vydání — pokud se vám je podaří sehnat).
- *Object Oriented Software Construction* autora jménem Bertrand Meyer (určitě chtějte druhé vydání).

Poznámka překladatele: Není mi známo, že by uvedená literatura byla přeložena do českého jazyka. Pokud je skutečnost jiná, dejte mi, prosím, vědět.

Knihy jsou uvedeny v pořadí rostoucí hloubky, velikosti a akademické exaktnosti. Většině neprofesionálních programátorů bude vyhovovat první kniha. Úvod, který je více zaměřen na programování, naleznete v *Object Oriented Programming* autora jménem Timothy Budd (druhé vydání). Osobně jsem tuto knihu nečetl, ale opěvují ji recenze lidí, jejichž názorů si vážím.

A konečně celou hromadu informací o všech možných tématech kolem objektově orientovaného programování (OOP) naleznete na webových stránkách http://www.cetus-links.org.

Protože předpokládám, že teď nemáte čas ani sklony k tomu, abyste zkoumali obsah všech uvedených knih a odkazů, předložím vám stručný přehled tohoto konceptu. (**Poznámka:** Některým lidem se koncept objektové orientace zdá těžce pochopitelný, jiným *sedne* hned. Pokud patříte k té první kategorii, netrapte se tím. Objekty můžete docela dobře používat i v případě, že vám jejich výhody nejsou zcela zřejmé.)

A ještě jedna poznámka na závěr. V této části budeme používat pouze Python, protože ani BASIC, ani Tcl objekty nepodporují. Při dodržování určitých konvencí zápisu kódu lze koncept objektově orientovaného návrhu využít i v jazycích, které nejsou objektově orientované, ale v takovém případě jde spíše jen o možné východisko z nouze než o doporučovanou strategii. Pokud je váš problém výhodně řešitelný technikami objektově orientovaného návrhu a programování, pak je vždy nejlepší, když použijete objektově orientovaný jazyk.

Dejme data a funkce dohromady

Objekty v sobě zahrnují nejen data ale i funkce, které nad uvedenými daty pracují. Data i funkce jsou svázány dohromady takovým způsobem, že objekt můžete předat z jedné části programu do druhé a obě části mohou přistupovat nejen k datovým *atributům*, ale přístupné jsou i *operace*.

Takže například objekt typu řetězec (string) poskytuje nejen prostor pro uložení znaků řetězce, ale poskytuje i *metody* pro provádění operací nad uloženým řetězcem — vyhledávání, změnu malých písmen na velká, určení délky řetězce a podobně.

V souvislosti s objekty se hovoří o komunikaci *zasíláním zpráv*. Jeden objekt zašle jinému objektu zprávu a přijímající objekt na ni zareaguje provedením jedné ze svých operací, takzvané *metody*. Takže říkáme, že metoda je vlastním objektem *vyvolána* při příjmu odpovídající zprávy. Způsob zápisu tohoto obratu bývá různý, ale nejběžnější z nich se snaží napodobit přístup ke složkám záznamu — používá tečkové notace. Takže pro třídu fiktivního prvku (widget[1]) můžeme psát:

```
w = Widget() # vytvoř novou instanci w třídy Widget()
w.paint()    # zašli mu zprávu 'paint' (tj. 'vykresli')
```

Tento zápis způsobí, že bude vyvolána metoda paint().

Poznámka překladatele: Připomeňme si znovu, že je to vnitřní funkce objektu. Zatímco v neobjektových jazycích (například Pascal, C) se tento zápis používal pouze pro zpřístupnění datových složek záznamu, u objektově orientovaných jazyků se používá jak pro zpřístupnění datových složek objektu (v tomto smyslu je objekt totéž, co v neobjektových jazycích záznam), tak pro zpřístupnění jeho vnitřních funkcí (říká se jim také členské funkce, protože jsou členy objektu nebo třídy). Ale nejpoužívanějším a obecně srozumitelným pojmem pro takové funkce je *metoda*. To, že se nejedná o datovou složku záznamu (nebo objektu) se v různých jazycích obvykle vyjadřuje tím, že zápis připomíná volání funkce. Typicky se za identifikátor metody zapisují kulaté závorky. V nich se mohou uvádět i požadované argumenty — jako u funkcí.

Definice tříd

Objekty mohou být různého typu ve stejném smyslu, jako mohou být i data různého typu. Množina objektů se shodnými charakteristikami je známa pod pojmem *třída*. Třídu můžeme nadefinovat a potom můžeme vytvářet *instance* této třídy, což jsou vlastně skutečné objekty. Odkazy (reference) na tyto objekty můžeme v našem programu ukládat do proměnných.

Podívejme se na konkrétní příklad a uvidíme, jestli se to podaří vysvětlit lépe. Vytvoříme třídu Zprava, která bude popisovat existenci datové složky typu řetězec — tj. textu zprávy — a metodu k vytištění (zobrazení) zprávy (vytisknout).

```
class Zprava:
    def __init__(self, retezec):
        self.text = retezec

    def vytisknout(self):
        print self.text
```

Poznámka 1: Jedna z metod této třídy se nazývá __init__. Jde o speciální metodu, které se říká *konstruktor*. Říká se jí tak, protože se volá v okamžiku vytváření (konstruování) nové instance objektu. Pokud uvnitř této metody nějaké proměnné něco přiřadíme (což v jazyce Python zajistí její vytvoření), pak bude tato proměnná patřit výhradně nové instanci. V jazyce Python existuje řada podobných metod. Téměř všechny podobné jsou od zbytku odlišeny tím, že používají speciální tvar jména __xxx__ (tedy dva znaky podtržení, slovo a opět dva znaky podtržení).

Poznámka 2: Obě uvedené metody používají první parametr self. (Při personifikaci objektu by se to dalo přeložit jako *já* nebo *já sám*.) Toto pojmenování je dáno pouze konvencí, ale vhodně vyjadřuje výskyt (existenci) objektu. Jak uvidíme později, tento parametr bude naplněn až za běhu, a to interpretem — nikoliv tedy programátorem. To jinými slovy znamená, že metoda vytiskni bude volána bez zadávání argumentů takto: m.vytiskni().

Poznámka překladatele: Teoreticky bychom tomuto parametru mohli přidělit jakékoliv jméno, ale řada pomocných nástrojů předpokládá dodržování této konvence. Navíc tomu všichni uživatelé jazyka Python rozumí na první pohled. Pokud nejde o pouhé pokusy, nepoužívejte jiné jméno prvního parametru. Dodržujte uvedenou konvenci i vy.

Poznámka 3: Uvedenou třídu jsme pojmenovali Zprava s velkým 'Z'. Jde opět pouze o konvenci, která se ovšem používá velmi často, a to nejen v jazyce Python, ale i v jiných objektově orientovaných jazycích. Daná konvence říká, že jména metod by měla začínat malým písmenem a další slova, ze kterých se jméno metody skládá, by měla začínat velkým písmenem. Takže například metody "vypočítej stav účtu" bychom zapsali:vypocitejStavUctu.

Poznámka překladatele: pojmy *třída* a *instance třídy* (tedy objekt) jsou velmi důležité. Bez jejich dokonalého pochopení budete v problematice objektově orientovaného programování jen tápat. Pokud se považujete za laiky, mohl by vám věc osvětlit výklad těchto pojmů napasovaný na Pohádku o Popelce[2].

V tomto okamžiku se možná budete chtít znovu podívat na kapitolu Data, datové typy a proměnné a zopakovat si *uživatelsky definované typy*. Příklad s adresou by měl být nyní o něco jasnější.

Třída je v jazyce Python v podstatě jediným uživatelsky definovaným typem. Třída, která má pouze atributy (datové složky), ale žádné metody (s výjimkou __init__), odpovídá záznamům v jazyce BASIC a v jiných neobjektových jazycích.

Používání tříd

Teď, když už máme definovánu třídu Zprava, můžeme vytvářet její instance a můžeme s nimi manipulovat:

```
z1 = Zprava("Ahoj lidi!")
z2 = Zprava("Sbohem. Bylo to krátké, ale sladké.")

poznamky = [z1, z2]  # vlož objekty do seznamu
for zpr in poznamky:
    zpr.vytisknout() # každou zprávu vytiskni
```

Takže vidíme, že s třídou zacházíme, jako kdyby to byl standardní datový typ jazyka Python. A to byl vlastně cíl tohoto cvičení.

Když dva dělají totéž, není to totéž...

Zatím tedy umíme definovat své vlastní typy (třídy), umíme vytvářet jejich instance a umíme je přiřazovat do proměnných. Těmto objektům (instancím) pak můžeme zasílat zprávy, což způsobí provedení metod, které jsme definovali. Ale co se týká objektově orientovaného přísupu, je zde ještě jedna věc. Z mnoha pohledů jde o nejdůležitější vlastnost vůbec.

Mějme dva objekty různých tříd, které podporují stejnou množinu zpráv, ale definují své vlastní odpovídající metody. V takovém případě můžeme tyto objekty udržovat ve společné kolekci a v našem programu s nimi můžeme zacházet stejným

způsobem. Objekty se však budou chovat každý jinak (po svém). Této schopnosti — chovat se jinak při zpracování stejné zprávy — se říká *polymorfismus* (doslova *mnohotvárnost*, ale nepřekládá se).
Typicky se toho využívá například při existenci několika různých grfických objektů, které se umí vykreslit, když obdrží zprávu 'paint'. Objekt kruhu vykreslí ve srovnání s objektem trojúhelníku velmi odlišný obrazec, ale pokud oba definují metodu paint, můžeme tento rozdíl jako programátoři ignorovat a můžeme o nich uvažovat jako o *tvarech*.
Podívejme se na příklad, kde místo vykreslování tvarů budeme vypočítávat jejich plochy. Nejdříve vytvoříme třídy Ctverec a Kruh:

```
class Ctverec:
def __init__(self, strana):
self.strana = strana
def vypocitejPlochu(self):
return self.strana**2

class Kruh:
def __init__(self, polomer):
self.polomer = polomer
def vypocitejPlochu(self):
import math
return math.pi*(self.polomer**2)
```

Nyní vytvoříme seznam tvarů (buď kruhů nebo čtverců) a poté vytiskneme jejich plochy:

```
seznam = [Kruh(5), Kruh(7), Ctverec(9), Kruh(3), Ctverec(12)]

for tvar in seznam:
print "Plocha je: ", tvar.vypocitejPlochu()
```

Pokud nyní zkombinujeme uvedené myšlenky s moduly, dostaneme velmi mocný mechanismus pro opakované použití kódu. Uložme definice tříd do modulu — řekněme tvary.py — a když potom budeme chtít manipulovat s tvary, jednoduše nejdříve provedeme import tohoto modulu. Přesně takto je do systému Python zařazena řada standardních modulů. To je důvod, proč se přístup k metodám objektu tolik podobá používání funkcí z modulu.

Dědičnost

Dědičnost se často používá jako mechanismus pro implementování polymorfismu. V mnoha objektově orientovaných jazycích je to ve skutečnosti jediný způsob pro implementování polymorfismu. Funguje to následovně.
Třída může z *rodičovské třídy* nebo také *nadtřídy* (super class) *dědit* jak atributy (datové prvky) tak operace. To znamená, že nová třída, která se ve většině věcí podobá jiné třídě, nemusí znovu implementovat všechny metody existující třídy. Místo toho může její schopnosti zdědit a *předefinovat* (override) jen to, co se má dělat jinak (například metodu pro vykreslování, o které jsme se zmínili ve výše uvedeném případě).
Nejlepší bude ukázat vše na příkladu. Vytvoříme *hierarchii tříd* pro bankovní účty, u kterých můžeme ukládat hotovost, zjišťovat stav účtu a realizovat výběr. Některé z účtů poskytují úroky (pro naše účely budeme předpokládat, že úrok je vypočítán při každém vkladu — zajímavá inovace pro bankovní svět) a u jiných se při výběru účtuje poplatek.

Třída BankovniUcet

Podívejme se, jak by to mohlo vypadat. Nejdříve uvažme atributy a operace bankovního účtu na nejobecnější (nebo *abstraktní*) úrovni.
Obvykle je nejlepší uvažovat nejdříve o operacích a teprve podle potřeby doplnit atributy tak, abychom mohli oprerace realizovat. Takže u bankovního účtu můžeme:

- Vkládat hotovost,
- vybírat hotovost,
- zjišťovat současný stav účtu a
- převádět peníze na jiný účet.

Pro tyto operace potřebujeme znát číslo bankovního účtu (pro operaci převodu) a současný stav účtu. Vytvořme odpovídající třídu:

```
class ChybaZustatku(Exception): pass

class BankovniUcet:
def __init__(self, pocatecniVklad):
self.stav = pocatecniVklad
print "Byl založen účet s počátečním stavem %5.2f korun." % self.stav

def vlozit(self, castka):
self.stav = self.stav + castka

def vybrat(self, castka):
if self.stav >= castka:
self.stav = self.stav - castka
else:
raise ChybaZustatku("Litujeme. Na vašem účtu je jen %6.2f korun."
% self.stav)

def zustatek(self):
return self.stav

def prevod(self, castka, ucet):
try:
self.vybrat(castka)
ucet.vlozit(castka)
except ChybaZustatku, e:
print str(e)
```

Poznámka 1: Před výběrem z účtu kontrolujeme stav účtu a pro ošetření chyb používáme výjimky. Chyba ChybaZustatku samozřejmě neexistuje, takže si ji musíme vytvořit. **Definujeme ji jako třídu, která je odvozena od zabudované třídy Exception. Ta je bázovou třídou všech zabudovaných výjimek systému Python a měla by se používat pro všechny uživatelsky definované výjimky. Při vytváření instance této třídy (příkazem raise) lze předat řetězec, který lze z objektu výjimky extrahovat zabudovanou funkcí str().**.

Poznámka překladatele: V novějších verzích jazyka Python se pro výjimky vždy doporučuje používat třídy, které odvodíme od bázové třídy Except. V roli instancí výjimek se již nedoporučuje používat řetězce.

Poznámka 2: Metoda prevod používá pro realizaci převodu *členské funkce* neboli metody vybrat a vlozit třídy BankovniUcet. Tento přístup je při objektově orientovaném programování velmi běžný a je znám jako *zasílání zpráv sobě samému* (self messaging). Znamená to, že *odvozené třídy* mohou implementovat své vlastní verze metod vlozit a vybrat, přičemž metoda prevod může u všech typů účtů zůstat stejná.

Třída UrocenyUcet

Teď za použití dědičnosti vytvoříme účet, na který budou připisována procenta (budeme předpokládat 3 %) při každém vkladu. Třída se bude shodovat s dříve uvedenou třídou BankovniUcet s výjimkou metody vlozit. Jednoduše ji předefinujeme (override):

```
class UrocenyUcet(BankovniUcet):
    def vlozit(self, castka):
        BankovniUcet.vlozit(self, castka)
        self.stav = self.stav * 1.03
```

A je to! Začíná se ukazovat síla objektově orientovaného programování. Všechny ostatní metody byly zděděny z třídy BankovniUcet (tím, že jsme BankovniUcet uvedli do závorek za jméno nové třídy). Povšimněte si, že metoda vlozit místo kopírování kódu raději volá metodu vlozit své *nadtřídy* (superclass). Takže pokud nyní upravíme metodu vlozit třídy BankovniUcet například přidáním nějakých kontrol chyb, projeví se tyto změny automaticky i v *podtřídě*.

Poznámka překladatele: Místo pojmu *nadtřída* (super class) se často používá pojem *bázová třída* (base class) a místo pojmu *podtřída* (sub-class) se často používá pojem *odvozená třída* (derrived class).

Třída UcetSPoplatkem

Tento typ účtu se opět shoduje s třídou BankovniUcet pro standardní bankovní účet s tím rozdílem, že se tentokrát při každém výběru účtují tři koruny. Stejně jako v případě třídy UrocenyUcet můžeme novou třídu vytvořit děděním z třídy BankovniUcet a úpravou metody vybrat.

```
class UcetSPoplatkem(BankovniUcet):
    def __init__(self, pocatecniVklad):
        BankovniUcet.__init__(self, pocatecniVklad)
        self.poplatek = 3

    def vybrat(self, castka):
        BankovniUcet.vybrat(self, castka + self.poplatek)
```

Poznámka 1: Velikost poplatku je uložena v *členské proměnné*, takže ji podle potřeby můžeme později měnit. Povšimněte si, že zděděnou metodu __init__ můžeme volat stejně jako každou jinou metodu.

Poznámka 2: Poplatek jednoduše přičítáme k požadované hodnotě výběru a provedení celé operace zajistíme voláním metody vybrat třídy BankovniUcet.

Poznámka 3: Vedlejším efektem tohoto postupu je to, že se poplatek uplatní i při převodu na jiný účet. Pravděpodobně to takto chceme, takže je to v pořádku.

Testujeme náš systém

Abychom si vyzkoušeli, že to všechno funguje, zkuste spustit následující kus kódu (buď z příkazového řádku systému Python nebo vytvořením souboru s těmito testy). Následující kód předpokládá, že jste výše uvedené definice tříd uložili do souboru bankovniucet.py. Pokud byste je uložili do jiného souboru, musíte změnit jméno modulu v prvním řádku souboru.

```
from bankovniucet import *

# Nejdříve vyzkoušíme standardní BankovniUcet.
a = BankovniUcet(500)
b = BankovniUcet(200)
a.vybrat(100)
# a.vybrat(1000)
a.prevod(100, b)
print "A = ", a.zustatek()
print "B = ", b.zustatek()

# Teď vyzkoušíme UrocenyUcet.
c = UrocenyUcet(1000)
c.vlozit(100)
print "C = ", c.zustatek()

# A ještě UcetSPoplatkem.
d = UcetSPoplatkem(300)
d.vlozit(200)
print "D = ", d.zustatek()
d.vybrat(50)
print "D = ", d.zustatek()
d.prevod(100, a)
print "A = ", a.zustatek()
print "D = ", d.zustatek()


# A nakonec provedeme převod z účtu s poplatky na úročený účet.
# Z účtu s poplatky by se měl odečíst i poplatek a na úročeném
```

```
# účtu by měl přibýt i úrok.
print "C = ", c.zustatek()
print "D = ", d.zustatek()
d.prevod(20, c)
print "C = ", c.zustatek()
print "D = ", d.zustatek()
```

Nyní odkomentujte řádek a.vybrat(1000) a uvidíte, jak zafunguje výjimka.

A je to. Jde o poměrně zjednodušený příklad, ale ukazuje, jak můžeme využít dědičnosti k rychlému rozšíření existující funkčnosti o nové rysy.

Ukázali jsme si, jak lze příklad vytvořit po etapách a jak lze vytvořit testovací program, kterým funkčnost řešení ověříme. Naše testy nebyly úplné v tom smyslu, že jsme nepokryli všechny možné případy. Mohli bychom přidat také další kontroly. Co kdyby byl například vytvořen účet se záporným zůstatkem…

Kolekce objektů

Jedna z otázek, která vás mohla napadnout, zní: *Jak bychom měli zacházet s větším množstvím objektů?* Nebo také: *Jak bychom měli zacházet s objekty, které vytvoříme za běhu programu?* Statické vytvoření objektů bankovních účtů — jak jsme to učnili výše — je velmi snadné:

```
ucet1 = BankovniUcet(...)
ucet2 = BankovniUcet(...)
ucet3 = BankovniUcet(...)
atd.
```

Ale u reálných řešení dopředu nevíme kolik účtů budeme chtít vytvořit. Jak si s tím poradíme? Uvažujeme nyní o problému trochu podrobněji.

Potřebujeme nějaký druh *databáze*, která by nám dovolila nalézt požadovaný bankovní účet podle jména vlastníka (nebo spíše podle čísla účtu, protože jedna osoba může mít více účtů a také více lidí může mít stejné jméno).

V kolekci potřebujeme něco vyhledat podle jednoznačného klíče… hmmm — to vypadá na použití slovníku! (Připomeňme si, že pro takto pojmenovanou strukturu jazyka Python lze použít i pojem *vyhledávací tabulka*.)Podívejme se, jak bychom použili strukturu typu slovník (dictionary) pro uložení dynamicky vytvořených objektů.

```
from bankovniucet import *
import time

# Funkce pro generování unikátních identifikačních čísel.
def ziskejDalsiID():
    ok = raw_input("Vytvořit účet [a/n]? ")
    if ok[0] in 'aA':       # v případě souhlasu...
        id = time.time()    # použij aktuální čas jako základ ID,
        id = int(id) % 10000 # převeď na max. 4místné celé číslo
        # Poznámka překladatele: time.time() vrací reálné číslo
        # odpovídající času v sekundách (s přesností obvykle
        # lepší než 1 sekunda).
    else: id = -1           # tato hodnota zastaví cyklus
    return id


tabulkaUctu = {}  # nový slovník
while 1:          # nekonečný cyklus
    id = ziskejDalsiID()
    if id == -1:
        break     # break násilně ukončí cyklus while
    vklad = float(raw_input("Počáteční vklad? "))

    # Identifikaci id použijeme pro vytvoření nové položky tabulky.
    tabulkaUctu[id] = BankovniUcet(vklad)
    print "Byl vytvořen nový účet číslo %04d s vkladem %0.2f" % (id, vklad)

# Zobrazíme si zůstatky na všech účtech.
for id in tabulkaUctu:
    # Poznámka překladatele: od Python 2.0 není nutné
    # v zápisu cyklu uvádět tabulkaUctu.keys(), jak tomu bylo
    # u starších verzí.
    print "%04d\t%0.2f" % (id, tabulkaUctu[id].zustatek())

# Nyní vyhledáme konkrétní účet. Pokud chcete ukončit
# program, vložte nečíselnou hodnotu.
while 1:
    id = int(raw_input("Zadejte číslo účtu: "))
    if id in tabulkaUctu:
        print "Zůstatek = %0.2f" % tabulkaUctu[id].zustatek()
    else: print "Chybné číslo účtu."
```

V roli klíče, který se používá pro vyhledávání ve slovníku, může být samozřejmě použito cokoliv, co jednoznačně identifikuje objekt. Může to být nějaký z jeho atributů, například jméno. Cokoliv, co je jednoznačné. Možná teď pro vás bude užitečné, když si znovu projdete kapitolu Data, datové typy a proměnné, a přečtete si konkrétně část, která se týká slovníků (vyhledávacích tabulek). Jsou to opravdu velmi užitečné kontejnery.

