Mark Pilgrim

Ponořme se do Python(u) 3

Python 3

Dive Into Python 3

Edice CZ.NIC

Mark Pilgrim
PONOŘME SE DO PYTHON(U) 3

Vydavatel:
CZ.NIC, z. s. p. o.
Americká 23, 120 00 Praha 2
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2011
Kniha vyšla jako 3. publikace v Edici CZ.NIC.
ISBN 978-80-904248-2-1

CZ.NIC | SPRÁVCE DOMÉNY CZ

ISBN 978-80-904248-2-1

— Mark Pilgrim

Ponořme se do Python(u) 3
Dive Into Python 3

— Edice CZ.NIC

Předmluva a ediční poznámka

Vážení čtenáři,

po úspěchu naší předchozí publikace ProGit jsme se rozhodli, že třetí kniha v Edici CZ.NIC bude tak trochu na podobné téma a v podobném duchu. Opět jde o překlad velice kvalitní zahraniční publikace a také v tomto případě se dá očekávat, že jej ocení hlavně programátoři. Samozřejmě jsme i tentokrát sáhli po knize, která je pod volnou licencí a tedy filozofie její distribuce je blízká naší edici.

Podobně jako v případě nástroje Git je i Python technologie, která je mým kolegům velice dobře známa. Právě v programovacím jazyce Python je napsána podstatná část našeho centrálního registru pro správu domén, který se jmenuje FRED. Toto je jen jeden z mnoha důkazů, proč je nutné se tímto programovacím jazykem vážně zabývat.

Autor knihy **Mark Pilgrim** není ve světě Pythonu rozhodně žádným neznámým jménem. Své renomé si vybudoval již napsáním předchůdce této knihy s téměř stejným jménem. Právě úspěch dřívějšího díla je pro nás zárukou, že i tato verze si najde své čtenáře.

Ať už jste tedy v Pythonu nováčky nebo si jen chcete rozšířit své dosavadní znalosti, přeji Vám příjemnou četbu.

Ondřej Filip
Praha 17. listopadu 2010

Ediční poznámka autora

Ponořme se do Pythonu 3 pokrývá vlastnosti jazyka Python 3 a popisuje rozdíly proti jazyku Python 2. Ve srovnání s **Dive Into Python** zde naleznete asi 20 % revidovaného textu a asi 80 % nového materiálu. Knihu považuji za dokončenou, ale zpětná vazba je vždy vítána.

Obsah

— Přehled kapitol

“Isn’t this where we came in?”

— Pink Floyd, *The Wall*

-1. Co najdete v „Ponořme se do Pythonu 3“ nového

-1.1. aneb „záporná úroveň”

Už jste v jazyce Python programovali? Četli jste originální publikaci „Dive Into Python“? Koupili jste si ji v knižní podobě? (Pokud ano, díky!) Jste připraveni ponořit se do jazyka Python 3?... Pokud tomu tak je, čtěte dál. (Pokud nic z toho neplatí, měli byste raději začít od začátku.)

Kap. Python 3 se dodává se skriptem nazvaným `2to3`. Naučte se jej. Milujte jej. Používejte jej. Přepis kódu do Pythonu 3 s využitím `2to3` je referenční příručkou ke všem věcem, které skript `2to3` umí opravit automaticky. A protože řada těchto věcí souvisí se změnami syntaxe, je tato příručka dobrým výchozím bodem ke studiu syntaktických změn, které Python 3 přináší. (Z příkazu `print` se stala funkce, obrat `` `x` `` přestal fungovat atd.)

Kap. Případová studie: Přepis `chardet` pro Python 3 popisuje mé (nakonec úspěšné) úsilí o přepis netriviální knihovny z Pythonu 2 do Pythonu 3. Možná vám tato studie pomůže, možná ne. Učící křivka je zde poměrně strmá, protože nejdříve musíte porozumět knihovně samotné. Teprve potom můžete rozumět tomu, proč přestala fungovat a jakým způsobem jsem ji opravil. Řada problémů se váže na řetězce. Když už o nich mluvíme...

Řetězce. Uffff. Kde mám začít? Python 2 používal „řetězce“ a „řetězce v Unicode“. Python 3 rozlišuje „bajty“ a „řetězce“. Všechny řetězce se nyní stávají řetězci v Unicode. Pokud s obsahem chceme zacházet jako s bajty, musíme použít nový datový typ nazvaný `bytes`. Python 3 *nikdy* skrytě nepřevádí řetězce na bajty a naopak. Takže pokud si v každém momentě nejste jistí, zda používáte ten či onen typ, kód
Kap. vašeho programu téměř jistě přestane fungovat. Další podrobnosti naleznete v kapitole Řetězce.