Ukládání vašich objektů

Výše uvedené řešení má jednu nevýhodu. Jakmile ukončíte program, všechny údaje budou ztraceny. Potřebujeme tedy nějaký způsob pro ukládání objektů. S vaším postupujícím programátorským růstem se později naučíte, jak pro tento účel používat databáze. Ale v tomto okamžiku nám bude pro ukládání a opětovné načítání objektů stačit textový soubor. Python

definuje moduly (zvané Pickle a Shelve), které umí s objekty v tomto smyslu zacházet efektivněji. Ale ukažme si raději generický (tedy *obecně použitelný*) způsob, který by fungoval v libovolném programovacím jazyce. Technický termín pro schopnost uložení a opětovné obnovení stavu objektů se shodou okolností nazývá *persistence*. (Tento pojem se chápe jako termín a obvykle se nepřekládá. Z hlediska významu bychom jej ale mohli přeložit jako *schopnost přetrvat* — rozumí se uchovat svůj stav po dobu, kdy aplikace neběží.)

Generický (tedy *obecně použitelný*) způsob spočívá ve vytvoření metod save a restore v objektu nejvyšší úrovně (poznámka překladatele: autor má na mysli *bázovou třídu*) a předefinujeme je v každé odvozené třídě tak, že nejdříve zavolají zděděnou verzi a poté přidají své lokálně definované atributy. Poznámka překladatele: Tyto metody nebudeme překládat (save [sejv] = uložit; restore [ristór] = obnovit), protože se jim typicky dávají právě tato anglická jména.

```
class A:
def __init__(self, x, y):
self.x = x
self.y = y

def save(self, fn):
f = open(fn, "w")
# Poznámka překladatele: Od verze Python 2 by se
# měla dávat přednost zápisu f = file(fn, "w")
f.write(str(self.x) + '\n') # převeď na řetězec
f.write(str(self.y) + '\n')
return f  # do stejného souboru budou své hodnoty
# připisovat objekty odvozených tříd

def restore(self, fn):
f = open(fn)
self.x = int(f.readline()) # převeď zpět na původní typ
self.y = int(f.readline())
return f

class B(A):
def __init__(self, x, y, z):
A.__init__(self, x, y)
self.z = z

def save(self, fn):
f = A.save(self, fn)        # zavolej rodičovskou metodu
f.write(str(self.z) + '\n')  # přidej vlastní hodnoty
return f                    # pro případné další potomky

def restore(self, fn):
f = A.restore(self, fn)
self.z = int(f.readline())
return f

# Vytvoříme instance.
a = A(1, 2)
b = B(3, 4, 5)

# Uložíme instance.
a.save('a.txt').close() # nezapomeň uzavřít soubor
b.save('b.txt').close()

# Obnovíme instance.
newA = A(5, 6)
newA.restore('a.txt').close() # nezapomeň uzavřít soubor
newB = B(7, 8, 9)
newB.restore('b.txt').close()
print "A: ", newA.x, newA.y
print "B: ", newB.x, newB.y, newB.z
```

Poznámka: Vytisknou se hodnoty, které jsou obnoveny načtením ze souborů, nikoliv hodnoty, které jsme použili při vytváření instancí.

Klíčovým požadavkem je předefinování metod save a restore v každé třídě a také to, aby byla nejdříve volána rodičovská metoda (tj. metoda bázové třídy). V odvozené třídě se pak musíme postarat pouze o přidané atributy. Způsob, jakým atribut převedeme na řetězec a uložíme, závisí samozřejmě na nás, ale musí být umístěn na zvláštním řádku. Při obnovování jednoduše obrátíme postup, který jsme použili při ukládání.

Poznámka překladatele: V uvedeném příkladu se skrývá problém. Otevřený soubor by se měl vždy explicitně uzavřít. V tomto smyslu je uvedené řešení poměrně nedokonalé. Pokud se považujete za začátečníky, soustřeďte se především na hlavní myšlenku, kterou autor prezentuje. Nespoléhejte na správnost příkladu v detailech.

Při práci se soubory by se mělo používat nepsané pravidlo, které říká, že soubor by se měl uzavírat na stejné úrovni, kde se otevřel. Tím se obvykle vyhneme komplikacím s předáváním odpovědnosti za uzavření souboru do volaného kódu nebo naopak do volajícího kódu. Z tohoto pravidla vyplývá, že by se místo textového jména souboru měl metodám save() a restore() předávat již objekt otevřeného souboru.

Další problém spočívá v tom, že explicitně určujeme jméno souboru, do kterého se objekt ukládá. Pokud byste někdy takto vybudovanou třídu chtěli použít například v jiné aplikaci, mohli byste se setkat s komplikacemi.

V tomto místě pokládám za vhodné zmínit se o tom, že u odvozené třídy *musíme sami zajistit* uvnitř metody __init__() volání stejnojmenné metody bázové třídy, které jako první parametr předáváme self.

Doufám, že vám tato kapitola dala přičichnout k objektově orientovanému programování. Další informace a příklady můžete nalézt v nějaké dalším on-line učebnici nebo si můžete přečíst některou z knih, o kterých jsme se zmiňovali na začátku.

Událostmi řízené programování

Zatím jsme se zabývali pouze dávkově orientovanými programy. Vzpomeňte si, že programy mohou být *dávkově orientované* — tyto programy jsou spuštěny, něco udělají a skončí — nebo *řízené událostmi* — ty jsou spuštěny, čekají na výskyt *události*, na kterou nějak zareagují, a skončí teprve v případě, kdy je jim to řečeno příslušnou událostí. Jak tedy vytvoříme událostmi řízený program? Na věc se podíváme dvěma způsoby — nejdříve si nasimulujeme prostředí s událostmi a potom si vytvoříme velmi jednoduchý program s grafickým uživatelským rozhraním (GUI), který pro generování událostí využívá operačního systému a jeho okolí.

Simulace cyklu pro zpracování událostí

Součástí každého událostmi řízeného programu je cyklus, ve kterém se zachytávají vzniklé události a zpracovávají se. Události mohou být generovány operačním systémem, což je případ prakticky všech programů s grafickým uživatelským rozhraním, nebo program sám zjišťuje, zda nenastávají určité události, což se často děje v případech zabudovaných řídicích systémů, jako jsou ty, které používají fotoaparáty, atd.

Poznámka překladatele: V uvedeném druhém případě můžeme také říci, že události přicházejí z vnějšího prostředí. Podobným zdrojem událostí jsou vaše prsty dopadající na klávesy klávesnice nebo pohyb ruky, který je snímán myší. Klávesnice nebo myš převádějí mechanické události z vnějšího světa do podoby elektrických signálů uvnitř počítače. Ty se nakonec přes vhodná zařízení dostávají až na úroveň volání podprogramů operačního systému. A operační systém je převede až do podoby datových údajů, které zařadí do příslušné fronty událostí.

Vytvoříme program, který sleduje jediný typ událostí — vstup z klávesnice — a zpracovává je až do doby, kdy je přijata ukončující událost. V našem případě touto ukončující událostí bude stisk mezerníku. Příchozí události se budou zpracovávat velmi jednoduchým způsobem — vytiskneme prostě ASCII kód dané klávesy. Program vytvoříme v jazyce BASIC, protože nám poskytuje pěknou a snadno použitelnou funkci pro čtení jednotlivých kláves — funkci INKEY$.

Nejdříve implementujeme hlavní tělo programu, které jednoduše zahájí cyklus, ve kterém se zachycují události. Pokud je rozpoznána platná událost, volají se příslušné podprogramy její zpracování.

```
' Deklarujeme podprogramy pro ošetření událostí.
DECLARE SUB zpracujUdalostKlavesy(k AS STRING)
DECLARE SUB zpracujUdalostUkonceni(k AS STRING)

' Nejdříve smažeme obrazovku a oznámíme uživateli,
' jak se dá činnost ukončit.
CLS
PRINT "Končí se stiskem mezerníku..."
PRINT

' Prováděj nekonečný cyklus.
WHILE 1
k$ = INKEY$
delka = LEN(k$)
IF delka <> 0 THEN
' Předej událost ke zpracování příslušné funkci.
IF k$ <> " " THEN
CALL zpracujUdalostKlavesy(k$)
ELSE
CALL zpracujUdalostUkonceni(k$)
END IF
END IF
WEND
```

Povšimněte si, že se hlavní tělo nestará o to, co se s událostmi stane. Události se zde jen rozpoznávají a předávají se ke zpracování příslušným funkcím. Nezávislost zachytávání události na způsobu jejího zpracování patří ke klíčovým rysům událostmi řízeného programování.

Nyní můžeme implementovat zmíněné dvě funkce pro zpracování událostí. První z nich, zpracujUdalostKlavesy, jednoduše vytiskne ASCII kód znaku, který odpovídá stisknuté klávese:

```
SUB zpracujUdalostKlavesy(k AS STRING)
' Vytiskni platné znaky
delka = LEN(k)
IF delka = 1 THEN ' Jde o jednoduchý znak.
PRINT ASC(k)
ELSE
IF delka = 2 THEN
' Nealfanumerický znak. Tiskneme kód druhého znaku.
PRINT ASC(MID$(k, 2, 1))
END IF
END IF
END SUB
```

Funkce zpracujUdalostUkonceni je velmi jednoduchá. Jednoduše ukončí běh programu použitím příkazu STOP.

```
SUB zpracujUdalostUkonceni(k AS STRING)
STOP
END SUB
```

Pokud by uvedený kód byl vytvářen jako pracovní rámec (framework) používaný v mnoha projektech, pak bychom na jeho začátek zařadili volání *funkce pro inicializaci* (tedy *pro počáteční nastavení* a na jeho konec volání *funkce pro úklid* (clean up). Programátor by pak mohl použít část s cyklem bez zásahů do kódu a doplnil by jen své vlastní funkce pro inicializaci, zpracování a pro úklidové práce.

Přesně tak to dělá většina prostředí, orientovaných na grafické uživatelské rozhraní (GUI). Smyčka zpráv je součástí operačního prostředí (jádra operačního systému) nebo programátorského prostředí (framework). Aplikace jsou *smluvně zavázány* k tomu, aby poskytly funkce pro obsluhu událostí a nějakým způsobem je *navázaly* na kód smyčky zpráv. Ukažme si to prakticky při současném seznámení se s knihovnou Tkinter, která se dodává se systémem Python.

Program s grafickým uživatelským rozhraním

V tomto příkladu použijeme nástrojovou sadu (toolkit [túlkit]) zvanou Tkinter, která se dodává se systémem Python. Jde o obal (wrapper), vytvořený pro jazyk Python a obalující originální nástrojovou sadu zvanou Tk, která byla původně napsána jako rozšíření k Tcl a která je dostupná i pro jazyk Perl. Verze pro Python má podobu objektově orientovaného programátorského prostředí (framework), které se — podle mého názoru — používá mnohem snadněji, než původní procedurální podoba Tk. Nebudu zde příliš zabíhat do problematiky grafického uživatelského rozhraní. Chci se spíše soustředit na styl programování — na to, jak s použitím Tkinter pracovat se smyčkou událostí, jak musí programátor vytvořit grafické uživatelské rozhraní a jak se zpracovávají příchozí události.

V příkladu vytvoříme aplikační třídu KlavesovaAplikace, která v rámci metody __init__ vytvoří grafické uživatelské rozhraní a *naváže* klávesu mezerníku na metodu zpracujUdalostUkonceni. Třída definuje i požadovanou metodu zpracujUdalostUkonceni.

Grafické uživatelské rozhraní se skládá z okna pro textový vstup (widget — viz poznámka k dřívějšímu výskytu tohoto pojmu). Jeho standardní chování spočívá v opisování zadaných znaků na displej, tedy do plochy svého okna.

U objektově orientovaných prostředí řízených událostmi je vytvoření třídy pro celou aplikaci běžným zvykem. Je to dáno tím, že mezi konceptem událostí zasílaných programu a konceptem zpráv zasílaných objektů existuje řada podobností. Oba koncepty se na sebe vzájemně velmi snadno převádějí. Funkce pro zpracování událostí se pak stávají metodami aplikační třídy.

Jakmile máme třídu definovánu, jednoduše vytvoříme její instanci a zašleme jí zprávu mainloop ([mein lúp], tedy hlavní smyčka — toto a další jména nelze ve zdrojovém textu přeložit do českého jazyka, protože jde o jména, která jsou definována uvnitř Tkinter).

Kód vypadá následovně:

```
# Použijeme import ve tvaru 'from X import *', abychom se vyhnuli
# nutnosti zapisovat vše jako 'Tkinter.xxx'.
from Tkinter import *

# Vytvoříme třídu aplikace, která definuje grafické uživatelské
# rozhraní (GUI) a metody pro zpracování událostí.
class KlavesovaAplikace(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.txtBox = Text(self)
        self.txtBox.bind("<space>", self.zpracujUdalostUkonceni)
        self.txtBox.pack()
        self.pack()

    def zpracujUdalostUkonceni(self, udalost):
        import sys
        sys.exit()


# Nyní vytvoříme instanci a nastartujeme smyčku zpráv.
mojeAplikace = KlavesovaAplikace()
mojeAplikace.mainloop()
```

Ve verzi psané v jazyce BASIC jsme samozřejmě tiskli ASCII kódy odpovídající všem klávesám a neopisovali jsme jenom tisknutelné znaky odpovídajích kláves, jak to děláme zde. Ale nic nám nebrání, abychom zachytávali stisky všech kláves a dělali přesně totéž. Za tím účelem musíme do metody __init__ přidat následující řádek:

```
self.txtBox.bind("<Key>", self.zpracujStiskKlavesy)
```

Musíme také doplnit následující metodu, pro zpracování odpovídajících událostí:

```
def zpracujStiskKlavesy(self, udalost):
    str = "%d\n" % udalost.keycode
    self.txtBox.insert(END, str)
    return "break"
```

Poznámka 1: Kód klávesy je uložen v položce keycode objektu události. Abych to zjistil, musel jsem se podívat do zdrojového kódu Tkinter.py... Vzpomínáte si, že zvědavost patří ke klíčovým vlastnostem programátora?

Poznámka 2: Příkaz return "break" je magickým signálem pro Tkinter, který říká, že daný prvek (widget) nemá provádět standardní (default) zpracování události. Pokud bychom tento řádek neuvedli, pak by se v textovém okně zobrazoval ASCII kód a za ním by byl opsán příslušný znak — což zde neodpovídá našemu přání.

Poznámka překladatele: pokud si chcete hrát s klávesnicí a podívat se na kódy více kláves, zkuste předpis pro přechod na nový řádek nahradit například čárkou a mezerou.

To by pro tuto chvíli stačilo. Výše uvedený text, nebyl míněn jako text pro výuku Tkinter. Tím se bude zabývat téma následující kapitoly. Používáním Tk a Tkinter se zabývá také několik knih.

Programování grafického uživatelského rozhraní s Tkinter

Následující téma se věnuje nejdříve způsobu výstavby programu s grafickým uživatelským rozhraním (GUI) v obecném smyslu. Poté se zaměříme na to, jak se pro tento účel používá *rodná* nástrojová sada systému Python pro tvorbu grafického uživatelského rozhraní — Tkinter. Nečekejte, že půjde o dokonalou referenční příručku pro Tkinter. Nejedná se dokonce ani o ucelenou učebnici. Velmi dobrá a detailní učebnice, která se tomuto tématu věnuje, již existuje. Odkaz na ni naleznete na webovských stránkách systému Python. Tato kapitola se vás spíše bude snažit provést základy programování grafického uživatelského rozhraní (GUI), seznámí vás s jeho základními prvky a způsobem jejich použití. Podíváme se také na to, jak nám může při výstavbě aplikace s grafickým uživatelským rozhraním pomoci objektově orientované programování.

Ze všeho nejdříve bych rád řekl, že se zde nenaučíte nic nového, co se týká programování. Programování grafického uživatelského rozhraní je stejné jako jakýkoliv jiný druh programování. Můžete používat posloupnosti příkazů, cykly, větvení a moduly stejně, jak jsme si ukázali dříve. Čím se programování grafického uživatelského rozhraní obvykle liší je to, že obvykle používáme nějakou *sadu nástrojů* (toolkit) a to nás nutí postupovat v souladu se vzory, které do návrhu sady nástrojů vnesl její tvůrce. Každá nová sada nástrojů definuje své aplikační programátorské rozhraní (API) a množinu návrhových pravidel, které se vy, jako programátor, musíte naučit. A právě to je ten důvod, proč se většina programátorů snaží tvořit standardy na základě pouze několika nástrojových sad, které jsou dostupné pro více programovacích jazyků. Zvládnutí nové nástrojové sady (toolkit [túlkit]) bývá mnohem obtížnější, než zvládnutí nového programovacího jazyka. Většina programovacích jazyků, které se používají pro vytváření aplikací s okny, bývá dodávána spolu s toolkitem. (Jde obvykle o tenkou vrstvu nad nejjednoduššími nástroji, které jsou zabudovány přímo do systému, který okna podporuje.) Příkladem mohou být Visual Basic, Delphi (Kylix) a Visual C++/.NET.

Java se od nich odlišuje tím, že se jazyk dodává s jeho vlastním grafickým toolkitem (nazývá se Swing). Ten je podporován na každé platformě, kde může běžet Java — což jsou téměř všechny platformy.

Poznámka překladatele: VB, Delphi jsou jazyky s podporou Rapid prototyping, resource, nástroje, podpora pro okna uvnitř jazyka. Visual C++ v podstatě standardní překladač jazyka C++ s některými nestandardními rozšířeními. .NET je jazykově nezávislá, objektově orientovaná platforma, kde část pro okna je jen částí. Longhorn (2005/2006) bude překrývat celou množinu funkcí jádra systému (kompatibilita). .Net jazykově neutrální, jazyky CLI, CLR se podobá Java runtime, WindowForms lze přirovnat k Swingu, ale je jazykově nezávislý, C# lze přirovnat k Javě. Celkově má .Net blíže k jádru systému. Stručné vysvětlení ponechat zde, detaily do cztuttrn.

Existují ale i další toolkity, které můžete pro konkrétní operační systém (Unix, Mac, Windows, atd.) získat samostatně. Jejich součástí jsou obvykle adaptéry, které umožňují jejich použití z různých jazyků. Některé z nich jsou komerční, ale řada z nich je volně dostupná (freeware). Jako příklad uveďme GT/K, Qt, Tk. Všechny mají své webové stránky. Vyzkoušejte například:

- wxPython — pythonovská verze toolkitu wxWindows, který je ve skutečnosti napsán v C++.
- PyQt — pythonovský obal toolkitu Qt, který lze používat s většinou jazyků.
- pyGTK — pythonovský obal The Gimp Toolkit neboli GTK+. Jde o volně použitelný projekt (freeware), který je intenzivně využíván v rámci komunity uživatelů systému Linux.

V toolkitech Qt a GT/k je napsána většina linuxových aplikací. Oba jsou pro nekomerční použití dostupné zdarma. (To znamená, že je můžete volně používat, pokud nechcete své programy prodávat za účelem výdělku.) Pokud chcete, můžete pro Qt získat i komerční licenci.

U jazyka Python se za standardní prostředí pro tvorbu grafického uživatelského rozhraní považuje Tkinter (je součástí instalace). Prostředí Tkinter je založeno na Tk, což je velmi starý toolkit, dostupný pro více operačních systémů. A právě na tuto nástrojovou sadu se podíváme blíže. Její verze jsou k dispozici i pro jazyky Tcl a Perl.

Principy, na kterých je toolkit Tk založen, se od ostatních nástrojových sad mírně liší. Proto si na závěr uvedeme stručný přehled jiného populárního nástroje pro tvorbu grafického uživatelského rozhraní v systému Python (a také v jazycích C/C++), který je založen na obvyklejších přístupech. Nejdříve si ale uveďme obecné principy.

Jak už jsme se dříve několikrát zmínili, přirozenou vlastností aplikací s grafickým uživatelským rozhraním je to, že jsou téměř vždy řízeny událostmi. Pokud si nevzpomínáte, co se tím myslí, zopakujte si téma událostmi řízeného programování. Předpokládám, že *z uživatelského hlediska* již grafické uživatelské rozhraní znáte. Zaměříme se na to, jak takové programy fungují *z hlediska programátora*. Nebudeme zabíhat do takových detailů, jak se například tvoří rozsáhlá a složitá grafická uživatelská rozhraní s mnoha okny, rozhraní pro práci s více dokumenty (MDI) a podobně. Přidržíme se takových základů, jako je vytváření jednoduchého okna aplikace s nějakými popisnými texty, s tlačítky, prvky pro vstup textu a s okny pro zobrazování zpráv (message box).

Nejdříve si zkontrolujme naši slovní zásobu. Programování grafického uživatelského rozhraní používá svou vlastní sadu programátorských pojmů. Nejběžnější s nich jsou uvedeny v následující tabulce:

Pojem	Vysvětlení
Okno (Window)	Plocha na obrazovce, která je ovládána aplikací. Okna mají obvykle obdélníkový tvar, ale některá prostředí pro tvorbu grafického uživatelského rozhraní dovolují použití i jiných tvarů. Okna mohou obsahovat další okna. Často je každý ovládací prvek grafického uživatelského rozhraní tvořen svým vlastním oknem.
Ovládací prvek (Control)	Ovládací prvek je objekt grafického uživatelského rozhraní, který se používá pro ovládání aplikace. Ovládací prvky mají určité vlastnosti a obvykle generují nějaké události. Ovládací prvky obvykle souvisejí s odpovídajícími objekty na aplikační úrovni a jejich události jsou svázány s metodami aplikačních objektů. Při výskytu události se tedy provede jedna z odpovídajících metod. Prostředí pro tvorbu grafického uživatelského rozhraní obvykle poskytuje mechanismus, kterým se vazba mezi událostí a metodou ustanoví.
Widget	Ovládací prvky mají někdy viditelnou podobu. Některé ovládací prvky (jako třeba časovače) sice mohou být spojeny s nějakým oknem, ale samy o sobě nejsou viditelné. Prvky typu widget tvoří tu podmnožinu ovládacích prvků, které jsou viditelné a se kterými může uživatel nebo programátor manipulovat. Ukážeme si použití následujících prvků typu widget: • Rámec (frame), • popisný text (label), • tlačítko (button), • pole pro vstup textu (text entry), • okno se zprávou (message box). Jinde v této učebnici jsou použity další prvky, kterými se ale v této kapitole nebudeme zabývat: • Okno pro psaní textu (text box), • přepínací tlačítko (radio button). A nakonec si uveďme prvky, kterými se nebudeme zabývat vůbec: • Kreslicí plocha (canvas),

	• prvek pro výběr (check button) — lze vybírat více nabídnutých možností najednou, • obrázek (image) — pro zobrazování obrázků ve formátu BMP, GIF, JPEG a PNG, • okno se seznamem (list box), • Menu/MenuButton — pro tvorbu menu, • Scale/Scrollbar — pro znázornění a úpravu pozice pohledu. **Poznámka překladatele:** Protože pro pojem *widget* nemáme dostatečně stručný český ekvivalent, překládám jej v zájmu dobré čitelnosti textu jako *ovládací prvek*, i když jsem si vědom, že tento pojem je obecnější, než v případě anglického originálu. Pokud by mohly vzniknout nejasnosti kolem charakteru prvku nebo pokud chci naznačit, co obsahoval originální text, uvádím slovo widget jako součást opisu, kterým obcházím nutnost jeho skloňování. V některých případech slovo widget uvádím v závorkách.
Rámec (Frame)	Jde o prvek typu widget, který se používá k seskupení dalších prvků typu widget dohromady. Rámec se často používá jako reprezentant celého okna. Uvnitř rámce se mohou nacházet další rámce.
Předpis pro rozložení prvků (Layout)	Ovládací prvky jsou uvnitř rámce umístěny podle určitého předpisu. Ten může být definován různým způsobem. Buď se používají souřadnice odpovídající pixelům na obrazovce, nebo se poloha určuje relativně vůči jiným prvkům (zarovnání vlevo, nahoru, atd.), nebo se využívá uspořádání do mřížky nebo do tabulky. Použití souřadnicového systému je sice snadno srozumitelné, ale obtížně se používá například v situaci, kdy dochází ke změnám rozměrů okna. Pokud se umístění prvků předepisuje souřadnicemi, pak by začátečníci měli používat raději okna, u kterých nelze měnit rozměry.
Potomek (Child)	Při tvorbě aplikací s grafickým uživatelským rozhraním často vzniká hierarchické uspořádání ovládacích prvků. Rámec na nejvyšší úrovni, který představuje okno aplikace, se skládá z podrámců, které obsahují další rámce nebo ovládací prvky. Vazby mezi ovládacími prvky si můžeme zobrazit jako stromovou strukturu, ve které má každý ovládací prvek nadřazen jeden rodičovský prvek a několik potomků (podřízených prvků). Ve skutečnosti je tato struktura závislostí přímo uložena v jednotlivých prvcích (prvek si udržuje odkazy na své podřízené prvky — potomky), takže programátor — nebo častěji samo prostředí grafického uživatelského rozhraní — může provádět některé akce nad ovládacím prvkem a všemi jeho potomky najednou.

Exkurze mezi některé běžné ovládací prvky

V této sekci vytvoříme přes příkazový řádek systému Python jednoduchá okna a ovládací prvky (widget). Poznamenejme, že aplikaci, která využívá Tkinter, nemůžeme spolehlivě spouštět z prostředí IDLE, protože IDLE samotné je aplikací, která Tkinter využívá. Z IDLE samozřejmě můžeme použít jeho editor a vytvořit v něm zdrojové texty, ale výsledek musíme spustit z příkazového řádku operačního systému. Uživatelé prostředku Pythonwin naopak takovou aplikaci spouštět mohou, protože Pythonwin používá jinou nástrojovou sadu pro tvorbu grafického uživatelského rozhraní — MFC (Microsoft Foundation Classes). Nicméně i v prostředí Pythonwin můžeme u tkinterovských aplikací pozorovat jisté neočekávané projevy chování. Proto zde raději použijeme příkazový řádek systému Python, který máme k dispozici po spuštění interpretu jazyka Python prostředky operačního systému (v DOSovém okně).

```
>>> from Tkinter import *
```

Mezi první požadavky každého tkinterovského programu patří importování jmen ovládacích prvků. Mohli byste samozřejmě importovat jen modul, ale velice rychle byste se unavili tím, že byste před každé jméno museli připisovat Tkinter.

```
>>> top = Tk()
```

Tento příkaz vytvoří ovládací prvek na nejvyšší úrovni hierarchie našich ovládacích prvků. Všechny ostatní ovladací prvky budou vytvořeny jako jeho potomci. Povšimněte si, že se zobrazilo nové prázdné okno s textemtk v titulku okna, s ikonou Tk a s obvyklou sadou ovládacích tlačítek (zmenšení do ikony, zvětšení přes celou obrazovku, atd.). Tak, jak budeme aplikaci postupně vytvářet, budeme do tohoto okna přidávat další prvky.

```
>>> dir(top)
['_tclCommands', 'children', 'master', 'tk']
```

Funkce dir nám ukáže všechna jména, která jsou zadanému argumentu známa. Můžeme ji použít i pro moduly, ale v tomto případě se chceme podívat na vnitřek objektu top, což je instance třídy Tk. Jde o jeho atributy. Povšimněte si zejména atributů children a master, která zachycují vazby v hierarchii ovládacích prvků. Povšimněte si také atributu _tclCommands, který má svůj původ ve skutečnosti — jak si můžete vzpomenout —, že Tkinter je vytvořen nad nástrojovou sadou systému Tcl, která se jmenuje Tk.

Poznámka překladatele: Vypsaný seznam jmen je ve skutečnosti mnohem delší a popíše vám celou obrazovku — přinejmenším u verze Python 2.2.

```
>>> F = Frame(top)
```

Vytvoří se ovládací prvek (widget) Frame, ve kterém budou umístěny ovládací prvky, které budeme používat. Při vytváření instance Frame je jako první argument (a v tomto případě jediný) použit top. Tím říkáme, že F bude ovládací prvek, vytvořený jako potomek ovládacího prvku top.

```
>>> F.pack()
```

Povšimněte si, že po provedení tohoto příkazu se okno Tk scvrkne na velikost přidaného ovládacího prvku třídy Frame. Ten je v současnosti prázdný, takže okno je teď velmi malé. Metoda pack() aktivuje *správce rozložení*(Layout Manager), který je znám jako *packer* (pakovač, stlačovač). Ten se při jednoduchých rozloženích prvků používá velmi snadno, ale s tím, jak se rozložení prvků stává složitějším, začíná být poněkud neohrabaný. Pro tyto chvíle se jej budeme držet — snadno se používá. Povšimněte si, že ovládací prvky (widget) nebudou v naší aplikaci vidět až do té doby, než provedeme jejich "spakování" (nebo použijeme jinou metodu správce rozložení prvků).

```
>>> lHello = Label(F, text="Ahoj, vy tam!")
```

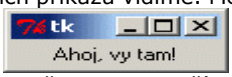
Tímto příkazem vytvoříme nový objekt lHello jako instanci třídy Label. Rodičovským (nadřízeným) ovládacím prvkem je F a atributu text přiřazujeme hodnotu "Ahoj, vy tam!". Konstruktory objektů modulu Tkinter mívají obvykle mnoho parametrů (každý z nich má přednastavenou hodnotu). Všimněte si, že se jim často předávají argumenty způsobem, kdy využíváme možnosti určení příslušného parametru jménem. (Srovnejte to s častějším, pozičním způsobem předávání argumentů, kdy udáváme pouze hodnotu parametru, ale nikoliv jméno. V takovém případě musíme hodnotu uvést na správné pozici.) Povšimněte si rovněž, že objekt není dosud viditelný, protože jsme dosud neprovedli "spakování".

Nakonec si uveďme poznámku ke konvenci pro volbu jména: Před jméno Hello jsem přidal malé l jako Label, které má připomínat význam objektu. Stejně jako u ostatních konvencí pro volbu jména je dodržování této konvence věcí vašeho názoru. Podle mě je její dodržování užitečné.

Poznámka překladatele: V souvislosti s tvorbou programů pro první verze operačního systému Microsoft Windows byl vypracován celý systém předpon přidávaných ke jménům proměnných, funkcí a dalších prvků. Je znám jako *maďarská notace*. Podrobnosti můžete najít v Charles Simonyi: "Hungarian notation". V poslední době ovšem převládá názor, že používání podobné notace může způsobovat více problémů, než užitku. Týká se to především větších projektů a jazyků s velmi silnou typovou kontrolou, jako je například jazyk C++. Používejte proto podobných konvencí s mírou. Vhodná volba identifikátoru (tj. jména) může potřebu používání podobných předpon zmírnit.

```
>>> lHello.pack()
```

Teď už výsledek předchozích příkazů vidíme. Měl by vypadat nějak takto:



Objektu třídy Label můžeme parametry konstruktoru předepsat i další vlastnosti, jako je například typ a barva písma. Tyto vlastnosti si ale můžeme zpřístupnit voláním metody configure, kterou ovládací prvky (widget) modulu Tkinter podporují:

```
>>> lHello.configure(text="Nashledanou.")
```

Zpráva se změnila. Bylo to docela snadné, že? Použití metody configure je výhodné především v případech, kdy chcete změnit několik vlastností najednou, protože je můžeme najednou předepsat jako její argumenty. Pokud ovšem chcete změnit jen jedinou vlastnost, jako jsme to učinili v naposledy uvedeném případě, můžeme se k objektům chovat, jako kdyby se jednalo o slovníky (dictionary, vyhledávací tabulky). Takže můžeme psát:

```
>>> lHello['text'] = "Ahoj, jsem tady zase!"
```

... je to kratší a snad i srozumitelnější.

Objekty typu Label (popisné texty) patří k docela nudným ovládacím prvkům. Mohou pouze zobrazit text, který je určen jen ke čtení — i když v různých barvách, různým písmem a v různé velikosti. (Ve skutečnosti je lze použít i pro zobrazení jednoduché grafiky, ale jak to udělat si ukážeme až později.)

Dříve než se podíváme na další typ objektu, zbývá nám předvést ještě jednu věc — způsob, jak můžeme nastavit titulek okna. Dosáhneme toho použitím metody ovládacího prvku na vrcholu hierarchie, objektu top:

```
>>> F.master.title("Ahoj")
```

Stejného efektu jsme mohli dosáhnout přímým použitím objektu top, ale technika, využívající přístup prostřednictvím vlastnosti master objektu třídy Frame, bývá užitečná — jak uvidíme později.

```
>>> bQuit = Button(F, text="Konec", command=F.quit)
```

Tímto příkazem vytvoříme nový ovládací prvek, tlačítko (button, čti [batn]). Tlačítko nese nápis "Konec" a je spojeno s příkazem F.quit. Povšimněte si, že předáváme jméno metody. Neprovádíme volání této metody, protože jsme za jméno nepřidali závorky. To znamená, že se předává objekt s charakterem funkce ve smyslu chápaném v jazyce Python. Může to být vestavěná metoda modulu Tkinter, jako v tomto případě, nebo jakákoliv jiná, námi definovaná funkce. Funkce nebo metoda nesmí mít žádné argumenty. Metoda quit, podobně jako metoda pack, je definována v bázové třídě, kterou dědí všechny ovládací prvky modulu Tkinter.

```
>>> bQuit.pack()
```

Metoda pack opět zajistí zviditelnění tlačítka.

```
>>> top.mainloop()
```

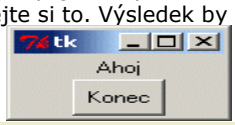
Tímto odstartujeme provádění tkinterovské smyčky zpráv. Povšimněte si, vyzývací znaky '>>> ' příkazového řádku systému Python nyní zmizely. Podle toho poznáme, že řízení další činnosti přešlo do režie Tkinter. Pokud stisknete tlačítko Konec, vyzývací znaky příkazového řádku se znovu objeví, což je důkaz toho, že zafungoval náš parametr command.

Poznamenejme, že pokud totéž provádíme z prostředí Pythonwin nebo IDLE, může být chování odlišné. Pokud tomu tak skutečně je, zkuste dosud uvedené příkazy zapsat do pythonovského skriptu, tedy do textového souboru s příponou py, a spusťte jej z příkazového řádku operačního systému.

On vlastně nastal příhodný okamžik k tomu, abychom to stejně vyzkoušeli. Když se to tak vezme, tímto způsobem se v praxi provozuje většina tkinterovských programů. Použijme klíčové příkazy z těch, o kterých jsme se zatím bavili:

```
from Tkinter import *

# Vytvoříme samotné okno.
top = Tk()
F = Frame(top)
F.pack()

# Přidáme ovládací prvky.
lHello = Label(F, text="Ahoj")
lHello.pack()
bQuit = Button(F, text="Konec", command=F.quit)
bQuit.pack()

# Spustíme smyčku událostí.
top.mainloop()
```

Volání metody top.mainloop zahájí provádění tkinterovské smyčky událostí. V tomto případě bude jedinou zachycenou událostí ta, která odpovídá stisku tlačítka a která je spojena s provedením metody F.quit. Její provedení způsobí ukončení aplikace. Vyzkoušejte si to. Výsledek by měl vypadat takto:



Bližší průzkum umísťování prvků (layout)

Poznámka: V následujícím textu budou příklady uváděny v podobě, jakou mají v pythonovských zdrojových souborech. Nebudou tedy uvozeny řetězcem '>>> ', který se vypisuje na začátku vstupního řádku interpretu jazyka Python.

V této části bych se rád zaměřil na to, jak Tkinter umísťuje prvky (widget) uvnitř okna. V předchozím textu jsme si již ukázali prvky typu Frame, Label a Button. Ty nám pro potřeby této části textu postačí. V předchozím příkladu jsme

používali metodu prvku (widget) zvanou pack k umístění prvku uvnitř jeho rodičovského okna. Technicky vzato jsme tím aktivovali *správce rozložení prvků* systému Tk, kterému se říká *packer*. Úkolem správce rozložení prvků (Layout Manager) je určení nejlepšího rozložení prvků, které je založeno na nápovědě předepsané programátorem a na omezeních, jako je například velikost okna, kterou ovlivňuje uživatel. Některé typy správců rozložení prvků používají přesné umístění uvnitř okna, které je předepsáno v pixelech[1]. S tímto přístupem se běžně setkáte v systému Microsoft Windows, například při používání programátorského prostředí Visual Basic. V modulu Tkinter dosáhneme téhož při použití správce rozložení prvků, kterému se říká *placer*(doslova "umísťovač") — činíme tak voláním jeho metody place. V této učebnici se uvedeným správcem rozložení zabývat nebudeme, protože obvykle bývá lepší, když si vybereme jeden ze zbývajících, inteligentnějších správců rozložení prvků. Jejich použití zbavuje programátory starosti o to, co se stane, když okno změní své rozměry.

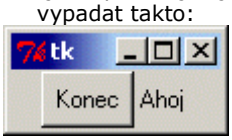
V Tkinter je nejjednodušším správcem rozložení prvků takzvaný *packer*, který jsme již používali v předchozím textu. *Packer*, pokud mu neřekneme jinak, jednoduše skládá ovládací prvky (widget) jeden na druhý. Z hlediska běžných ovládacích prvků tuto vlastnost využijeme velmi zřídka, ale pokud sestavujeme rozhraní naší aplikace z rámečků (Frame), pak můžeme považovat skládání rámečků na sebe za docela rozumný přístup. Ostatní ovládací prvky můžeme do rámečků umísťovat buď s využitím správce rozložení typu *packer* nebo uvnitř rámečku podle potřeby využijeme vlastností jiného správce rozložení. Příklad použití takového přístupu můžete najít v případové studii.

Ale dokonce i tak jednoduchý správce rozložení prvků, jako je *packer*, poskytuje celou řadu voleb. Například uvedením argumentu side (strana, do strany, stranově) můžeme předepsat uspořádání našich prvků ve vodorovném, místo ve svislém směru:

```
lHello.pack(side="left")
bQuit.pack(side="left")
```

Tyto příkazy přinutí prvky, aby se skládaly zleva (left [left], znamená levý nebo vlevo). Takže první prvek (typu Label) se objeví úplně vlevo. Za ním následuje další prvek (typu Button). Pokud uvedené řádky příkladu upravíme uvedeným způsobem, bude výsledek vypadat takto:

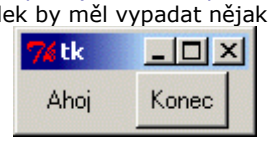
A pokud změníme hodnotu "left" na "right" ([rajt] znamená pravý, vpravo), pak se prvek typu Label objeví úplně vpravo a prvek typu Button vlevo od něj, jinými slovy, co nejvíc vpravo, jak je to za aktuálního stavu možné. Výsledek bude vypadat takto:



Jedna z věcí, které si můžete všimnout je, že to nevypadá moc hezky, protože pvky jsou příliš nalepeny na sebe. Správce *packer* nám ale nabízí další parametry, které nám umožní vypořádat se i s touto situací. Snadno použitelné jsou takzvané *vycpávky* (také výplně; v originále padding, čti pading). Můžeme předepsat vodorovné vycpávky (padx) a svislé vycpávky (pady). Jejich hodnoty se udávají v pixelech. Doplňme tedy do našeho příkladu vodorovné vycpávky:

```
lHello.pack(side="left", padx=10)
bQuit.pack(side='left', padx=10)
```

Výsledek by měl vypadat nějak takto:



Pokud zkusíte měnit velikost okna, můžete pozorovat, že oba prvky zachovávají svou vzájemnou pozici, ale zůstávají uprostřed okna. Proč tomu tak je? Vždyť jsme je přeci nechali poskládat zleva? Odpověď zní: prvky jsme poskládali dovnitř obalujícího rámečku (Frame), ale samotný rámeček jsme do okna vložili (metodou pack) bez uvedení parametru side. Takže rámeček je jako celek v okně umístěn nahoře uprostřed, což odpovídá základnímu chování správce rozložení typu *packer*. Pokud bychom chtěli, aby byly prvky umístěny na požadované straně okna, musíme i při volání metody pack pro objekt typu Frame uvést vhodnou hodnotu parametru side:

```
F.pack(side='left')
```

Nyní si můžete všimnout, že při změně svislého rozměru okna zůstávají prvky uprostřed výšky okna — jde opět o základní chování správce rozložení typu *packer*.

Nechám už na vás, abyste si sami pohráli s hodnotami parametrů padx a pady. Pozorujte vliv jejich různých hodnot a kombinací. Zejména parametry side a padx/pady umožňují při použití správce rozložení typu *packer*poměrně pružné možnosti umísťování prvků typu widget. Existují ještě další parametry. Každý z nich přidává další, jemnější podobu řízení umístění. Detaily hledejte na referenčních stránkách modulu Tkinter.

Modul Tkinter poskytuje ještě další správce rozložení, které jsou známy jako *grid* (mřížka) a *placer*(umísťovač). Použití správce typu grid aktivujeme voláním metody grid() místo pack(). V případě použití správce typu *placer* voláme místo metody pack() metodu place(). Každá z uvedených metod má svou sadu parametrů, ale protože se zde zabýváme pouze správcem typu *packer*, budete muset detaily hledat v učebnici a v referenční příručce Tkinter. Zmíním se jen o tom, že správce typu *grid* zařídí uspořádání prvků do mřížky (jaké překvapení!) uvnitř okna. Jeho použití je užitečné například v případě dialogových oken se zarovnanými poli pro vkládání textu. U správce typu *placer* můžeme použít buď pevné souřadnice v pixelech nebo relativní souřadnice uvnitř okna. Posledně zmiňovaná možnost umožňuje, aby vložený prvek měnil své rozměry současně s pvkem například tak, aby vždy zabíral například 75 procent svislého prostoru. Tento správce umožňuje řešit zvláštní návrhové požadavky, ale vyžaduje to od nás, abychom si předem vše naplánovali. Vřele vám doporučuji, abyste si pro tyto účely obstarali čtverečkovaný papír, tužku a gumu.