Problém bajty versus řetězce se v textu této knihy vynořuje znovu a znovu.

Kap. • V kapitole Soubory se seznámíte s rozdílem mezi čtením souborů v „binárním“ a „textovém“ režimu. Při čtení (ale také při zápisu) souborů v textovém režimu se vyžaduje zadání parametru určujícího kódování (`encoding`). Některé metody textových souborů počítají znaky, ale jiné metody zase počítají bajty. Pokud ve svém zdrojovém kódu předpokládáte, že se jeden znak rovná jednomu bajtu, pak to při přechodu na vícebajtové znaky *přestane fungovat*.

Kap. • V kapitole Webové služby nad `HTTP` čte modul `httplib2` hlavičky a data prostřednictvím protokolu `HTTP`. Hlavičky se vracejí v podobě řetězců, ale těla se vracejí jako bajty.

Kap. • V kapitole Serializace pythonovských objektů se naučíte, proč modul `pickle` pro Python 3 definuje nový datový formát, který je zpětně nekompatibilní s verzí pro Python 2. (Nápověda: Důvodem jsou bajty a řetězce.) Python 3 podporuje také serializaci objektů do a z `json`, který dokonce nepracuje s typem `bytes`. Ukážeme si, jak se to dá obejít.

Kap. • V části Případová studie: Přepis `chardet` pro Python 3 se setkáte se zatraceným zmatkem mezi bajty a řetězci úplně všude.

Dokonce i kdyby vás Unicode nechával úplně chladné (ale ne, nenechá), budete si určitě chtít něco přečíst o formátování řetězců v jazyce Python 3. Zcela se liší od předpisu formátování řetězců v jazyce Python 2.

S iterátory se v Pythonu 3 setkáte všude. A teď už jim rozumím mnohem víc, než tomu bylo před pěti lety, kdy jsem napsal „Dive Into Python“. Snažte se jim porozumět také, protože mnoho funkcí, které v jazyce Python 2 vracely seznamy, vrací v Pythonu 3 právě iterátory. Přinejmenším byste si měli přečíst druhou polovinu kapitoly Iterátory a druhou polovinu kapitoly Iterátory pro pokročilé.

Kap. Na přání čtenářů jsem přidal přílohu Jména speciálních metod, která se podobá kapitole Data Model (Datový model) uvedené v dokumentaci jazyka Python.

V době, kdy jsem psal „Dive Into Python“, měly všechny dostupné knihovny pro práci s XML mizernou kvalitu. Pak ale Fredrik Lundh napsal modul **ElementTree**, který není *ale vůbec mizerný*. Pythonovští bohové moudře začlenili ElementTree do standardní knihovny, a tak se tento modul stal základem mé nové kapitoly o XML. Starší způsoby zpracování XML jsou stále podporované, ale měli byste se jim vyhnout, protože jsou zkrátka mizerné!

V Pythonu je nové také to — ne v jazyce, ale v komunitě uživatelů —, že se objevila úložiště kódu, jako je **Python Package Index (PyPI)**. Python se dodává s utilitami k zabalení vašeho kódu do standardního formátu a tyto balíčky pak mohou být zveřejněny na PyPI. O podrobnostech se dočtete v kapitole Kap. Balení pythonovských knihoven.

“Tempora mutantur nos et mutamur in illis. ”
(Časy se mění a my se měníme s nimi.)

— přísloví ze starého Říma

0. Instalujeme Python

0.1. Ponořme se

Než začneme programovat v jazyce Python 3, musíme si jej nainstalovat. Nebo ne?

0.2. Který Python je pro vás ten správný?

Pokud používáte účet na hostovaném serveru, mohl být Python 3 již nainstalován jeho správcem. Pokud provozujete Linux doma, můžete mít Python 3 již také k dispozici. Nejpopulárnější distribuce systému GNU/Linux obsahují v základní instalaci Python 2. Malá, ale zvětšující se skupina distribucí obsahuje také Python 3. Mac OS X se dodává s Pythonem 2 (verze spouštěná přes příkazový řádek), ale v době psaní této knihy neobsahoval Python 3. Microsoft Windows se nedodává s žádnou verzí Pythonu. Ale nepropadejte zoufalství! Nezávisle na tom, jaký operační systém používáte, můžete Python nainstalovat na několik kliknutí.

Nejjednodušší způsob ověření si, zda máte k dispozici Python 3 na svém systému Linux nebo Mac OS X, začíná tím, že se dostanete na příkazový řádek. Jakmile se nacházíte za vyzývacím řetězcem příkazového řádku, napište jednoduše `python3` (vše malými písmeny, bez mezer), stiskněte `ENTER` a uvidíte, co se stane. Na svém domácím systému Linux už mám Python 3.1 nainstalovaný. Uvedeným příkazem vstoupím do *pythonovského interaktivního shellu*.

```
mark@atlantis:~$ python3
Python 3.1 (r31:73572, Jul 28 2009, 06:52:23)
[GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(Až budete chtít pythonovský interaktivní shell opustit, napište `exit()` a stiskněte `ENTER`.)

Můj poskytovatel webového prostoru používá také Linux a umožňuje přístup přes příkazový řádek, ale Python 3 není na serveru nainstalován. (Bééé!)

```
mark@manganese:~$ python3
bash: python3: command not found
```

Takže zpět k otázce, kterou jsme tuto podkapitolu zahájili: „Který Python je pro vás ten správný?“ Ten, který poběží na počítači, který máte k dispozici.

> Následuje návod pro instalaci pod Windows, nebo přeskočte na Instalace pod Mac OS X, Instalace pod Ubuntu Linux nebo Instalace na jiných platformách.

0.3. Instalace pod Microsoft Windows

V dnešní době se Windows dodávají ve dvou architekturách: 32bitové a 64bitové. Máme tu samozřejmě řadu různých verzí Windows — XP, Vista, Windows 7 —, ale Python běží na všech. Rozlišení mezi 32bitovou a 64bitovou architekturou je důležitější. Pokud nemáte vůbec tušení, jakou architekturu používáte, pak je to pravděpodobně 32bitová.

Přejděte na stránku `python.org/download/` a stáhněte si windowsovský instalátor Python 3, který se hodí pro vaši architekturu. Možnosti vaší volby budou vypadat nějak takto:

- **Python 3.1 Windows installer** (Windows binary — does not include source)
- **Python 3.1 Windows AMD64 installer** (Windows AMD64 binary — does not include source)

Nechci zde uvádět konkrétní odkazy, protože Python neustále prochází drobnými úpravami a nechci být zodpovědný za to, že jste nějakou důležitou úpravu prošvihli. Vždy byste měli nainstalovat co nejnovější verzi Pythonu 3.x, tedy pokud nemáte nějaké esoterické důvody k tomu, abyste tak neučinili.

Jakmile se stahování dokončí, poklepejte na soubor s příponou `.msi`. Protože se snažíte o spuštění programu, zobrazí Windows bezpečnostní varování. Oficiální instalátor Pythonu je digitálně podepsán jménem organizace *Python Software Foundation*, která dohlíží na vývoj jazyka Python. Nepřijímejte imitace!

Instalaci Pythonu 3 zahájíme stisknutím tlačítka `Run`.