Řízení vzhledu za použití rámečků a správce typu *packer*

U prvku (widget) typu Frame můžeme ve skutečnosti ovlivnit několik užitečných vlastností. Když se to tak vezme, není špatné, když můžeme prvky uživatelského rozhraní z logického hlediska obalit rámečkem, ale někdy navíc chceme také něco vidět. Hodí se nám to zejména v případech seskupení ovládacích prvků jako jsou přepínací tlačítka (radio buttons) nebo zaškrtávací pole voleb (check boxes). Třída Frame tento problém řeší tím, že poskytuje vlastnost *relief* — tak jako mnoho dalších prvků Tk typu widget. Relief může nabývat libovolné z následujících hodnot: sunken ([sankn]; ponořený,

vmáčknutý), raised ([reizd]; vystouplý, vyzvednutý), groove([grúv]; drážka, vyrytý) ridge ([ridž]; hřbet, geometrický opak drážky) nebo flat ([flat]; plochý). Vyzkoušejme u našeho dialogového okna hodnotu sunken. Jednoduše změníme řádek, na kterém se vytváří prvek třídy Frame:

```
F = Frame(top, relief="sunken", border=1)
```

Poznámka 1: Musíme uvést i nenulovou hodnotu parametru border ([bódr]; hranice). Pokud tak neučiníme, bude sice plocha prvku typu Frame ponořená, ale hranice mezi ponořenou a okolní plochou bude neviditelná, takže nezpozorujeme žádný rozdíl.

Poznámka 2: … o tom, proč tloušťku hranice (border) neuvádíme v uvozovkách. Znalost toho, zda máme použít uvozovky kolem hodnoty parametru a kdy je vynechávat, patří k jedné z matoucích vlastností Tk. Obecně se dá říci, že u číselných nebo jednoznakových hodnot můžeme uvozovky vynechávat. Pokud jde o směs číslic a písmen nebo o řetězec, musíme použít uvozovky. Podobný problém spočívá v tom, kdy použít malá nebo velká písmena. Naneštěstí zde neexistuje jednoduchý návod. Musíte se prostě učit ze zkušeností. V případě chyby Python často v chybových hlášeních vypisuje seznam přípustných hodnot parametrů.

Poznámka překladatele k poznámce 2: Aby se zvýšila čitelnost zdrojových textů, jsou pro vyhrazené řetězcové argumenty definovány řetězcové proměnné, které se používají v roli předdefinovaných konstant. Jejich jména jsou psána velkými písmeny. Naleznete je v souboru Tkconstants.py. Zde definovaná jména jsou zviditelněna v rámci importu Tkinter. Ve zdrojových textech proto místo

```
F = Frame(top, relief="sunken", border=1)
```

můžeme psát:

```
F = Frame(top, relief=SUNKEN, border=1)
```

Podobně můžeme místo side="left" psát side=LEFT a podobně. Uvedené obraty můžete pozorovat dále v textu.

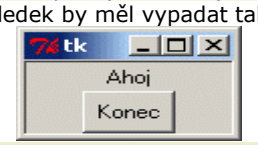
Další věc, které si můžete všimnout, je ta, že Frame nevyplňuje okno. Můžeme to napravit použitím dalšího parametru správce typu *packer*, parameru fill ([fil]; vyplnit). Při volání metody pack() tedy zapíšeme:

```
F.pack(fill=X)
```

Uvedený parametr způsobí vyplnění prostoru ve vodorovném směru. Vyplnění prostoru ve svislém směru zajistíme použitím fill=Y. Mezi běžné požadavky patří vyplnění prostoru v obou směrech. Pro tyto případy máme k dispozici hodnotu parametru BOTH. (Nechtějte po mě, abych zde normálními písmenky zapisoval výslovnost. Každopádně slovo *both* znamená *oba* — tedy vyplňování v obou směrech.)

```
F.pack(fill=BOTH)
```

Výsledek by měl vypadat takto:



Přidejme další prvky

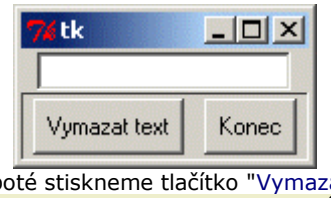
Podívejme se nyní na prvek (widget) pro třídy Entry ([entry]; vstup). Jde o známý prvek pro zadávání jednořádkového textu. Řada jeho metod se shoduje s metodami propracovanějšího prvku (widget) třídy Text, ale tím se zde zabývat nebudeme. Přesto doufám, že používáním metod prvku třídy Entry získáte dobré základy pro pozdější experimenty s prvkem třídy Text.

Vrátíme se opět k našemu programu, který zobrazuje Ahoj, vy tam!, přidáme do něj prvek pro vkládání textu do samostatného prvku typu Frame a také tlačítko, které umí vymazat text, který do pole vepisujeme. Tím si ukážeme nejen to, jak se dá vytvořit a používat prvek typu Entry, ale také jak můžeme definovat své vlastní funkce pro ošetření (zpracování) událostí a jak je navážeme na ovládací prvky.

```
from Tkinter import *

# Nejdříve vytvoříme funkci pro ošetření události.
def evVymazat():
eTxt.delete(0, END)

# Vytvoříme hierarchicky nejvyšší okno a rámeček.
top = Tk()
F = Frame(top)
F.pack(expand=True)

# Nyní vytvoříme rámeček s polem pro vstup textu.
fVstup = Frame(F, border=1)
eTxt = Entry(fVstup)
fVstup.pack(side=TOP, expand=True)
eTxt.pack(side=LEFT, expand=True)

# Nakonec vytvoříme rámeček s tlačítky.
# Pro zvýraznění jej vytvoříme jako ponořený (vmáčknutý).
fTlacitka = Frame(F, relief=SUNKEN, border=1)
bVymazat = Button(fTlacitka, text="Vymazat text", command=evVymazat)
bVymazat.pack(side=LEFT, padx=5, pady=2)
bKonec = Button(fTlacitka, text="Konec", command=F.quit)
bKonec.pack(side=LEFT, padx=5, pady=2)
fTlacitka.pack(side=TOP, expand=True)

# Nyní spustíme smyčku zpráv.
F.mainloop()
```

Povšimněte si, že jméno funkce pro ošetření události (evVymazat) opět předáváme jako hodnotu argumentucommand při vytváření tlačítka bVymazat. Povšimněte si také konvence pro vytváření jména evXXX funkce pro ošetření události — dáváme jí najevo vazbu s odpovídajícím prvkem typu widget.

Po spuštění programu obdržíme následující výsledek:



Pokud něco napíšeme do vstupního pole a poté stiskneme tlačítko "Vymazat text", bude napsaný text opět odstraněn.

Navázání událostí — od ovládacích prvků ke kódu

Doposud jsme pro propojení pythonovských funkcí s událostmi tlačítek — jako prvků grafického uživatelského rozhraní — používali vlastnost tlačítek zvanou command. Někdy ovšem potřebujeme zajistit přesněji a přímo vyjádřený způsob ovládání. Chceme například zachytit událost stisku zvláštní kombinace kláves. Můžeme toho dosáhnout použitím funkce bind ([bajnd]; svázat, spojit), kterou lze přímo vyjádřit vazbu mezi nějakou událostí pythonovskou funkcí.

Do předchozího příkladu dodefinujeme "horkou klávesu" (hot key) — dejme tomu Ctrl-c —, která rovněž způsobí vymazání textu. Potřebujeme tedy navázat kombinaci kláves Ctrl-c na stejnou funkci pro obsluhu událostí, na kterou se váže událost tlačítka Vymazat. Máme tu ale jednu neočekávanou nepříjemnost. Parametrucommand jsme museli předávat jméno funkce, která nesměla mít žádné parametry. Pokud chceme použít k provedení stejné činnosti funkci bind, musí navazovaná funkce definovat jeden parametr. Proto musíme vytvořit novou funkci, která přebírá jeden argument a volá evVymazat. Za definici funkce evVymazat proto přidejme následující definici:

```
def evHorkaKlavesa(udalost):
    evVymazat()
```

A za definici prvku typu Entry přidejme následující řádek:

```
# Definice klávesy je citlivá na velikost písmen.
eTxt.bind("<Control-c>", evHorkaKlavesa)
```

Spusťte znovu upravený program. Nyní můžete text vymazat buď stiskem příslušného tlačítka nebo stiskem kombinace kláves Ctrl-c. Funkci bind můžeme použít i pro zachycení takových událostí, jako jsou kliknutí myši, událost získání nebo ztráty aktivity okna (fokus) nebo dokonce událost, která doprovází situaci, kdy se okno stane viditelným. Více informací na toto téma naleznete v dokumentaci k Tkinter. Nejsložitější obvykle bývá zjistit podobu zápisu požadované události.

Krátká zpráva

Chceme-li uživatelům našeho programu zobrazit krátkou zprávu, můžeme k tomu využít prvek zvaný *Message Box* ([mesidž box]; doslova okno se zprávou). Při využití Tk je to velmi snadné. Za tímto účelem můžeme použít funkce modulu tkMessageBox například takto:

```
import tkMessageBox
tkMessageBox.showinfo("Titulek okna", "Krátká zpráva")
```

Pro zobrazování oken chybových hlášení, varování, dotazů typu Ano/Ne nebo OK/Storno existují také další funkce nazvané showXXX ([šou]; ukaž). Příslušná okna se odlišují různými ikonami a tlačítky. Dvě poslední zmíněné varianty používají místo názvu tvaru showXXX názvy askXXX ([ásk]; zeptej se) a vracejí hodnotu, která říká, jaké tlačítko uživatel stiskl:

```
vysledek = tkMessageBox.askokcancel("Co zvolíte?", "Chcete zastavit činnost?")
print vysledek
```

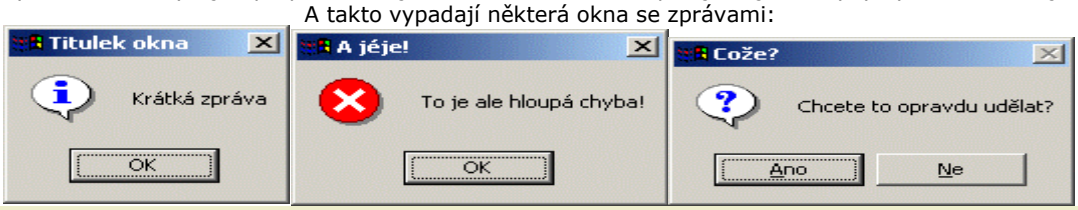
Poznámka překladatele k českým textům s diakritikou: Více podrobností o problémech hledejte v poznámce, ke vstupu českých znaků, která se vztahuje k části učebnice, kde jsme se zabývali vstupem z klávesnice. Naleznete v ní ovšem i údaje k používání českých textů pro prvky grafického uživatelského rozhraní. Stručně: pro zobrazení českého textu v oknech Tk můžeme využít převodu do kódování Unicode. Využijeme k tomu funkci unicode():

```
# -*- coding: cp1250 -*-
import tkMessageBox
vysledek = tkMessageBox.askokcancel(
    unicode("Co zvolíte?", "cp1250"),
    unicode("Chcete zastavit činnost?", "cp1250"))
print vysledek
```

První komentářový řádek říká, že zdrojový text byl programu byl zapsán v kódování cp1250 — je známé také jako windows-1250. Řádek se uvádí hned na začátku skriptu, obvykle jako první nebo druhý. (V unixovém světě se na prvním řádku uvádí jiný typ komentáře, který pro skripty s příznakem spustitelnosti určuje jméno programu, který má skript interpretovat.) Zvláštní tvar řádku s posloupnostmi -*- souvisí s konvencemi, které byly v minulosti zavedeny u některých známých textových editorů. Pokud tento řádek neuvedeme, pak se při spuštění skriptu (přinejmenším od verze Pythonu 2.3) setkáme s varovným hlášením, že byl v řetězci použit znak s kódem větším, než 127 a přitom nebylo upřesněno použité kódování.

Abych v dalších příkladech nemusel vymýšlet texty, které vypadají česky a přitom neobsahují znaky s diakritikou, budu tento obrat používat. V praxi je ale výhodnější nadefinovat si obalující funkce nebo metody tříd (případně odvozené třídy), které převody kódování ukrývají a při psaní zdrojového textu se nám to pak jeví, jako kdyby Python uměl odjakživa česky.

A takto vypadají některá okna se zprávami:



Z pohledu jazyka Tcl

V úvodních částech této učebnice jsme srovnávali Python s Tcl. Proto považuji za rozumné, abychom si ukázali, jak by úvodní příklad s prvky typu Label a Button vypadal v originální podobě zapsané v Tcl/Tk:

```
Label .lHello -text "Ahoj, vy tam!"
Button .bHello -text Konec -command "exit"
wm title . Ahoj
pack .lHello .bHello
```

Jak sami vidíte, zápis je velmi stručný. Hierarchie prvků typu widget je vyjadřována s využitím konvence jejich pojmenování, kde prvek se jménem '.' stojí na nejvyšší úrovni. Jak už je v Tcl zvykem, prvky typu widget jsou vyjadřovány

příkazy, kterým jsou požadované vlastnosti předány formou argumentů. Doufám, že je vám převod parametrů prvků do podoby pojmenovaných argumentů v jazyce Python docela jasný. Pokud tedy při programování s Tkinter potřebujete vyřešit nějaké problémy, můžete použít dokumentaci systému Tcl/Tk (které je velmi mnoho). Přepis do Tkinter je většinou zřejmý.

Dál už se v tomto místě do Tcl/Tk pouštět nebudeme. V následujícím textu si ukážeme běžně používanou techniku pro zabalení aplikací s grafickým uživatelským rozhraním využívajících Tkinter do podoby objektů.

Zabalení aplikací do podoby objektů

Při programování aplikací s grafickým uživatelským rozhraním se běžně celá aplikace obaluje do podoby třídy. To vyvolává otázku, jak do této struktury tříd napasujeme prvky typu widget modulu Tkinter? Na výběr máme dvě možnosti. Buď se rozhodneme pro odvození třídy aplikace od tkinterovské třídy Frame, nebo uložíme referenci na hierarchicky nejvyšší okno do členské proměnné. Posledně zmíněný přístup se běžně používá i u jiných prostředků (toolkit), takže jej použijeme i my.

Pokud byste chtěli vidět použití prvního ze zmiňovaných přístupů, vraťte se k příkladu v kapitole Událostmi řízené programování. (Zmíněný příklad mimo jiné ukazuje základy použití neuvěřitelně univerzálního tkinterovského prvku (widget) třídy Text.)

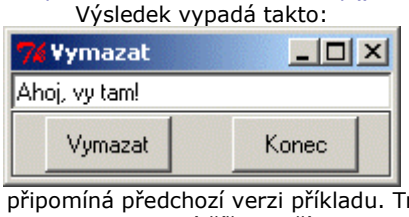
Poznámka překladatele: Přístup, kdy ukládáme referenci na hierarchicky nejvyšší okno odpovídá obecnému doporučení při objektově orientovaném návrhu aplikací. To říká, že bychom měli *dávat přednost kompozici před dědičností*. Jinými slovy to znamená, že pokud si můžeme vybrat, zda spojit funkčnost dvou tříd dohromady, bývá lepší, když nějak spojíme *objekty* dvou jednodušších tříd, než kdybychom vytvářeli jednu novou, složitější třídu. Ve svém důsledku to vede k vyšší pružnosti při budoucích úpravách návrhu aplikace. Návrh bývá také přehlednější. Dědičnost (tj. odvozování jedné třídy objektů z jiné) bychom měli používat především tehdy, když pouze *upravujeme* funkčnost bázové třídy pro speciální účel. Neměli bychom ji používat, když chceme propojit funkčnosti dvou tříd s odlišným účelem.

Výše uvedený příklad, využívající vstupní pole typu Entry, tlačítko Vymazat a tlačítko Konec, převedeme do objektově orientované podoby. Nejdříve si vytvoříme třídu aplikace a v rámci jejího konstruktoru poskládáme viditelné části grafického uživatelského rozhraní.

Referenci na výsledný prvek typu Frame přiřadíme do self.hlavniOkno. Tím lze zajistit přístup k hierarchicky nejvyššímu prvku typu Frame ostatním metodám třídy. Ostatní prvky (widget), ke kterým bychom mohli chtít přistupovat (jako je například pole typu Entry) jsou podobným způsobem přiřazeny do členských proměnných instance třídy Frame. Při využití popsané techniky se funkce pro zpracování událostí stanou metodami aplikační třídy a každá z těchto metod může přistupovat k libovolným datovým členům aplikace (ačkoliv v tomto případě žádné datové členy nevytváříme) prostřednictvím reference self. Tím zajistíme přirozené propojení prvků grafického uživatelského rozhraní s ostatními aplikačními objekty:

```
from Tkinter import *

class AplikaceVymazat:
def __init__(self, rodic=0):
self.hlavniOkno = Frame(rodic)
# Vytvoříme widget třídy Entry
self.vstup = Entry(self.hlavniOkno)
self.vstup.insert(0, "Ahoj, vy tam!")
self.vstup.pack(fill=X)

# Nyní přidáme dvě tlačítka a použijeme efekt drážky.
fTlacitka = Frame(self.hlavniOkno, border=2, relief=GROOVE)
bVymazat = Button(fTlacitka, text="Vymazat",
width=8, height=1, command=self.vymazatText)
bKonec = Button(fTlacitka, text="Konec",
width=8, height=1, command=self.hlavniOkno.quit)
bVymazat.pack(side=LEFT, padx=15, pady=1)
bKonec.pack(side=RIGHT, padx=15, pady=1)
fTlacitka.pack(fill=X)
self.hlavniOkno.pack()

# Nastavíme nadpis okna.
self.hlavniOkno.master.title("Vymazat")

def vymazatText(self):
self.vstup.delete(0, END)

aplikace = AplikaceVymazat()
aplikace.hlavniOkno.mainloop()
```

Výsledek vypadá takto:



Stojí za povšimnutí, že výsledek výrazně připomíná předchozí verzi příkladu. Trochu jsme upravili spodní rámeček, aby získal pěknější podobu s drážkou okolo. Nastavili jsme také šířky tlačítek, abychom se přiblížili vzhledu, který bude mít další příklad, využívající nadstavbu wxPython.

Do podoby objektu samozřejmě můžeme zabalit nejen hlavní aplikaci. Mohli bychom vytvořit třídu s prvkem typu Frame, který obaluje standardní sadu tlačítek. Tu pak můžeme využívat například při vytváření dialogových oken. Mohli bychom dokonce vytvořit třídy pro celé dialogy a ty pak používat v několika projektech. Nebo bychom mohli rozšířit schopnosti standardních prvků typu widget definicí odvozených tříd. Například bychom mohli vytvořit tlačítko, které mění barvu v závislosti na svém stavu. Něco takového provádí modul *Python Mega Widgets* (PMW), což je rozšíření Tkinter — PMW si můžete stáhnout (download).

Alternativa jménem wxPython

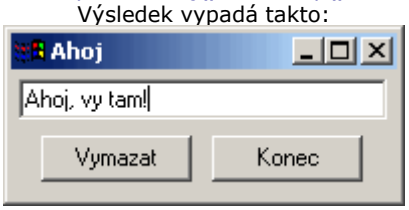
Pro práci s grafickým uživatelským rozhraním je k dispozici mnoho dalších nástrojů (toolkit), ale jedním z nejpopulárnějších je wxPython. Ten je pro změnu pythonovskou obálkou kolem nástroje wxWindows pro jazyk C++. Z obecného hlediska je wxPython mnohem typičtějším nástrojem pro práci s grafickým uživatelským rozhraním, než je Tkinter. V základní podobě také poskytuje více standardní funkčnosti, než Tk. Poskytuje prvky jako tooltip ([túltip]; *bublina* s textem pro prvek ležící pod kurzorem myši), stavová lišta (status bar) a další, které si v Tkinter musíte vytvořit sami. Pomocí wxPython si znovu přepíšeme dříve uvedený příklad "Ahoj, vy tam!", který používá prvky typu Label a Button.

Co se týká wxPython, nepůjdeme příliš do detailů. Pokud se chcete dozvědět více o tom, jak wxPython pracuje, budete si muset stáhnout instalační balík z webovských stránek wxPython.