Nejdříve se vás instalátor zeptá, zda chcete Python 3 nainstalovat pro všechny uživatele, nebo jen pro sebe. Volba „instalovat pro všechny uživatele“ je přednastavena. Pokud nemáte nějaký dobrý důvod pro jinou volbu, pak toto je ta nejlepší. (Jeden možný důvod, proč byste mohli chtít „instalovat jen pro mne“, je ten, že si chcete nainstalovat Python na počítači v práci a váš účet ve Windows nemá oprávnění administrátora. Ale proč byste v takovém případě chtěli instalovat Python bez svolení svého správce Windows? Ne abyste mě dostali do potíží!)

Svoji volbu způsobu instalace potvrdíte stiskem tlačítka `Next`.

Instalátor vás poté vyzve k výběru instalačního adresáře. Pro všechny verze Python 3.1.x je přednastavena hodnota `C:\Python31\`, která by měla vyhovovat většině uživatelů. Pokud ovšem nemáte zvláštní důvod cestu změnit. Pokud instalujete všechny aplikace na disk označený jiným písmenem, můžete příslušnou cestu vybrat příslušnými ovládacími prvky. Nebo prostě cestu k adresáři napíšete do spodního pole. Python nemusíte instalovat jen na disk `C:`. Můžete si jej nainstalovat na libovolný disk a do libovolného adresáře.

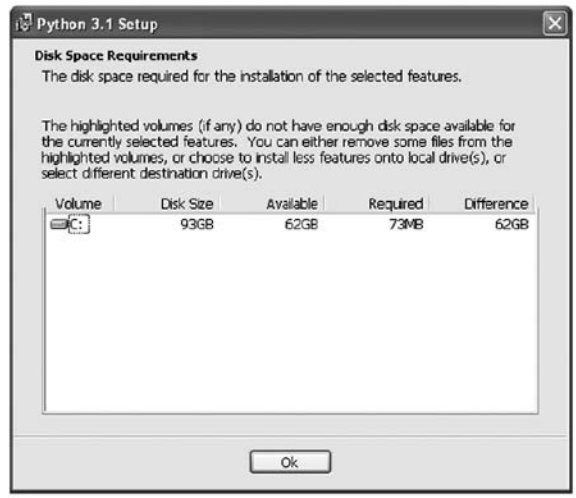
Volbu cílového adresáře potvrdíte stiskem tlačítka `Next`.

Další dialogová stránka vypadá komplikovaně, ale ve skutečnosti není. V případě Pythonu 3 máte možnost neinstalovat úplně všechny jeho komponenty — podobně jako u jiných instalačních programů. Pokud máte obzvlášť málo místa na disku, můžete některé komponenty vynechat.

- Volba **Register Extensions** (asociovat přípony) vám zajistí možnost spouštět pythonovské skripty (soubory s příponou `.py`) poklepáním na jejich ikonu. Je to sice doporučeno, ale není to nezbytné. (Tato volba nevyžaduje žádný diskový prostor, takže její potlačení není výhodné.)
- **Tcl/Tk** je grafická knihovna, kterou využívá pythonovský shell. Ten budeme používat v celé knize. Velmi doporučuji, abyste tuto volbu ponechali zapnutou.
- Volba **Documentation** vede k instalaci souborů s nápovědou, která obsahuje mnohé z informací uvedených na `docs.python.org`. Pokud máte omezený přístup k internetu nebo pokud používáte vytáčené připojení, doporučuji volbu ponechat zapnutou.
- Volba **Utility Scripts** v sobě zahrnuje i instalaci skriptu `2to3.py`, o kterém se budeme učit v této knize později. Pokud se chcete naučit přepisování existujícího kódu napsaného pro Python 2 do podoby pro Python 3, pak se zapnutí této volby vyžaduje. Pokud nemáte žádné programy napsané pro Python 2, můžete tuto volbu vypnout.
- Volba **Test Suite** zajistí instalaci sady skriptů, které se používají pro testování funkčnosti interpretu jazyka Python. V této knize je nebudeme používat. A nepoužíval jsem je nikdy ani během výuky programování v Pythonu. Volba je zcela na vás.

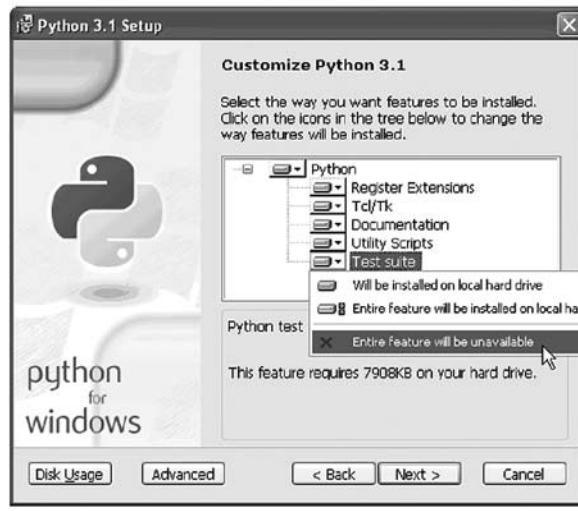