Obecně se dá říci, že tato nástrojová sada (toolkit) definuje pracovní rámec (framework), který nám dovolí vytvářet okna, umísťovat do nich ovládací prvky a navazovat na ně metody, tj. definovat, které metody se mají volat pro obsluhu událostí těchto ovládacích prvků. wxPython je plně objektově orientován, takže byste pro obsluhu událostí měli používat opravdu metody a ne funkce. Příklad použití vypadá následovně:

```
from wxPython.wx import *

# --- Definujeme uživatelský rámeček (Frame), který se stane hlavním oknem. ---
class RamecekAhoj(wxFrame):
def __init__(self, rodic, ID, titulek, pozice, velikost):
wxFrame.__init__(self, rodic, ID, titulek, pozice, velikost)
# Použití panelu zajistí správné pozadí.
panel = wxPanel(self, -1)

# Nyní vytvoříme text a tlačítka.
self.tAhoj = wxTextCtrl(panel, -1, "Ahoj, vy tam!", (3,3), (185,22))
tlacitko = wxButton(panel, 10, "Vymazat", (15, 32))
tlacitko = wxButton(panel, 20, "Konec", (100, 32))

# Nyní svážeme tlačítka s obslužnými metodami.
EVT_BUTTON(self, 10, self.OnVymazat)
EVT_BUTTON(self, 20, self.OnKonec)

# Následují naše metody pro obsluhu událostí.
def OnVymazat(self, udalost):
self.tAhoj.Clear()

def OnKonec(self, udalost):
self.Destroy()

# --- Definujeme aplikační objekt. ---
# Poznamenejme, že všechny wxPythonovské programy MUSÍ definovat
# třídu aplikačního objektu jako třídu odvozenou od wxApp.
class AplikaceAhoj(wxApp):
def OnInit(self):
frame = RamecekAhoj(NULL, -1, "Ahoj", (200,50), (200,90))
frame.Show(True)
# self.setTopWindow(frame)
return True

# Vytvoříme instanci třídy a spustíme smyčku zpráv.
AplikaceAhoj().MainLoop()
```

Výsledek vypadá takto:



Za povšimnutí stojí používání konvence pro pojmenování metod, které mají být volány z rámce (framework) wxPython — OnXxxx. (Předložku *On* bychom pro tento případ mohli doslova překládat jako *Při*.) Povšimněte si také funkcí EVT_XXX, kterými se definuje vazba na události prvků. (Zkratka EVT pochází z anglického *event*[ivent], tj. událost.) Podobných funkcí existuje celá rodina. Systém wxPython využívá celou řadu ovládacích prvků (widget) — mnohem více, než je tomu u Tkinter. Lze jimi realizovat poměrně náročná grafická uživatelská rozhraní. Naneštěstí se u nich používá převážně rozmisťovací schéma založené na souřadnicích, které budete již po chvíli vnímat jako velmi únavné. Existuje sice možnost použití schématu, které se velmi podobá tkinterovskému správci rozložení zvanému *packer*, ale tento prostředek není příliš dobře dokumentován. Pro tvorbu grafického uživatelského rozhraní existuje komerčně dostupný nástroj. Doufejme, že se brzy objeví i nějaká zdarma dostupná alternativa.

Za zmínku stojí to, že posledně uvedený příklad a velmi podobný, dříve uvedený příklad psaný v Tkinter, mají přibližně stejný počet řádků (v anglickém originále jich je 19 pro Tkinter a 20 pro wxPython — pokud nepočítáme komentářové a prázdné řádky. V českém překladu jsem se o dosažení přesně stejného počtu řádků nesnažil.)

Shrneme-li to, pak v případě, kdy chcete k nějakému textově orientovanému nástroji rychle vytvořit jednoduché grafické uživatelské rozhraní, pak by měl Tkinter vyhovět vašim požadavkům při současné minimalizaci nutného úsilí. Pokud chcete vytvářet aplikace s plnohodnotným grafickým uživatelským rozhraním, které mají být použitelné na více platformách, pak byste se měli blíže seznámit s wxPython.

Mezi další nástroje pro budování grafického uživatelského rozhraní patří MFC, .NET a jsou zde samozřejmě letité *curses*, což je vlastně grafické uživatelské rozhraní realizované v textovém prostředí. **Poznámka k .NET.**

Poznámka překladatele: Knihovna *curses* Využívá možností textového režimu zobrazovacích adaptérů, kdy lze předepisovat zobrazování znaků na daných pozicích textové obrazovky, určení barev takto zobrazeného textu, a další. Pokud si pod tímto popisem nedovedete nic představit, vzpomeňte si na klasickou verzi aplikace Norton Commander, jeho kvalitního windowsovského soupeře zvaného FAR, případně linuxovskou variantu zvanou Midnight Commander (mc). Můžete si představit i libovolnou *klasickou* dosovou aplikaci, která používala okénka tvořená z rámečkových znaků. Aplikace s podobným vzhledem vznikaly dříve, než se objevily první verze Windows. Knihovna *curses* ale má svůj původ v unixovém světě, z jehož promyšlených abstrakcí tvůrci Windows velmi často čerpají. Někdy to jde tak daleko, že někteří napůl žertem říkají, že až budou jednou MS Windows dokončené, bude to nejlépe dokumentovaný Unix na světě.

Řadu věcí, které jsme se naučili v souvislosti s Tkinter, lze aplikovat na všechny ze zmíněných prostředků pro tvorbu grafického uživatelského rozhraní. Každý z nich má ale své charakterické vlastnosti, zvláštnosti, podivnosti a neduhy. Vyberte si některý z nich, naučte se jej a užívejte si bláznivého světa návrhu grafického uživatelského rozhraní. Na závěr bych se měl zmínit, že pro řadu těchto nástrojů existují grafické prostředky pro návrh a tvorbu uživatelského rozhraní. Jako příklad uveďme Blackadder pro Qt a Glade pro GTK. Pro wxPython se o zjednodušení procesu výstavby grafického uživatelského rozhraní snaží prostředek zvaný Python Card.

To nám prozatím stačí. Nechceme zde vytvářet novou referenční příručku pro Tkinter. Cílem bylo pouze uvedení nezbytných věcí k tomu, abyste mohli učinit první kroky. Odkazy na další zdroje informací o Tkinter naleznete v sekci Tkinter na webovských stránkách systému Python.

Problematikou používání Tcl/Tk se také zabývá několik knih. Přinejmenším jedna se věnuje přímo Tkinter. K Tkinter se nicméně vrátíme v případové studii, kde si ukážeme jeden ze způsobů, jak obalit program s dávkovým charakterem grafickým uživatelským rozhraním. Tím se docílí zlepšení použitelnosti původního programu.

Rekurze

Poznámka: Téma *rekurze* je určeno pro poměrně pokročilé programátory. Při tvorbě většiny aplikací o rekurzi nemusíte vědět vůbec nic. Ale občas je užitečnost použití rekurze přímo neocenitelná, proto si zde o ní něco řekneme. Pokud vám to v prvním okamžiku nebude dávat smysl, nepropadejte panice.

Co to vlastně je?

V předchozích částech učebnice jsme se zmínili o tom, že použití cyklu patří k jednomu ze základních kamenů programování. Navzdory tomuto tvrzení je ve skutečnosti možné vytvářet programy, které nepoužívají přímo vyjádřenou konstrukci cyklu. V některých jazycích, jako je například Lisp, přímá konstrukce cyklu jako FOR, WHILE, a dalších ve skutečnosti vůbec neexistuje. Místo toho se zde používá technika známá jako *rekurze*. Pro řešení některých typů problémů se rekurze ukazuje být velmi mocnou technikou. Proto se na ni teď podíváme.

Rekurzí jednoduše rozumíme použití nějaké funkce jako části definice té samé funkce. Takže o definici akronymu GNU (což je zdroj velkého množství softwarových produktů dostupných zdarma) říkáme, že je rekurzivní, protože zkratka GNU znamená *GNU's Not Unix* (čili *GNU Není Unix*). Zkratka GNU je tedy součástí definice významu zkratky samé.

Klíčem k fungování rekurze je to, že **musí existovat podmínka ukončení** taková, že v určité situaci funkce pokračuje větví, která představuje nerekurzivní řešení. (Definice akronymu GNU tímto testem použitelnosti neprojde, protože vede k nekonečnému cyklu.)

Poznámka překladatele: Možná jste se někdy setkali s žertovnou podobou vysvětlení nekonečného cyklu, jak by mohl být uveden ve výkladovém slovníku:

Cyklus nekonečný
Viz *Nekonečný cyklus*.
Nekonečný cyklus
Viz *Cyklus nekonečný*.

Taková definice nekonečného cyklu je vlastně generována rekurzí. Pokud hesla *Cyklus nekonečný* a *Nekonečný cyklus* budeme považovat za funkce a jejich část *Viz xyz* za volání jiné funkce, pak jsme vytvořili nekonečný cyklus za použití takzvané *nepřímé rekurze*. Ta se od výše zmíněné rekurze liší pouze tím, že k volání funkce samé nedochází přímo, ale zprostředkovaně, jinou funkcí. Aby nepřímá rekurze byla k něčemu dobrá, musí být rovněž splněn předpoklad, že *v určitém bodě musí nastat nerekurzivní dořešení problému*.

Podívejme se na jednoduchý příklad. Matematická funkce *faktoriál* je definována jako součin všech čísel až po zadaný argument včetně. Faktoriál čísla 1 (jedna) je definován jako 1. Pokud se nad tím zamyslíme, pak zjistíme, že totéž můžeme vyjádřit jiným způsobem: faktoriál čísla N je roven N krát faktoriál čísla (N-1). Takže můžeme psát:

```
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 2! x 3 = 6
N! = 1 x 2 x 3 x .... (N-2) x (N-1) x N = (N-1)! x N
```

V jazyce Python to můžeme zapsat takto:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Protože v každém kroku snižujeme hodnotu čísla N a testujeme shodu na hodnotu 1, musí funkce skončit. V uvedené definici funkce je ovšem malá chyba. Pokud se ji pokusíte zavolat pro číslo menší než jedna, uvede se do nekonečného cyklu. Opravit to můžeme tím, že místo testu na rovnost použijeme operátor <=. Tento příklad ukazuje, jak snadno můžeme při zápisu podmínky ukončení udělat chybu. U rekurzivních funkcí jde pravděpodobně o jednu z nejběžnějších chyb. Abyste zajistili jejich správnou funkčnost, ujistěte se, že testujete všechny hodnoty v okolí bodu ukončení rekurze.

Podívejme se, co se děje, když funkci spustíme. Povšimněte si, že příkaz return vrací n * (*výsledek následujícího volání funkce factorial*), takže dostáváme:

```
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1
```

V tomto okamžiku se Python vrací do vyšších úrovní a dosazuje hodnoty:

```
factorial(2) = 2 * 1 = 2
factorial(3) = 3 * 2 = 6
```

```
factorial(4) = 4 * 6 = 24
```

Zápis funkce pro výpočet faktoriálu bez použití rekurze ve skutečnosti není tak obtížný. Vyzkoušejte si jej v rámci cvičení. V podstatě potřebujete provést cyklus přes všechna čísla až do N a během této činnosti provádíte násobení. Ale později uvidíme, že existují funkce, jejichž zápis je bez použití rekurze mnohem obtížnější (ve srovnání s rekurzivním zápisem).

Rekurzivní průchod seznamem

Jinou oblastí, kdy je použití rekurze velmi užitečné, je zpracování seznamů. Rekurzi můžeme snadno použít za předpokladu, že jsme schopni testovat prázdnost seznamu a generovat zeznam bez první položky. Pro získání části seznamu můžeme v jazyce Python použít techniku, které se říká *slicing* ([slajsing] = odkrajování, odřezávání). Plné vysvětlení naleznete v příručce jazyka Python. Pro naše účely postačí, když budeme vědět, že se při použití indexu ve tvaru [1:] vrací kopie všech prvků seznamu prvku na idexu 1 až do konce seznamu. Takže pokud chceme získat první prvek seznamu L, napíšeme:

```
prvni = L[0] # použijeme prostě normální indexování
```

A pokud chceme získat kopii zbytku seznamu, napíšeme:

```
zbytek = L[1:] # vyřízneme prvky na indexech 1, 2, 3, ...
```

Abyste se ujistili, že to funguje, napište na vyzývací řádek Pythonu následující:

```
>>> L = [1, 2, 3, 4, 5]
>>> print L[0]
1
>>> print L[1:]
[2, 3, 4, 5]
```

Poznámka překladatele: Obecně zápis Seznam[od:do] vyřízne z původního seznamu prvky počínaje indexemod až po prvek s indexem o jedno menším, než do. Pokud část od nebo do není uvedena, pak se do neuvedené části vnitřně doplní index, který zahrne prvky od začátku nebo do konce seznamu. Pří získávání části seznamu dochází ke kopii prvků. Zápis Seznam[:] se používá jako obrat pro získání kopie seznamu. Prostým přiřazením s2 = Seznam získáme pouze odkaz na originální seznam. Pokud změníme s2, změní se i Seznam. Požadavek na získání kopie seznamu proto nepatří k nějak výjimečným.

Nyní se vraťme k použití rekurze pro tisk seznamů. Uvažujme jednoduchý případ, kdy chceme každý prvek seznamu řetězců vytisknout voláním funkce tiskSeznamu:

```
def tiskSeznamu(Seznam):
    if Seznam:
        print Seznam[0]
        # Podrobnosti k [1:] — viz příručka jazyka Python, 'slicing'.
        tiskSeznamu(Seznam[1:])
```

Pokud je seznam neprázdný — dotaz na neprázdný seznam v boolovském kontextu vrací hodnotu True —, pak vytiskneme obsah prvního prvku seznamu a zpracujeme stejným způsobem zbytek seznamu takto (nepythonovský pseudo kód):

```
Jsme uvnitř tiskSeznamu([1,2,3])
tiskne se [1,2,3][0] => 1
spouští se tiskSeznamu([1,2,3][1:]) => tiskSeznamu([2,3])
=> nyní jsme uvnitř tiskSeznamu([2,3])
tiskne se [2,3][0] => 2
spouští se tiskSeznamu([2,3][1:]) => tiskSeznamu([3])
=> nyní jsme uvnitř tiskSeznamu([3])
tiskne se [3][0] => 3
spouští se tiskSeznamu([3][1:]) => tiskSeznamu([])
=> nyní jsme uvnitř tiskSeznamu([])
Podmínka v "if Seznam" není pro prázdný seznam splněna,
proto se vracíme z funkce (nejhlubší bod rekurze).
=> jsme zpět v tiskSeznamu([3])
narazili jsme na konec funkce a vracíme se
=> jsme zpět v tiskSeznamu([2,3])
narazili jsme na konec funkce a vracíme se
=> jsme zpět v tiskSeznamu([1,2,3]), tj. na nejvyšší úrovni
narazili jsme na konec funkce a vracíme se
```

Poznámka: Výše uvedené vysvětlení je s úpravami převzato z textu, který uvedl Zak Arntson v mailing listu "Python Tutor" v červenci 2003.

V případě jednoduchého seznamu lze totéž jednoduše řešit při použití jednoduchého cyklu for. Ale uvažujme, jak by to vypadalo v případě, kdyby byl Seznam složitější a mohl uvnitř obsahovat další seznamy. (Prvkem by mohl být buď řetězec nebo další seznam.) Pokud jsme schopni testovat, zda je prvek seznamu dalším seznamem, pak zavoláme funkci tiskSeznamu() rekurzivně. Pokud prvek není seznamem, pak jej jednoduše vytiskneme. Vyzkoušejme si to:

```
def tiskSeznamu(Seznam):
    # Jestliže je seznam prázdný, nedělej nic.
    if not Seznam: return
    # Pokud je první prvek seznamem, dosadíme jej do tiskSeznamu().
    if type(Seznam[0]) == type([]):
        tiskSeznamu(Seznam[0])
    else: # První prvek není seznam. Jednoduše jej vytiskneme.
        print Seznam[0]
    # Nyní zpracujeme zbytek Seznamu.
    tiskSeznamu(Seznam[1:])
```

Pokud se totéž pokusíte zapsat pomocí cyklu, zjistíte, že je to velmi obtížné. Rekurze umožní zapsat podobněsložité úlohy srovnatelně jednoduše. (Jinými slovy, rekurzivní řešení jednoduchého případu a uvedeného složitějšího případu je přibližně stejně obtížné.)

Je tu ovšem jeden zádrhel (jako vždycky). Rekurzivní zpracování velkých datových struktur typicky vede k velké spotřebě paměťového prostoru. Takže pokud máte k dispozici málo paměti nebo zpracováváte velmi velké datové struktury, může být složitější konvenční řešení bezpečnější.

Tak. A nyní udělejme další skok do neznáma — seznámíme se s *objektově orientovaným programováním*.

Funkcionální programování

V této kapitole se podíváme na to, jak jazyk Python podporuje další styl programování, takzvané *funkcionální programování*. Podobně jako u rekurze jde vyloženě o téma pro pokročilé, které v tomto okamžiku můžete jednoduše ignorovat. Techniky funkcionálního programování mají své použití i v každodenní programátorské praxi. Zastánci funkcionálního programování věří, že představuje od základů lepší způsob vývoje software.

Co to vlastně je funkcionální programování?

Funkcionální programování by se nemělo zaměňovat s příkazovým (nebo také procedurálním) programováním. Nepodobá se ani objektově orientovanému programování. Jde o něco jiného. Ne sice *zcela* jiného, protože koncepty, se kterými budeme pracovat, patří k známým konceptům programování. Jsou pouze vyjádřeny jiným způsobem. Mírně se liší i základní filozofie způsobu používání těchto konceptů při řešení problémů.

Funkcionální programování je založené na *výrazech* (ve smyslu matematickém). Funkcionální programování bychom vlastně mohli popsat termínem *programování orientované na výrazy*, protože se zde vše redukuje právě na výrazy. Asi byste si mohli vzpomenout, že výraz je složen z operací a proměnných takovým způsobem, že jeho výsledkem je jedna hodnota. Takže například x == 5 je boolovský výraz. Zápis 5 + (7 - y) představuje aritmetický výraz. A u zápisu string.search("Ahoj, vy tam!", "Aho") jde o řetězcový výraz. V posledním uvedeném případě jde současně o volání funkce v rámci modulu string. Jak později uvidíme, hrají funkce v rámci funkcionálního programování velmi důležitou roli. (To vás asi podle jména už napadlo.)

S funkcemi se ve funkcionálním programování zachází jako s objekty. To znamená, že jsou uvnitř programu často předávány podobným způsobem, jako se předávají proměnné. Příklady jsme viděli v našich programech s grafickým uživatelským rozhraním, kde jsme jméno funkce přiřazovali atributu command ovládacího prvku Button (tlačítko). S funkcí pro obsluhu události jsme zacházeli jako s objektem a odkaz na ni jsme předávali prvku tlačítka. Myšlenka předávání funkcí v rámci našeho programu je pro funkcionální programování myšlenkou klíčovou.

Funkcionální programy bývají rovněž silně orientovány na práci se seznamy.

Závěrem rekněme, že se funkcionální programování snaží soustředit spíše na *co*, než *jak* chceme řešit. To znamená, že funkcionální program by měl spíše popisovat řešený problém, než aby se soustředil na vlastní mechanismus řešení. Jazyků, které o sobě tvrdí, že pracují tímto způsobem, existuje několik. Jedním z nejrozšířenějších je Haskell. Na jeho webovém serveru (www.haskell.org) naleznete řadu článků, které popisují jak filosofii funkcionálního programování, tak samotný jazyk Haskell. (Podle mého osobního mínění jsou zmíněné cíle, jakkoliv jsou chvályhodné, zastánci funkcionálního programování poněkud přeceňovány.)

Struktura čistě funkcionálního programu je definována výrazem, který zachycuje požadovaný cíl. Každý *term*(viz následující poznámka) daného výrazu je zase vyjádřením určité charakteristiky řešeného problému. Vyhodnocením každého z těchto termů nakonec dospějeme k řešení.

Poznámka překladatele: Term neboli terminální podvýraz je v rámci zápisu výrazu jeho dále nedělitelná součást. Můžeme si zde představit volání pojmenované funkce.

Dobrá. To je tedy teoretický pohled. A funguje to vůbec? Ano, někdy to funguje velmi dobře. Pro řešení určitých typů problémů jde o přirozenou a mocnou techniku. Pro řadu dalších problémů tento přístup naneštěstí vyžaduje poměrně abstraktní způsob myšlení, který je velmi ovlivněn matematickými principy. Výsledný kód je pro laického programátora často velmi nečitelný. Ale výsledný kód bývá často mnohem kratší a spolehlivější, než odpovídající kód zapsaný příkazovým stylem. A právě posledně zmiňované kvality přitáhly ke studiu funkcionálního programování mnoho programátorů, kteří používali procedurální nebo objektově orientovaný způsob programování. Dokonce i když se mu nebudete věnovat s plným nasazením, poskytuje funkcionální programování několik mocných nástrojů, které mohou využívat všichni.

Jak se k tomu staví jazyk Python?

Python poskytuje několik funkcí, které umožňují uplatnění funkcionálního přístupu k programování. Tyto funkce patří k rysům, které byly zavedeny hlavně kvůli usnadnění práce, protože bychom je v jazyce Python mohli docela snadno zapsat sami. Mnohem důležitější je ovšem přímé vyjádření *záměru*, dané jejich existencí. Jde zejména o záměr poskytnout programátorovi v jazyce Python možnost práce ve stylu funkcionálního programování — pokud si přeje jej využít.

Podíváme se na některé z těchto funkcí a uvidíme, jak budou pracovat s některými vzorovými funkcemi, které definujeme jako:

```
spam = ['pork', 'ham', 'spices']
numbers = [1, 2, 3, 4, 5]

def eggs(item):
    return item
```

map(funkce, posloupnost)

Tato funkce aplikuje pythonovskou funkci funkce na každý z členů posloupnosti posloupnost. Mějme výraz:

```
L = map(eggs, spam)
print L
```

Výsledem je nový seznam (v tomto případě totožný se seznamem spam), který je vrácen v L.

Stejného efektu bychom mohli dosáhnout zápisem:

```
for i in spam:
    L.append(i)
print L
```

Ale povšimněte si skutečnosti, že použitím funkce map() odstraníme potřebu zápisu vnořeného bloku kódu. Z určitého pohledu tím snižujeme složitost programu o jednu úroveň. Uvidíme, že jde o jakési opakující se téma funkcionálního programování. Při používání těchto funkcí se relativní složitost kódu snižuje odstraňováním bloků.

filter(funkce, posloupnost)

Jak již jméno napovídá, funkce filter() vybírá všechny prvky posloupnosti posloupnost, pro které funkcevrací hodnotu True. Uvažujme náš seznam čísel numbers. Pokud chceme vytvořit nový seznam, který se skládá pouze z lichých čísel, pak jej můžeme vytvořit takto:

```
def jeLiche(n): return (n%2 != 0) # použijeme operátor modulo
L = filter(jeLiche, numbers)
print L
```

Řešení bychom mohli napsat i takto:

```
def jeLiche(n): return (n%2 != 0)
for i in numbers:
```

```
if jeLiche(i):
    L.append(i)
print L
```

Opět si povšimněte, že konvenční kód vyžaduje k dosažení stejného výsledku použití dvou úrovní odsazení. A opět, zvýšení úrovně odsazení je příznakem zvýšené složitosti kódu.

reduce(funkce, posloupnost)

Význam funkce reduce() je poněkud méně zřejmý. Tato funkce redukuje seznam na jedinou hodnotu tím, že jeho prvky kombinuje pomocí dodané funkce. Mohli bychom například sečíst hodnoty prvků seznamu a vrátit hodnotu celkového součtu takto:

```
def secti(i, j): return i + j
print reduce(secti, numbers)
```

Stejně jako v předchozích případech bychom mohli použít konvenční zápis:

```
vysledek = 0
for i in range(len(numbers)): # použijeme indexování
    vysledek = vysledek + numbers[i]
print vysledek
```

I když v tomto případě bude výsledek stejný, vždycky to není tak přímočaré. Funkce reduce() *ve skutečnosti*dělá to, že volá dodanou funkci, přičemž jí předává první dva členy posloupnosti. Výsledek uloží místo nich. Jinými slovy, přesnější reprezentace funkce reduce() vypadá takto:

```
def reduce(numbers):
    L = numbers[:]        # vytvoř kopii originálu
    while len(L) >= 2:
        i, j = L[0], L[1] # použijeme násobné přiřazení
        L = [i+j] + L[2:] # součet prvních dvou a zbytek
    return L[0]
```

Opět vidíme, že technika funkcionálního programování redukuje složitost kódu odstraněním nutnosti použití odsazených bloků kódu.

lambda

Jedním z rysů, kterého jste si v dosud uvedených příkladech mohli všimnout, je skutečnost, že funkce, které předáváme jako argument funkcí pro podporu funkcionálního programování, bývají velmi krátké. Často jde o jediný řádek kódu. Abychom si ušetřili námahu při definování velkého množství velmi malých funkcí, nabízí nám Python další pomocnou funkci pro podporu funkcionálního programování — lambda.

Pojem *lambda výraz* se používá pro referenci na *anonymní funkci*, to znamená pro blok kódu, který lze provést stejným způsobem, jako kdyby se jednalo o funkci, která ovšem nemá jméno. Lambda výrazy mohou být definovány uvnitř programu všude tam, kde se může vyskytovat legální pythonovský výraz. To znamená, že je můžeme používat i uvnitř zmiňovaných pomocných funkcí pro podporu funkcionálního stylu programování.

Zápis lambda výrazu vypadá takto:

```
lambda <seznam parametrů> : <pythonovský výraz používající parametry>
```

Takže výše zmíněnou funkci secti bychom mohli přepsat jako:

```
secti = lambda i, j: i + j
```

Použití řádku s definicí funkce se můžeme úplně vyhnout zápisem lambda výrazu přímo uvnitř volání funkce reduce():

```
print reduce(lambda i, j: i+j, numbers)
```

Podobným způsobem můžeme přepsat naše příklady používající funkce map a filter:

```
L = map(lambda i: i, spam)
print L

L = filter(lambda i: (i%2 != 0), numbers)
print L
```

List Comprehension

Poznámka překladatele: Pojem *list comprehension* (čti list komprihenšn) je obtížně přeložitelný tak, aby byl výsledek překladu krátký a přesný. Je založen na jednom z významů slova *comprehension*, pro který existuje synonymum *comprise* (čti kemprais). Vyjadřuje skutečnost *něco obsahovat*, *být z něčeho poskládán, složen*. Slovo *list* se zde jednoznačně překládá jako *seznam*. Pojem *list comprehension* se používá pro syntaktickou konstrukci, která předepisuje způsob vygenerování seznamu z jiné kolekce. Vulgárně bychom jej tedy mohli přeložit jako *sestavovač seznamů*. V odborné terminologii by se možná dal použít pojem *funkcionální konstruktor seznamu*. Ale raději zůstaňme u originálního pojmu. Smiřte se s tím, že *naučit se programovat* do značné míry znamená také *učit se anglicky*.

Mechanismus *list comprehension* představuje techniku pro vytváření nových seznamů, která pochází z jazyka Haskell a v jazyce Python byla uvedena od verze 2.0. Má poněkud zatemňující syntaxi, podobající se matematickému zápisu množin. Vypadá takto:

```
[<výraz> for <proměnná> in <kolekce> if <podmínka>]
```

Můžeme ji vyjádřit ekvivalentním zápisem:

```
L = []
for proměnná in kolekce:
    if podmínka:
        L.append(výraz)
```

Stejně jako u ostatních konstrukcí z funkcionálního programování i zde dochází k úspoře řádků a dvou úrovní odsazení.

Podívejme se na nějaké praktické příklady.

Nejdříve si vytvoříme seznam sudých čísel:

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

Tento zápis říká, že chceme seznam hodnot (n), kde se n pohybuje v rozmezí 0 až 9 a současně platí, že n je sudé (tedy n % 2 == 0).

Podmínka, uvedená na konci, může být samozřejmě nahrazena funkcí — za předpokladu, že tato funkce vrací hodnotu, kterou může Python interpretovat jako hodnotu boolovskou. Takže pokud se znovu podíváme na předchozí příklad, můžeme jej přepsat následovně:

```
>>> def jeSude(n): return ((n % 2) == 0)
```

```
>>> [ n for n in range(10) if jeSude(n) ]
[0, 2, 4, 6, 8]
```

Vytvořme nyní seznam druhých mocnit prvních pěti čísel:

```
>>> [n * n for n in range(5)]
[0, 1, 4, 9, 16]
```

Povšimněte si, že část if nemusíme povinně použít. Úvodní výraz zde má podobu n * n a používáme pro něj všechny hodnoty ze zadaného intervalu.

Na závěr použijme místo funkce range() již existující kolekci:

```
>>> hodnoty = [1, 13, 25, 7]
>>> [x for x in hodnoty if x < 10]
[1, 7]
```

Takový zápis bychom mohli použít jako náhradu za funkci filter:

```
>>> filter(lambda x: x < 10, hodnoty)
[1, 7]
```

To, který ze stylů se zdá přirozenější nebo vhodnější, je čistě subjektivní věc. Pokud budujeme novou kolekci z již existující kolekce, můžeme použít buď dříve uvedené funkce z oblasti funkcionálního programování, nebo můžeme použít *list comprehension*. Pokud vyloženě vytváříme úplně novou kolekci, pak je obvykle snazší použít *list comprehension*.

Když to shrneme, tak ačkoliv se nám tyto konstrukce mohou zdát přitažlivé, může pro dosažení požadovaného výsledku tento přístup vést k tak složitým výrazům, že je prostě jednodušší rozepsat je na tradiční pythonovské ekvivalenty. Není to žádná ostuda. Dobrá čitelnost je vždy lepší, než zatemňování věcí jen proto, abychom vypadali chytře.

Ostatní konstrukce

Zmíněné funkce jsou samozřejmě užitečné samy o sobě, ale přesto nestačí k tomu, aby v jazyce Python umožnily plně funkcionální styl programování. Funkcionálním způsobem musíme změnit, nebo alespoň nahradit, používání řídicích struktur jazyka. Jednou z cest, kterou toho můžeme dosáhnout, je využití vedlejšího efektu toho, jak Python vyhodnocuje boolovské výrazy.

Zkrácené vyhodnocování

Protože Python používá *zkrácené vyhodnocování* boolovských výrazů, lze některých vlastností těchto výrazů využít. Připomeňme si zkrácené vyhodnocování: Pokud se vyhodnocuje boolovský výraz, začíná vyhodnocování na levé straně výrazu a postupuje směrem vpravo. Vyhodnocování končí v okamžiku, kdy již pro určení konečného výsledku není další vyhodnocování nutné.

Pro zviditelnění zkráceného vyhodnocení použijeme zvláštní postup. Nejdříve nadefinujeme dvě funkce, které nám oznámí, že jsou volány, a poté vrátí hodnotu, odpovídající jejich jménům:

```
>>> def TRUE():
...     print 'TRUE'
...     return True  # True se ve verzi 2.2 tiskne jako 1
...
>>>def FALSE():
...     print 'FALSE'
...     return False # False se ve verzi 2.2 tiskne jako 0
...
```

Nyní je použijeme, abychom zjistili, jak se vyhodnocování boolovských výrazů provádí:

```
>>> print TRUE() and FALSE()
TRUE
FALSE
0
>>> print TRUE() and TRUE()
TRUE
TRUE
1
>>> print FALSE() and TRUE()
FALSE
0
>>> print TRUE() or FALSE()
TRUE
1
>>> print FALSE() or TRUE()
FALSE
TRUE
1
>>> print FALSE() or FALSE()
FALSE
FALSE
0
```

Poznámka překladatele: Boolovský typ s hodnotami False a True byl zaveden ve verzi Python 2.2. Jména False a True však pouze pojmenovávala konstanty 0 a 1, což se projevovalo při tisku. Předchozí verze jazyka Python boolovský typ vůbec neznaly a místo boolovských hodnot se používaly hodnoty 0 a 1, jako například v jazyce C. Řada programátorů si přesto definovala konstanty False a True. Od verze 2.3 se boolovský typ stal plnohodnotným zabudovaným typem jazyka. Místo hodnot 0 a 1 příkaz print vytiskne False nebo True. Od verze 2.3 tedy bude výstup vypadat takto:

```
>>> print TRUE() and FALSE()
TRUE
FALSE
False
>>> print TRUE() and TRUE()
TRUE
TRUE
```

```
True
>>> print FALSE() and TRUE()
FALSE
False
>>> print TRUE() or FALSE()
TRUE
True
>>> print FALSE() or TRUE()
FALSE
TRUE
True
>>> print FALSE() or FALSE()
FALSE
FALSE
False
```

Povšimněte si, že u *logického součinu* (operátor and) dochází k vyhodnocování druhé části výrazu *tehdy a jen tehdy, když* nabývá první část výrazu hodnoty True (pravda). Pokud první část nabývá logické hodnoty False (nepravda), pak se druhá část nevyhodnocuje, protože výraz jako celek již *nemůže* nabýt hodnoty True.

Podobně je tomu u *logického součtu* (operátor or) v případě, kdy první část nabývá hodnoty True. V takovém případě již druhá část nemusí být vyhodnocována, protože celek *musí* nabývat hodnoty True.

Při vyhodnocování boolovského výrazu se uplatňuje ještě jeden rys, kterého můžeme využít. Jde o to, že při vyhodnocování výrazu v boolovském kontextu Python nevrací jednoduše hodnotu 1 nebo 0 (nebo True čiFalse). Místo toho se vrací skutečná hodnota výrazu. Takže pokud testujeme řetězec na prázdnost (prázdný řetězec se chápe jako False) takto:

```
if "Tento retezec neni prazdny": print "Neni prazdny"
else: print "Prazdny retezec"
```

... Python prostě vrací testovaný řetězec.

Této vlastnosti můžeme využít k realizaci chování, které se podobá větvení. Dejme tomu, že máme například následující úsek kódu:

```
if TRUE(): print "Je to pravda (True)"
else: print "Je to nepravda (False)"
```

Můžeme jej nahradit konstrukcí ve stylu funkcionálního programování:

```
V = (TRUE() and "Je to pravda (True)") or ("Je to nepravda (False)")
print V
```

Zkuste tento příklad provést a poté nahraďte volání TRUE() voláním FALSE().

To znamená, že jsme při využití zkráceného vyhodnocování boolovských výrazů nalezli způsob, jak z našich programů odstranit konvenční příkazy if/else. Možná si vzpomínáte, že jsme u tématu rekurze zjistili, že rekurzi můžeme použít k nahrazení konstrukce cyklu. Takže kombinací těchto dvou efektů můžeme z našeho programu odstranit všechny konvenční řídicí struktury a nahradit je čistými výrazy. Jde o velký krok směrem možnosti řešení problémů čistě ve stylu funkcionálního programování.

Abychom to vše ukázali na praktickém příkladě, napišme si čistě funkcionálním stylem program pro výpočet faktoriálu, který používá rekurzi místo cyklu a zkrácené vyhodnocování výrazu místo if/else:

```
def factorial(n):
return ((n <= 1) and 1) or
(factorial(n - 1) * n)
```

To je opravdu celé. Možná to není tak dobře srozumitelné jako konvenčnější pythonovský kód, ale funguje to a jde o funkci zapsanou čistě ve funkcionálním stylu — jde tedy o čistý výraz.

Závěry

V tomto místě se možná pozastavujete nad tím, co je vlastně cílem toho všeho? A nejste sami. I když funkcionální programování přitahuje řadu teoretiků v oblasti výpočetních věd (Computer Science) — a často matematiky —, zdá se, že většina praktických programátorů používá techniky funkcionálního programování velmi zřídka. Navíc používají jakýsi hybridní přístup, kdy tyto techniky kombinují s tradičnějším, příkazovým stylem podle toho, jak to považují za vhodné.

Poznámka překladatele: Podle mého názoru může mít takové používání technik funkcionálního programování příčiny jak subjektivní, tak objektivní. Mezi subjektivní příčiny bych zařadil ten fakt, že se jedná přece jen o téma pro pokročilé. Do hloubky se s ním na vysokoškolské úrovni seznámil jen zlomek těch, kteří pracují jako praktičtí programátoři. Mezi objektivní příčiny bych zařadil skutečnost, že model, na kterém je funkcionální programování založeno, neodpovídá přesně modelu, který se používá pro výstavbu běžných počítačů. Von Neumannovská koncepce počítače s pamětí pro data a pro program a s procesorem, který vykonává instrukce, je většinou mnohem efektivněji využitelná pro programy napsané procedurálním stylem. Jinými slovy, dobře napsaný procedurální program na těchto počítačích bude typicky výkonnější, než dobře napsaný odpovídající program zapsaný ve funkcionálním jazyce. Z matematického (teoretického) pohledu je ale tato skutečnost nevýznamná. Proto se řada matematicky zaměřených programátorů, kteří typicky neřeší problémy denní praxe, otázkami skutečného výkonu příliš nezabývá. Funkcionální přístup je z hlediska matematického určitě jednodušší, lépe uchopitelný matematickými nástroji.

Pokud musíme na prvky seznamu aplikovat operace, které lze přirozeně vyjádřit pomocí map, reduce nebofilter, pak je v každém případě použijte. Občas můžete dokonce zjistit, že rekurze je v daném případě vhodnější, než konvenční cyklus. V ještě řidších případech můžete najít použití i pro zkrácené vyhodnocování místo konvenčního if/else — zejména při použití uvnitř výrazu. Tak jako je tomu u každého programátorského nástroje, nenechte se unést jen určitou filozofií. Pro řešení konkrétního zadání se raději snažte použít nejvhodnější nástroj, ať už je jakéhokoliv charakteru. Přinejmenším si buďte vědomi toho, že existují alternativy.

K otázce *lambda výrazů* zbývá poznamenat ještě jeden závěr. I když neuvažujeme ve stylu funkcionálního programování, existuje jedna oblast, ve které lze dobře použít operátor lambda. Jde o definice *funkcí pro obsluhu událostí* (event handler), se kterými se setkáváme při programování uživatelského rozhraní. Tyto funkce jsou často velmi krátké, případně se v nich volají nějaké rozsáhlejší funkce, kterým se předává několik napevno použitých hodnot argumentů. V těchto případech lze v roli funkce pro obsluhu událostí použít lambda funkci. Tím se vyhneme nutnosti definovat mnoho malých funkcí, které by nám zaplňovaly prostor jmény, použitými jen jednou. Vzpomeňte si, že příkaz lambda vrací objekt

typu *funkce*. Právě tento objekt (funkce) je předán prvku typu widget a je volán v době, kdy se vyskytne příslušná událost. Pokud si vzpomenete, jak se v Tkinter definuje prvek typu Button (tlačítko), pak můžeme lambda použít následovně:

```
def write(s): print s
b = Button(rodic, text="Stiskni mne",
command = lambda : write("Stiskl jsi mne!"))
b.pack()
```

V tomto případě bychom samozřejmě mohli dosáhnout téhož efektu tím, že bychom definovali přednastavenou hodnotu parametru funkce write(). Atributu command prvku typu Button bychom pak jednoduše přiřadili write(bez použití závorek). Nicméně i v takto jednoduchém případě nám použití lambda přináší tu výhodu, že jediná definice funkce write() může být použita pro více tlačítek tak, že jí přes lambda předáme různé řetězce. To znamená, že můžeme přidat druhé tlačítko takto:

```
b2 = Button(rodic, text="Nebo mne",
command = lambda : write("Stiskl jsi mne. (Tvoje druhe tlacitko.)"))
b2.pack()
```

Příkaz lamda můžeme použít i u techniky, kdy používáme metodu ovládacího prvku bind(), které jako argument předáváme objekt pro ošetření události:

```
b3 = Button(rodic, text="Mne taky stiskni")
b3.bind(<Button-1>, lambda ev : write("Stisknuto"))
```

To je vše, co se zde o funkcionálním programování dozvíte. Pokud se na ně chcete podívat trochu hlouběji, existuje celá řada dalších informačních zdrojů. Některé z nich jsou uvedeny níže.

Ostatní zdroje

- Na webovském serveru IBM naleznete vynikající článek Davida Mertze o funkcionálním programování v jazyce Python. Hlouběji se zabývá detaily kolem řídicích struktur a uvádí podrobnější příklady tohoto přístupu.
- Jiné jazyky podporují funkcionální programování ještě lépe, než Python. Jsou to například Lisp, Scheme, Haskel, ML a některé další. Cenné informace o funkcionálním programování naleznete zvláště na webovském serveru jazyka Haskell.
- Užitečné FAQ a nejnovější události kolem funkcionálního programování můžete nalézt v diskusní skupině comp.lang.functional.
- Na výše zmíněných místech naleznete odkazy na několik knih. Jedna z klasických knih, která se nevěnuje jen funkcionálnímu programování, ale dobře popisuje jeho principy, má název *Structure & Interpretation of Computer Programs*, jejímiž autory jsou Abelman, Sussman a Sussman. Tento text se soustředí na jazyk Scheme a na rozšířenou verzi jazyka Lisp. Mým osobním primárním zdrojem byl *The Haskell School of Expression* — autorem je Paul Hudak. Zabývá se, jak jinak, jazykem Haskell.

Pokud kdokoliv nalezne dobrý odkaz, pošlete mi zprávu prostřednictvím níže uvedeného odkazu. (Poznámka překladatele: Můžete použít odkaz z anglického originálu, ale i z českého překladu. V druhém případě jej autorovi předám.)

Případová studie

V této případové studii rozšíříme funkčnost programu pro počítání slov, který jsme vyvinuli již dříve. Vytvoříme program, který napodobí funkci unixovského programu wc v tom smyslu, že bude vypisovat počet řádků, slov a znaků v souboru. Ale půjdeme ještě dál a budeme vypisovat také počet vět, klauzulí (viz poznámka dále), slov, písmen a interpunkčních znamének v textovém souboru. Vývoj programu budeme provádět po etapách. Postupně budeme zvyšovat jeho schopnosti. Převedeme jej do podoby modulu, abychom zvýšili jeho znovupoužitelnost. A upravíme jeho implementaci do objektově orientované podoby, čím zvýšíme možnosti dalšího rozšiřování funkčnosti.

Implementovat jej budeme v jazyce Python, ale přinejmenším počáteční fáze mohou být napsány i v jazycích BASIC nebo Tcl. S tím, jak budeme řešit složitější části problému, budeme stále více používat zabudované datové struktury jazyka Python. Proto se bude obtížnost případného zápisu v jazyce BASIC zvyšovat, ačkoliv použití Tcl bude stále možné. Objektově orientované stránky konečného řešení budou vhodné pouze pro jazyk Python.

Jako cvičení pro čtenáře bude ponechána možnost implementace dalších rysů, jako jsou:

- Výpočet *FOG indexu* pro daný text, kde FOG index můžeme přibližně definovat jako ((průměrný počet slov na větu) + (procento slov s více než 5 písmeny) * 0.4). Toto číslo vyjadřuje složitost textu.
- Zjištění počtu používaných slov a frekvence jejich použití.
- Vytvoření nové verze, která analyzuje RTF soubory.

Počítání řádků, slov a znaků

Podívejme se znovu na dříve vytvořený program pro počítání slov (viz Práce se soubory — Počítání slov):

```
import string
def pocetSlov(s):
seznam = string.split(s)
# Poznámka překladatele: Od verze jazyka Python 2
# je pro standardní řetězec definována metoda split(), takže místo výše
# uvedeného zápisu můžeme psát s.split() a není nutné provádět import
# modulu string.
return len(seznam)  # vrátíme počet prvků seznamu

vstup = open("menu.txt", "r")
celkem = 0  # vytvoříme proměnnou a nastavíme jí počáteční hodnotu nula

for radek in vstup.readlines():
celkem = celkem + pocetSlov(radek) # sečti počty za každý řádek
print "Soubor má %d slov." % celkem

vstup.close()
```

Potřebujeme přidat počítadla řádků a znaků. Počítání řádků je snadné, protože cyklus zpracovává vstup po řádcích. Jednoduše přidáme nějakou proměnnou a budeme ji zvyšovat při každé obrátce cyklu. Počítadlo znaků je pouze mírně složitější, protože můžeme procházet seznam slov a jejich délky přičítat do další proměnné.

Rádi bychom také zvýšili obecnost použití programu tím, že jméno zkoumaného souboru zjistíme z parametru příkazového řádku nebo, pokud není zadáno, vyžádáme si zadání jména dotazem na uživatele. (Alternativní řešení by spočívalo ve čtení textu ze standardního vstupu, což právě dělá opravdový program wc.)

Takže konečné řešení ve stylu wc vypadá takto (poznámka překladatele: abychom se vyhnuli komplikacím s českými řetězci ve zdrojovém textu programu, zjednodušíme si řešení tím, že použijeme *cestinu bez hacku a carek*):

```
import sys, string

# Získáme jméno souboru buď z příkazového řádku
# nebo si je vyžádáme od uživatele.
if len(sys.argv) != 2:
jmenoSouboru = raw_input("Zadejte jmeno souboru: ")
else:
jmenoSouboru = sys.argv[1]

vstup = open(jmenoSouboru, "r")
# Poznámka překladatele: Od verze Python 2 by se měla dávat
# přednost zápisu vstup = file(jmenoSouboru, "r")

# Počáteční hodnoty počítadel nastavíme na nuly.
# Tím se také vytvoří příslušné proměnné.
slov = 0
radku = 0
znaku = 0

for radek in vstup.readlines():
radku = radku + 1

# Řádek rozložíme na slova a spočítáme je.
seznamSlov = string.split(radek)
# Poznámka překladatele: U Python 2 lze psát
# seznamSlov = radek.split()
slov = slov + len(seznamSlov)
# Počet znaků určíme z délky původního řádku,
# čímž započítáme mezery atd.
znaku = znaku + len(radek)

print "%s ma %d radku, %d slov a %d znaku" % (jmenoSouboru, radku, slov, znaku)
vstup.close()
```

Pokud znáte unixovský příkaz wc, pak víte, že mu můžete jméno souboru zadat v podobě masky. Tím získáte hledané údaje pro všechny soubory, které masce vyhovují, a získáte také celkový součet těchto údajů. Výše uvedený program pracuje pouze s přímo zadanými jmény souborů. Pokud chcete, aby zpracovával i soubory zadané maskou, podívejte se na modul glob. Ten vám umožní vytvořit seznam jmen vyhovujících souborů, který pak můžete zpracovat. Budete k tomu potřebovat dočasná počítadla pro každý soubor a navíc kumulativní počítadla pro celkové součty (součty součtů za jednotlivé soubory). Nebo místo toho můžete použít slovník…

Počítání vět místo řádků

Když jsem začal přemýšlet o tom, jak bychom mohli rozšířit funkčnost, abychom počítali věty a slova místo "skupin znaků" (což činíme ve výše uvedeném řešení), napadlo mě nejprve, že bychom měli ze souboru nejdříve v cyklu načíst jednotlivé řádky a pak bychom měli v cyklu zpracovat každý řádek a získat z něj slova do dalšího seznamu. Nakonec bychom měli zpracovat každé "slovo" za účelem odstranění nadbytečných znaků.

Když jsem o tom uvažoval o něco déle, začalo být zřejmé, že pokud bychom jednoduše shromažďovali slova a interpunkční znaménka, mohli bychom právě interpunkční znaménka použít pro počítání vět, klauzulí atd. (tím, že řekneme, co považujeme za větu nebo klauzuli s ohledem na použitá interpunkční znaménka). (Poznámka překladatele: V anglické gramatice se pojem *clause* (klauzule) používá ve významu větného členu, typicky hlavní a vedlejší věty. Pravidla pro výstavbu věty a pro psaní interpunkčních znamének jsou v anglickém jazyce mnohem propracovanější, takže se z nich dá strojově lépe určit stavba věty. V českém jazyce je to mnohem obtížnější. Proto od dále uvedeného programu neočekávejte zázraky.) To znamená, že stačí, když souborem projdeme pouze jednou a poté budeme procházet přes mnohem kratší seznam interpunkčních znamének. Zkusme si to načrtnout v pseudokódu:

```
pro každý řádek v souboru:
zvýšit počítadlo řádků
if je řádek prázdný:
zvýšit počítadlo odstavců
rozložit řádek na skupiny znaků

pro každou skupinu znaků:
zvýšit počítadlo skupin
odstranit interpunkční znaky a přidat do slovníku - {znak: počet}
if ve skupině nezbyl žádný znak:
zrušit skupinu
else: zvýšit počítadlo slov

počet vět = počet znaků ('.', '?', '!')
počet klauzulí = součet všech interpunkčních znaků (poněkud ubohá definice...)

vypsat počty odstavců, řádků, vět, klauzulí, skupin znaků, slov.
pro každý interpunkční znak:
vypsat počet (ze slovníku)
```

Vypadá to, že bychom mohli vytvořit asi 4 funkce, odpovídající výše uvedeným skupinám. To by nám pomohlo vybudovat modul, který by mohl být opakovaně použitelný buď jako celek nebo po částech.

Udělejme z toho modul

Klíčové funkce budou tyto: ziskejSkupinyZnaku(vstSoubor) a ziskejInerpunkci(seznamSkupin). Podívejme se, co vytvoříme na základě uvedeného pseudokódu…

```
##############################
# Modul:    gramatika
# Vytvořil: A.J. Gauld, 2000,8,12
#
# Funkce:
# Počítá odstavce, řádky, věty, 'klauzule', skupiny znaků, slova
# a interpunkční znaménka pro textové soubory s textem předpokládajícím
# prózu. Předpokládá se, že věty končí znaky [.!?] a odstavce jsou odděleny
# prázdným řádkem. Za 'klauzuli' se jednoduše považuje část věty, která je
# oddělena interpunkčním znakem (což je poněkud hloupá definice, ale jednoho
# dne můžeme doplnit něco lepšího).
#
# Použití: Při základním použití se čte jméno souboru z parametru a vypisují
#          se všechny získané údaje. Předpokládá se vytvoření druhého modulu,
#          který by používal zde definované funkce a poskytoval užitečnější
#          příkazy.
##############################
import string, sys

############################
# Počáteční nastavení globálních proměnných.
poc_odstavcu = 1 # Předpokládáme existenci nejméně jednoho odstavce.
poc_radku, poc_vet, poc_klauzuli, poc_slov = 0, 0, 0, 0
skupiny = []
poc_interpunkcnich_znaku = {}
alfanumericke_znaky = string.letters + string.digits

# Poznámka překladatele: Pro české texty bychom mezi
# alfanumerické znaky měli přidat i znaky s diakritickými znaménky. Věc se
# ale komplikuje tím, že bychom navíc měli uvažovat i způsob kódování
# vstupního souboru. Obecné řešení by nebylo úplně jednoduché, takže zatím
# nebudeme překlad originálního zdrojového textu z angličtiny upravovat.

koncove_znaky = ['.', '?', '!']
interpunkcni_znaky = ['&', '(', ')', '-', ';', ':', ','] + koncove_znaky
for c in interpunkcni_znaky:
    poc_interpunkcnich_znaku[c] = 0
format = """%s obsahuje:
%d odstavcu, %d radku a %d vet.
Ty obsahuji %d klauzuli a celkem %d slov."""


############################
# Nyni nadefinujeme funkce, které tvoří jádro činnosti.

def ziskejSkupinyZnaku(vstSoubor):
    pass

def ziskejInerpunkci(seznamSkupin):
    pass

def vypsatStatistiky():
    print format % (sys.argv[1], poc_odstavcu,
                    poc_radku, poc_vet,
                    poc_klauzuli, poc_slov)

def Analyzuj(vstSoubor):
    ziskejSkupinyZnaku(vstSoubor)
    ziskejInerpunkci(skupiny)
    vypsatStatistiky()


# Pokud je modul volán z příkazového řádku, zajisti spuštění následujícího
# kódu. (V tomto případě je magická proměnná __name__ nastavena na hodnotu
# "__main__".)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print "Pouziti: python gramatika.py <soubor>"
        sys.exit()
    else:
        Dokument = open(sys.argv[1], "r")
```

```
# Poznámka překladatele: Od verze Python 2 by se měla dávat
# přednost zápisu Dokument = file(sys.argv[1], "r")
Analyzuj(Dokument)
Dokument.close()
```

V tomto místě jsem nechtěl ukázat celé řešení v podobě jednoho dlouhého výpisu. Proberu raději uvedenou kostru a potom se podíváme na každou z uvedených tří významných funkcí. Nicméně k tomu, abyste program uvedli do chodu, budete muset na konci vše slepit dohromady.

První věcí, která stojí za povšimnutí, je komentář na začátku. Jeho uvedení patří k běžným praktikám. Má čtenáři naznačit, co soubor obsahuje a jak má být používán. Užitečná je rovněž informace o verzi (autor a datum) a to zejména pro někoho, kdo již možná používá novější nebo starší verzi.

Poslední úsek souvisí s rysem systému Python, který každému modulu spuštěnému z příkazového řádku vnitřně říká "__main__" (čti mein; hlavní). Můžeme si otestovat obsah speciální zabudované proměnné __name__ (čti nejm; jméno). Pokud je v ní uveden zmíněný řetězec, pak víme, že modul nebyl importován, ale že byl spuštěn. Takže v takovém případě můžeme provést spouštěcí kód, který je uveden v těle příkazu if.

Náš spouštěcí kód obsahuje uživatelsky přívětivou nápovědu o způsobu spuštění programu pro případ, kdybychom jej spustili bez zadání jména souboru, nebo kdybychom naopak uvedli příliš mnoho jmen souborů.

Na závěr poznamenejme, že funkce Analyzuj() jednoduše zavolá ostatní funkce ve správném pořadí. K běžným praktikám patří opět to, že si uživatel může vybrat, zda bude chtít používat celkovou funkčnost přímočarým způsobem (voláním funkce Analyzuj()) nebo zda bude přímo volat nízkoúrovňové, *primitivní* funkce.

ziskejSkupinyZnaku()

Pseudokód pro tento úsek vypadal následovně:

```
pro každý řádek v souboru:
zvýšit počítadlo řádků
if je řádek prázdný:
zvýšit počítadlo odstavců
rozložit řádek na skupiny znaků
```

V jazyce Python to můžeme implementovat velmi snadno:

```
# Použijeme globální proměnné počítadel a globální seznam skupin znaků.
def ziskejSkupinyZnaku(vstSoubor):
global poc_odstavcu, poc_radku, skupiny
try:
for radek in vstSoubor.readlines():
poc_radku = poc_radku + 1
if len(radek) == 1: # prázdný řádek => oddělení odstavce
poc_odstavcu = poc_odstavcu + 1
else:
skupiny = skupiny + string.split(radek)
# Poznámka překladatele: Od verze Python 2 lze psát
# skupiny = skupiny + radek.split()
except:
print "Nepodarilo se cist ze souboru ", sys.argv[1]
sys.exit()
```

Poznámka 1: Abychom oznámili, že budeme používat proměnné, které byly vytvořeny vně těla funkce, musíme použít klíčové slovo global. Pokud bychom tak neučinili, pak by při přiřazování do nich Python vytvořil nové proměnné se stejnými jmény, které by ovšem byly *lokální* uvnitř těla funkce. Změny obsahu takových lokálních proměnných by se u *globálních* proměnných (na úrovni modulu) neprojevily.

Poznámka 2: K případnému oznámení chyb při čtení souboru a ukončení běhu programu jsme použili konstrukci try/except.

ziskejInerpunkci()

Zde budeme muset vyvinout o něco větší úsilí. Použijeme také pár nových rysů jazyka Python. Příslušný pseudokód vypadal následovně:

```
pro každou skupinu znaků:
zvýšit počítadlo skupin
odstranit interpunkční znaky a přidat do slovníku - {znak: počet}
if ve skupině nezbyl žádný znak:
zrušit skupinu
else: zvýšit počítadlo slov
```

Můj první pokus vypadal nějak takto:

```
def ziskejInerpunkci(seznamSkupin):
global poc_interpunkcnich_znaku
for skupina in seznamSkupin:
while skupina and (skupina[-1] not in alfanumericke_znaky):
p = skupina[-1]
skupina = skupina[:-1]
if p in poc_interpunkcnich_znaku.keys():
poc_interpunkcnich_znaku[p] = poc_interpunkcnich_znaku[p] + 1
else: poc_interpunkcnich_znaku[p] = 1
```

Povšimněte si, že tato verze nezahrnuje závěrečnou konstrukci if/else, kterou obsahoval pseudokód. Vynechal jsem ji kvůli zjednodušení. Měl jsem také pocit, že se v praktických případech objeví velmi málo skupin, které obsahují pouze interpunkční znaky. Ale do konečné verze kódu tuto konstrukci doplníme.

Poznámka 1: Seznam skupin jsme se rozhodli předávat parametrem, takže uživatelé tohoto modulu mohou předat svůj vlastní seznam místo toho, aby byli nuceni pracovat se souborem.

Poznámka 2: Za zmínku stojí obrat, kdy jsme proměnné skupina přiřadili hodnotu skupina[:-1]. V jazyce Python je tento obrat znám jako *slicing* (čti slajsing; odřezávání, odkrajování). Dvojtečka říká, aby byl index chápán jako rozsah. Pokud bychom například chtěli získat seznam položek skupina[3], skupina[4] a skupina[5], vyjádřili bychom jej

jako skupina[3:6]. (Poznámka překladatele: tento obrat lze používat pro posloupnosti různých druhů, konkrétně pro řetězce a pro seznamy.)
Pokud neuvedeme číslo, pak se to chápe jako začátek nebo konec seznamu podle toho, na které straně dvojtečky necháme prázdné místo. Takže zápis skupina[3:] by měl význam všech členů skupina od skupina[3:] až do konce. Jde o jeden z velmi užitečných rysů jazyka Python. V našem příkladu je originální posloupnost skupinaztracena (a tím pádem náležitě uklizena, uvolněna). Nově vytvořená posloupnost (v našem případě řetězec) je přiřazena do skupina.
Poznámka 3: Pro získání posledního znaku z proměnné skupina používáme záporný index. Jde opět o velmi užitečný rys jazyka Python. Pro případ, že by se na konci skupiny objevilo více interpunkčních znaků, provádíme zpracování v cyklu. Během testů se ukázalo, že totéž musíme provádět i pro začátek skupiny, protože, ačkoliv uzavírací závorky detekovány jsou, otvírací závorky nikoliv. Pro vyřešení tohoto problému vytvoříme novou funkci trim(), která odstraní interpunkční znaky ze začátku a z konce jedné skupiny znaků:

```
##############################################################
# Poznámka: trim používá rekurzi, kde podmínkou ukončení je buď hodnota
# 0 nebo -1. Pokud se objeví něco jiného, než hodnoty -1, 0 nebo 2,
# je vyvolána chyba "InvalidEnd".
# Poznámka překladatele: Od verze Python 2.0 se doporučuje pro výjimky
# používat instance tříd odvozených od třídy Exception. Používání
# řetězců pro výjimky se již nedoporučuje.
##############################################################
def trim(polozka, end = 2):
""" Odstraní nealfanumerické znaky zleva (end = 0), zprava (-1) nebo
z obou stran proměnné polozka."""

if end not in [0, -1, 2]:
raise "InvalidEnd"

if end == 2:
trim(polozka, 0)
trim(polozka, -1)
else:
while (len(polozka) > 0) and (polozka[end] not in alfanumericke_znaky):
ch = polozka[end]
if ch in poc_interpunkcnich_znaku.keys():
poc_interpunkcnich_znaku[ch] = poc_interpunkcnich_znaku[ch] + 1
if end == 0: polozka = polozka[1:]
if end == -1: polozka = polozka[:-1]
```

Povšimněte si, jak nám kombinace použití rekurze a implicitní (přednastavené) hodnoty parametru umožnila definovat jedinou funkci trim, která standardně ošetří oba konce. Pokud ale zadáme hodnotu parametru end, můžeme ji použít k ošetření pouze jednoho konce. Hodnoty parametru end jsou zvoleny tak, aby odpovídaly způsobu indexování v jazyce Python: nula pro levou stranu a -1 pro pravou. Původně jsem vytvořil dvě funkce trim, jednu pro každý konec. Ale díky množství pozorovaných podobností jsem zjistil, že je při použití parametru mohu zkombinovat dohromady.
Funkce ziskejInerpunkci se tím velmi zjednoduší:

```
def ziskejInerpunkci(seznamSkupin):
for skupina in seznamSkupin:
trim(skupina)
# Nyní vypustíme prázdná 'slova'.
for i in range(len(seznamSkupin)):
if len(seznamSkupin[i]) == 0:
del(seznamSkupin[i])
```

Poznámka 1: Nová implementace navíc provádí vypouštění prázdných slov.
Poznámka 2: V zájmu znovupoužitelnosti by možná bývalo bylo lepší, kdybychom funkci trim rozbili na ještě menší kousky. To by nám umožnilo vytvořit funkci pro odstranění jediného interpunkčního znaménka — buď ze začátku, nebo z konce slova — a odstraněný znak bychom mohli vracet jako výsledek. Takovou funkci by pak mohla opakovaně volat jiná funkce, čímž bychom získali požadovaný výsledek. Ale náš modul se má zabývat zjišťováním konkrétních statistických údajů ze zadaného textu a ne nějakým obecným zpracováním textu. Pro uvedené funkce bychom tedy správně měli vytvořit samostatný modul, který bychom pak importovali. Jenže ten by obsahoval jen jednu funkci, která se navíc nezdá být příliš užitečnou. Takže to raději nechejme, jak to je.
Konečná podoba modulu gramatika
Teď už nám zbývá jenom vylepšit výpis výsledků tím, že do něj zahrneme další počítadla a vliv interpunkčních znaků. Existující funkci vypsatStatistiky() nahraďte následujícím kódem:

```
def vypsatStatistiky():
global poc_vet, poc_klauzuli
for c in koncove_znaky:
poc_vet = poc_vet + poc_interpunkcnich_znaku[c]
for c in poc_interpunkcnich_znaku.keys():
poc_klauzuli = poc_klauzuli + poc_interpunkcnich_znaku[c]
print format % (sys.argv[1], poc_odstavcu,
poc_radku, poc_vet,
poc_klauzuli, poc_slov)
print "Byly pouzity nasledujici interpunkcni znaky:"
for c in poc_interpunkcnich_znaku.keys():
print "\t%s\t:\t%3d" % (c, poc_interpunkcnich_znaku[c])
```

Pokud jste pečlivě vložili všechny výše uvedené funkce na správná místa, měli byste nyní po napsání

```
C:> python gramatika.py mujsoubor.txt
```

obdržet výpis statistických údajů pro váš soubor mujsoubor.txt (ať už jej nazvete jak chcete). Užitečnost tohoto programu je diskutabilní, ale snad vám sledování vývoje jeho kódu pomohlo získat představu o tom, jak můžete tvořit své vlastní programy. Za hlavní považuji to, abyste si vše zkoušeli. A ovšem, měli byste si je také pečlivě otestovat. V případě tohoto programu například rychle zjistíte způsoby, jak z něj vylákat falešné odpovědi. Pokud například větu zakončíte třemi tečkami, bude počítadlo vět nabývat příliš vysoké hodnoty. Pro rozpoznání takových situací můžete doplnit k tomu určený kód. Můžete se také rozhodnout, že s ohledem na občasné použití programu vám podobné věci nevadí. Je to na vás. Nepovažuji za žádnou ostudu, když si vyzkoušíte několik různých přístupů. Často během toho získáte cenné zkušenosti.
Náš kurs ukončíme přepracováním modulu gramatika na použití technik objektově orientovaného programování. Během tohoto procesu uvidíte, že objektově orienovaný přístup vede k modulům, které jsou pro koncového uživatele ještě pružnější a také rozšiřitelnější.

Třídy a objekty

Jeden z největších problémů, se kterým se uživatel našeho modulu setká, spočívá ve spoléhání se na globální proměnné. Vede to k tomu, že můžeme analyzovat vždy jen jeden dokument najednou. Jakýkoliv pokus o zpracování více dokumentů by vedl k přepisování hodnot globálních proměnných.
Pokud přeneseme tyto globální údaje dovnitř třídy, můžeme vytvořit několik instancí dané třídy (pro každý soubor jednu). Každá instance tím získá svou vlastní sadu proměnných. Když navíc metody třídy dostatečně rozčleníme, můžeme vytvořit architekturu, u které bude moci tvůrce objektu pro nový typ dokumentu snadno upravit kritéria tak, aby vyhovovala novým pravidlům (ze seznamu slov můžeme například vyloučit všechny HTML značky).
Náš první pokus vypadá takto:

```
#! /usr/local/bin/python
################################
# Modul: dokument.py
# Autor: A.J. Gauld
# Datum: 2000/08/12
# Verze: 2.0
################################
# Tento modul poskytuje třídu Dokument, ze které
# lze odvozovat další třídy pro různé kategorie
# dokumentů (text, HTML, LaTeX, atd.). Jako vzor
# jsou uvedeny třídy pro text a HTML.
#
# K nejdůležitějším službám patří
# - ziskejSkupinyZnaku(),
# - ziskejSlova(),
# - vypsatStatistiky().
################################
import sys, string

class Dokument:
def __init__(self, jmenoSouboru):
self.jmenoSouboru = jmenoSouboru
self.poc_odstavcu = 1
self.poc_radku, self.poc_vet = 0, 0
self.poc_klauzuli, self.poc_slov = 0, 0
self.alfanum = string.letters + string.digits
self.koncove_znaky = ['.','?','!']
self.interpunkcni_znaky = ['&', '(', ')', '-', ';',
':', ','] + self.koncove_znaky
self.radky = []
self.skupiny = []
self.poc_interp_znaku = {}
for c in self.interpunkcni_znaky + self.koncove_znaky:
self.poc_interp_znaku[c] = 0
self.format = """%s obsahuje:
%d odstavcu, %d radku a %d vet.
Ty zase obsahuji %d klauzuli a celkem %d slov."""

def nactiRadky(self):
try:
self.vstSoubor = open(self.jmenoSouboru, "r")
# Poznámka překladatele: Od verze Python 2 by se
# měla dávat přednost zápisu
# self.vstSoubor = file(self.jmenoSouboru, "r")
self.radky = self.vstSoubor.readlines()
self.vstSoubor.close()
except:
print "Chyba pri cteni ze souboru ", self.jmenoSouboru
sys.exit()

def ziskejSkupinyZnaku(self, radky):
for radek in radky:
radek = radek[:-1]  # odstraníme koncový '\n'
self.poc_radku = self.poc_radku + 1
if len(radek) == 0: # prázdný => další odstavec
self.poc_odstavcu = self.poc_odstavcu + 1
else:
```

```
            self.skupiny = self.skupiny + string.split(radek)
            # Poznámka překladatele: Od verze Python 2 lze psát
            # self.skupiny = self.skupiny + radek.split()


        def ziskejSlova(self):
            pass

    def vypsatStatistiky(self, odstavcu=1, radku=1, vet=1, slov=1, interpun=1):
        pass

    def Analyzuj(self):
        self.nactiRadky()
        self.ziskejSkupinyZnaku(self.radky)
        self.ziskejSlova()
        self.vypsatStatistiky()

class TextovyDokument(Dokument):
    pass

class HTMLDokument(Dokument):
    pass

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print "Pouziti: python dokument.py <jmeno souboru>"
        sys.exit()
    else:
        D = Dokument(sys.argv[1])
        D.Analyzuj()
```

Následujícím krokem při implementaci této třídy je definice metody ziskejSlova. Mohli bychom jednoduše okopírovat to, co jsme vytvořili v předchozí verzi a vytvořit nějakou metodu trim. Jenže my chceme, aby byla objektově orientovaná verze snadno rozšiřitelná. Takže místo toho rozdělíme metodu ziskejSlova na posloupnost několika kroků. V odvozených třídách pak stačí přepsat jen nové verze těchto podkroků a nikoliv celou metodu ziskejSlova. To by mělo usnadnit zpracování mnohem širšího rozsahu typů dokumentů.

Konkrétně přidáme metodu pro odmítnutí skupin, které budou rozpoznány jako chybné, metodu pro odstranění nechtěných znaků ze začátku a z konce. To znamená, že do třídy Dokument přidáme tři metody a metoduziskejSlova implementujeme pomocí nich.

```
class Dokument:
    # ... viz výše
    def ziskejSlova(self):
        for w in self.skupiny:
            self.orezatLeve(w)
            self.orezatPrave(w)
        self.odstranitVyjimky()

    def odstranitVyjimky(self):
        pass

    def orezatLeve(self, slovo):
        pass

    def orezatPrave(self, slovo):
        pass
```

Povšimněte si, že těla uvedených funkcí definují použití jediného příkazu pass (čti pás; projít), který nedělá vůbec nic. Místo něj budeme později definovat způsob zpracování pro každý *konkrétní* typ dokumentu.

Textový dokument

Třída pro textový dokument vypadá takto:

```
class TextovyDokument(Dokument):
    def orezatLeve(self, slovo):
        while (len(slovo) > 0) and (slovo[0] not in self.alfanum):
            ch = slovo[0]
            if ch in self.poc_interp_znaku.keys():
                self.poc_interp_znaku[ch] = self.poc_interp_znaku[ch] + 1
            slovo = slovo[1:]
        return slovo

    def orezatPrave(self, slovo):
        while (len(slovo) > 0) and (slovo[-1] not in self.alfanum):
            ch = slovo[-1]
            if ch in self.poc_interp_znaku.keys():
                self.poc_interp_znaku[ch] = self.poc_interp_znaku[ch] + 1
            slovo = slovo[:-1]
        return slovo

    def odstranitVyjimky(self):
```

```
self.skupiny = filter(lambda g: len(g) > 0, self.skupiny)
```

Ořezávací funkce (anglicky se tato funkčnost označuje slovem *trim*) jsou v podstatě shodné s funkcí trim v našem modulu gramatika.py, která byla ovšem rozdělena na dvě. Funkce odstranitVyjimky byla definována tak, aby odstraňovala prázdná slova. Povšimněte si použití funkce filter(), o které jsme se zmínili v části věnované funkcionálnímu programování.

HTML dokument

Při zpracování HTML dokumentů použijeme funkčnost jazyka Python, se kterou jsme se ještě nesetkali — *regulární výrazy*. Jde o speciální řetězcové vzorky, které se dají použít pro nalezení složitějších řetězců. Použijeme je zde k odstranění čehokoliv mezi znaky < a >. To znamená, že budeme muset předefinovat metodu ziskejSlova. Odstraňování interpunkčních znamének by mělo být shodné jako v případě zpracování holého textu. Takže místo abychom dědili přímo z třídy Dokument, odvodíme novou třídu z TextovyDokument a použijeme jí definované metody pro ořezávání.

Takže třída HTMLDokument bude vypadat takto:

```
class HTMLDokument(TextovyDokument):
def odstranitVyjimky(self):
""" Použijeme regulární výrazy pro odstranění všech <.+?>"""
import re
tag = re.compile("<.+?>")# použijeme non greedy re
L = 0
while L < len(self.radky):
if len(self.radky[L]) > 1: # pokud řádek není prázdný
self.radky[L] = tag.sub('', self.radky[L])
if len(self.radky[L]) == 1:
del(self.radky[L])
else: L = L + 1
else: L = L + 1


def ziskejSlova(self):
self.odstranitVyjimky()
for i in range(len(self.skupiny)):
slovo = self.skupiny[i]
slovo = self.orezatLeve(slovo)
self.skupiny[i] = self.orezatPrave(slovo)
TextovyDokument.odstranitVyjimky(self)# odstraní prázdná slova
```

Poznámka: V tomto místě stojí za zmínku pouze volání self.odstranitVyjimky před ořezáváním a poté voláníTextovyDokument.odstranitVyjimky. Pokud bychom se spoléhali na zděděnou metodu ziskejSlova, byla by po provedení ořezání volána naše metoda odstranitVyjimky, což nechceme.

Poznámka překladatele: Pojem *greedy* (čti grídy), který se objevil v poznámce u regulárního výrazu, odpovídá významu anglického slovíčka — chamtivý, lakomý, žravý. Vzorek <.+>, který bychom použili pro regulární výraz lze číst jako *řetězec, který začíná levou úhlovou závorkou (znak 'menší'), pokračuje jedním a více (znak plus) libovolných znaků (znak tečka) a končí pravou úhlovou závorkou (znak 'větší')*. Pokud neurčíme jinak, snaží se Python najít odpovídající řetězec, který je co největší (tj. *greedy*, neboli žravé chování). To ovšem znamená, že by se k prvnímu znaku 'menší' na daném řádku nalezl až poslední znak 'větší'. Při přeskakování libovolných znaků by mohly být přeskočeny i znaky menší/větší, které se nacházejí mezi nimi. My ovšem potřebujeme, aby se přeskakování zastavilo na nejbližším znaku 'větší', který ukončuje HTML značku. To znamená, že chceme předepsat opačné chování (tj. *non greedy*), kdy se nalezne co nejkratší vyhovující řetězec. V jazyce Python tento požadavek vyjádříme tím, že za popis skupiny doplníme otazník. Použijeme tedy vzorek <.+?>.

Přidáme grafické uživatelské rozhraní

Pro vytvoření grafického uživatelského rozhraní použijeme Tkinter, který jsme stručně představili v kapitole Událostmi řízené programování a kterému jsme se věnovali podrobněji v rámci tématu věnovanému grafickému uživatelskému rozhraní. Tentokrát vytvoříme o něco propracovanější grafické uživatelské rozhraní a použijeme více ovládacích prvků, zvaných také *widget*, které nám Tkinter poskytuje.

Refaktorizace[1] třídy Dokument

Dříve než se do tohoto stadia dostaneme, musíme upravit naši třídu Dokument. Dosavadní verze provádí zobrazení výsledků tiskem na standardní výstup (stdout) v rámci metody Analyzuj. Při vytváření grafického uživatelského rozhraní to ale není to pravé. Místo toho bychom raději přivítali, kdyby metoda Analyzuj jednoduše uložila výsledky v atributech s charakterem počítadel, ke kterým bychom pak přistupovali podle potřeby. Dosáhneme toho jednoduše rozdělením nebo *refaktorizací* metody vypsatStatistiky() na dvě části: metodu vypocetStatistik(), která vyhodnotí výsledky a uloží je v počítadlech, a na metodu tiskStatistik(), která vytiskne výsledky na standardní výstup.

Nakonec musíme upravit metodu Analyzuj() tak, aby volala metodu vypocetStatistik(), a hlavní sekvenci příkazů tak, aby po volání metody Analyzuj() volala tiskStatistik(). Po provedení těchto úprav bude stávající kód pracovat stejným způsobem, jako předtím — přinejmenším z pohledu uživatele, který takový program spouští z příkazového řádku. Ostatní uživatelé (tj. ti, kteří zdrojový soubor používají jako modul, vytvářejí si sami instance třídy Dokument a volají sami jeho metody) budou muset provést ve svém kódu drobné změny, kdy po použití metody Analyzuj() zavolají metodu tiskStatistik() — a to není příliš obtížné.

Revidované úseky kódu vypadají takto:

```
def vypocetStatistik(self):
self.poc_slov = len(self.skupiny)
for c in self.koncove_znaky:
self.poc_vet = self.poc_vet + self.poc_interp_znaku[c]
for c in self.poc_interp_znaku.keys():
self.poc_klauzuli = self.poc_klauzuli + self.poc_interp_znaku[c]

def tiskStatistik(self):
print self.format % (self.jmenoSouboru, self.poc_odstavcu,
self.poc_radku, self.poc_vet,
self.poc_klauzuli, self.poc_slov)
```

```
print "Byly pouzity nasledujici interpunkcni znaky:"
for c in self.poc_interp_znaku.keys():
print "\t%s\t:\t%4d" % (c, self.poc_interp_znaku[c])
```

a tělo prováděné při spuštění z příkazového řádku:

```
if __name__ == "__main__":
if len(sys.argv) != 2:
print "Pouziti: python dokument.py <jmeno souboru>"
sys.exit()
else:
try:
D = HTMLDokument(sys.argv[1])
D.Analyzuj()
D.tiskStatistik()
except:
print "Chyba pri analyze souboru: %s" % sys.argv[1]
```

Nyní jsme připraveni k tomu, abychom naše třídy dokumentů obalili grafickým uživatelským rozhraním.

Návrh grafického uživatelského rozhraní

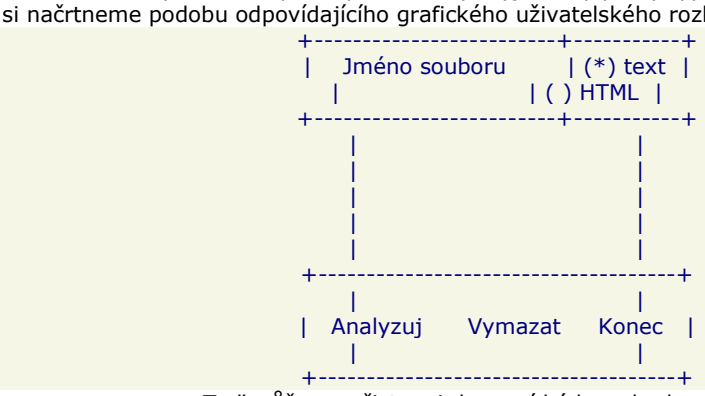
V prvním kroku si pokusíme představit, jak to vše bude vypadat. Musíme zadat jméno souboru, takže budeme potřebovat ovládací prvky *Edit* nebo *Entry*. Musíme určit, zda chceme provádět analýzu holého textu nebo obsahu v podobě HTML. Takový způsob výběru *jedné z několika možností* je obvykle reprezentován sadou prvků typu *Radiobutton*. Tyto ovládací prvky by měly být sdruženy dohromady, aby bylo vidět, že spolu souvisejí.

Dále požadujeme nějaký způsob zobrazení výsledků. Mohli bychom použít několik prvků typu *Label* — jeden pro každé počítadlo. Místo toho použiji jednoduchý prvek typu *Text*, do kterého můžeme vkládat řetězce. Tento přístup se přibližuje duchu dřívějšího řádkového výstupu, ale konkrétní způsob výstupu je věcí volby návrháře.

Na závěr, potřebujeme nějaké prostředky pro zahájení analýzy a pro ukončení aplikace. Protože pro zobrazení výsledků použijeme ovládací prvek typu text, mohlo by být užitečné, kdybychom mohli do počátečního stavu uvést i zobrazování.

Všechny uvedené příkazy mohou být vyjádřeny prvky typu *Button* (tlačítko).

Pokud si načrtneme podobu odpovídajícího grafického uživatelského rozhraní, dostaneme něco takového:



Teď můžeme přistoupit k psaní kódu — krok po kroku:

```
from Tkinter import *
import dokument

################### Definice tříd ######################
class AplikaceGramatika(Frame):
def __init__(self, rodic=0):
Frame.__init__(self, rodic)
self.typ = 2 # vytvoř proměnnou s počáteční hodnotou
self.master.title('Pocitadlo gramatickych prvku')
self.vybudovatUI()
```

Nejdříve jsme provedli import modulů Tkinter a dokument. V prvním případě jsme si v rámci vytvářeného modulu zajistili viditelnost všech jmen z Tkinter, zatímco v druhém případě budeme muset před jména přidávat předponu 'dokument'. Definovali jsme i metodu __init__, která volá metodu Frame.__init__ své bázové třídy. Tím se zajistí správná vnitřní nastavení v rámci Tkinter. Poté vytváříme atribut, ve kterém bude uložena hodnota typu dokumentu. A nakonec voláme metodu vybudovatUI, která nám vytvoří všechny potřebné ovládací prvky.

```
def vybudovatUI(self):
# Informace o souboru: Jméno a typ
fSoubor = Frame(self)
Label(fSoubor, text="Jmeno souboru: ").pack(side=LEFT)
self.eJmeno = Entry(fSoubor)
self.eJmeno.insert(INSERT, "test.htm")
self.eJmeno.pack(side=LEFT, padx=5)


# Pro zajištění zarovnání přepínacích tlačítek (radio buttons)
# se jménem potřebujeme další rámec.
fTyp = Frame(fSoubor, borderwidth=1, relief=SUNKEN)
self.rText = Radiobutton(fTyp, text="text",
variable = self.typ, value=2,
command=self.udalostText)
self.rText.pack(side=TOP, anchor=W)
self.rHTML = Radiobutton(fTyp, text="HTML",
variable=self.typ, value=1,
```

```
                command=self.udalostHTML)
        self.rHTML.pack(side=TOP, anchor=W)
        # Na počátku vybereme 'text'
        self.rText.select()
        fTyp.pack(side=RIGHT, padx=3)
        fSoubor.pack(side=TOP, fill=X)


        # V textovém okně se zobrazuje výstup. Použijeme vycpávku, abychom
        # získali rámeček. Rodičovským rámcem bude rámec celé aplikace
        # (tj. self)
        self.txtBox = Text(self, width=60, height=10)
        self.txtBox.pack(side=TOP, padx=3, pady=3)


        # Nakonec umístíme příkazová tlačítka, která budou spouštět činnosti.
        fTlacitka = Frame(self)
        self.bAnalyzuj = Button(fTlacitka, text="Analyzuj",
                command=self.udalostAnalyzuj)
        self.bAnalyzuj.pack(side=LEFT, anchor=W, padx=50, pady=2)
        self.bReset = Button(fTlacitka, text="Reset",
                command=self.udalostReset)
        self.bReset.pack(side=LEFT, padx=10)
        self.bKonec = Button(fTlacitka, text="Konec",
                command=self.udalostUkonceni)
        self.bKonec.pack(side=RIGHT, anchor=E, padx=50, pady=2)

        fTlacitka.pack(side=BOTTOM, fill=X)
        self.pack()
```

Nebudu zde vysvětlovat všechny detaily. Místo toho vám doporučuji k nahlédnutí učebnici Tkinter, kterou naleznete na webových stránkách jazyka Python. Jde o vynikající úvod i referenční příručku k Tkinter. Obecný princip uvedeného kódu spočívá ve vytváření ovládacích prvků odpovídajících tříd, při kterém zadáváme nastavení formou *pojmenovaných parametrů*. Poté je příslušný prvek umístěn do svého obklopujícího rámce voláním metody pack.

Poznamenejme, že k dalším klíčovým bodům patří použití pomocných prvků typu Frame, které sdružují přepínací a příkazová tlačítka. U přepínacích tlačítek (radio buttons) se mimo jiné uvádí dvojice parametrů s názvy variable (proměnná) a value (hodnota). První z uvedených svazuje přepínací tlačítka dohromady tím, že udává stejnou vnější proměnnou (self.typ). Druhý parametr přiděluje každému přepínacímu tlačítku jednoznačnou hodnotu. Všimněte si také parametru command=xxx, který se předává prvkům tlačítek. Jde o metody, které bude Tkinter volat v okamžiku stisku tlačítka. Jejich kód je uveden níže:

```
    ################## Metody pro ošetření událostí ####################
    # je načase vše skoncovat...
    def udalostUkonceni(self):
        import sys
        sys.exit()


    # nastavíme vše do počátečního stavu
    def udalostReset(self):
        self.txtBox.delete(1.0, END)
        self.rText.select()


    # nastavíme hodnotu přepínacího tlačítka
    def udalostText(self):
        self.typ = 2

    def udalostHTML(self):
        self.typ = 1
```

Uvedené metody jsou velmi jednoduché a doufám, že není nutné je vysvětlovat. Poslední metoda pro ošetření události zajišťuje provedení analýzy:

```
    # Vytvoříme odpovídající typ dokumentu a provedeme analýzu.
    # Poté zobrazíme výsledky v podobě řetězců.
    def udalostAnalyzuj(self):
        jmenoSouboru = self.eJmeno.get()
        if jmenoSouboru == "":
            self.txtBox.insert(END,"\nNo filename provided!\n")
            return
        if self.typ == 2:
            doc = dokument.TextovyDokument(jmenoSouboru)
        else:
            doc = dokument.HTMLDokument(jmenoSouboru)
        self.txtBox.insert(END, "\nAnalyzuji...\n")
        doc.Analyzuj()
        str = doc.format % (doc.jmenoSouboru,
```

```
doc.poc_odstavcu, doc.poc_radku,
doc.poc_vet, doc.poc_klauzuli, doc.poc_slov)
self.txtBox.insert(END, str)
```

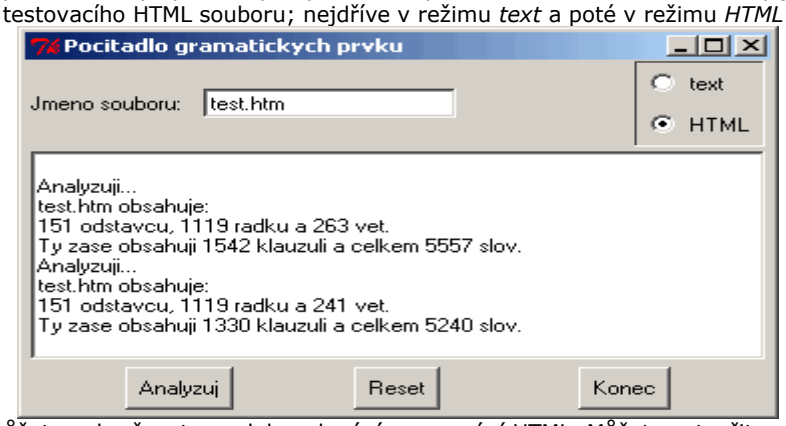
I výše uvedený text byste již měli být schopni přečíst a pochopit, co dělá. Jeho klíčové body jsou následující:

- Před vytvořením objektu třídy Dokument zkontroluje platnost zadaného jména souboru.
- Konkrétní typ dokumentu je vytvořen podle hodnoty proměnné self.typ, která je nastavena podle přepínacích tlačítek.
- Výsledek je připojen na konec obsahu okna s textem (viz argument END metody insert. To znamená, že můžeme provést několik analýz a porovnávat výsledky. Ve srovnání s dříve zmíněným přístupem, kdy by byly výsledky zobrazovány formou ovládacích prvků s popisným textem (label), jde o jednu z výhod použití výstupu do okna s textem.

Teď už nám zbývá jen vytvořit hlavní objekt aplikace a spustit smyčku pro zpracování událostí:

```
mojeAplikace = AplikaceGramatika()
mojeAplikace.mainloop()
```

Podívejme se, jak vypadá konečný výsledek při spuštění v systému MS Windows. Zobrazeny jsou výsledky analýzy testovacího HTML souboru; nejdříve v režimu *text* a poté v režimu *HTML*:



A je to. Pokud chcete, můžete pokračovat ve zdokonalování zpracování HTML. Můžete vytvořit nové moduly pro nové typy dokumentů. Můžete zkusit zaměnit okno s textem za několik prvků s popisným textem vložených do rámce. Ale z pohledu našeho původního záměru jsme hotovi. Následující kapitola nabízí náměty k dalšímu studiu v závislosti na vašich programátorských tužbách. Hlavní je, aby vás to bavilo. A vždycky si pamatujte: *Počítač je hloupý!*