Pokud si nejste jisti, kolik máte místa na disku, klikněte na tlačítko `Disk Usage`. Instalátor zobrazí seznam písmen vašich disků, zjistí, kolik místa je na každém z nich, a vypočítá, kolik místa na nich zbude po instalaci.

Stiskem tlačítka `OK` se dostaneme na dialogovou stránku „Customizing Python“.



Pokud se rozhodnete volbu vynechat, stiskněte tlačítko pro rozbalení seznamu a vyberte „Entire feature will be unavailable“ (celá část bude nedostupná). Vynecháním Test Suite ušetříte na disku pěkných 7908 kb.

Výběr voleb potvrdíte stiskem tlačítka `Next`.

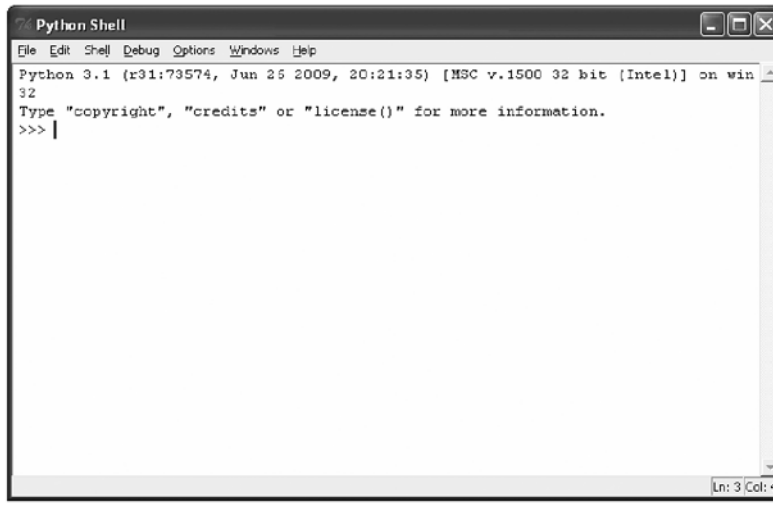


Instalátor nakopíruje všechny nezbytné soubory do vámi vybraného adresáře. (Proběhne to tak rychle, že jsem to musel zkusit třikrát, než se mi podařilo zachytit obrázek tohoto procesu.)

Stiskem tlačítka `Finish` ukončíme činnost instalátoru.

Ve vašem menu `Start` by se měla objevit položka s názvem `Python 3.1`. V ní se nachází program `IDLE`. Výběrem této položky spustíte interaktivní pythonovský shell. (Poznámka překladatele: Někdy ho autor označuje jako „grafický“ interaktivní shell. Jde o obdobu interaktivního pythonovského shellu, který se spouští v konzolovém okně. Tentokrát ale využívá prostředky grafického uživatelského

rozhraní (GUI) a v menu okna nalezneme i položky pro spuštění editoru nebo pro spuštění ladicího režimu. Dalo by se říct, že je to nástroj „téměř úplně, ale ne zcela naprosto nepodobný...“ klasickým IDE (integrované vývojové prostředí). Jenže to není soustředěné kolem editoru, ale spíš kolem shellu. Je to prostě IDLE. No zkrátka se na to podívejte a rozhodněte se sami, jak tomu budete říkat.)



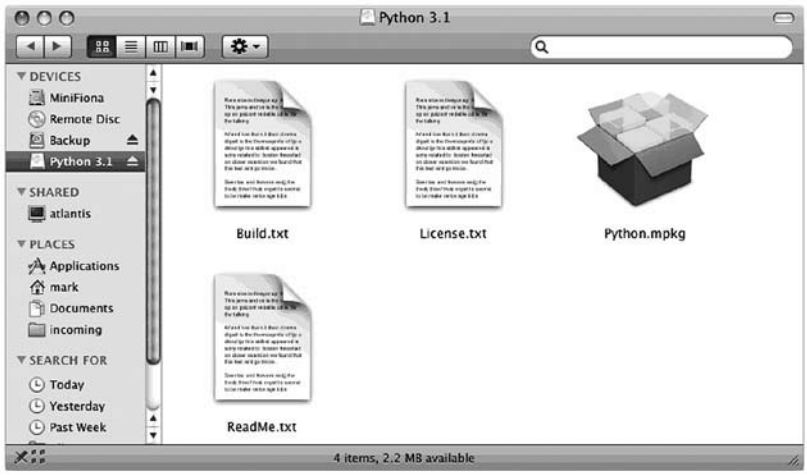
0.4. Instalace pod Mac OS X

Všechny moderní počítače Macintosh používají procesor firmy Intel (stejný jako většina osobních počítačů s Windows). Starší počítače Mac používají procesory PowerPC. Rozdílům rozumět nemusíte, protože existuje jen jeden jediný instalátor Pythonu pro všechny počítače Macintosh.

Přejděte na stránku `python.org/download/` a stáhněte si příslušný instalátor pro Mac. Bude u něj napsáno něco ve stylu **Python 3.1 Mac Installer Disk Image**, ačkoliv číslo verze se může lišit. Ujistěte se, že stahujete verzi 3.x a ne 2.x.

Váš prohlížeč by měl automaticky připojit obraz disku a otevřít okno Finder zobrazující jeho obsah. (Pokud se tak nestane, budete muset najít obraz disku ve svém adresáři pro stažené soubory a připojit jej poklepáním. Jmenuje se `python-3.1.dmg` nebo podobně.) Obraz disku obsahuje řadu textových souborů (`Build.txt, License.txt, ReadMe.txt`) a také skutečný instalační balík `Python.mpkg`.

Poklepejte na `Python.mpkg` a instalátor Mac Python se spustí.

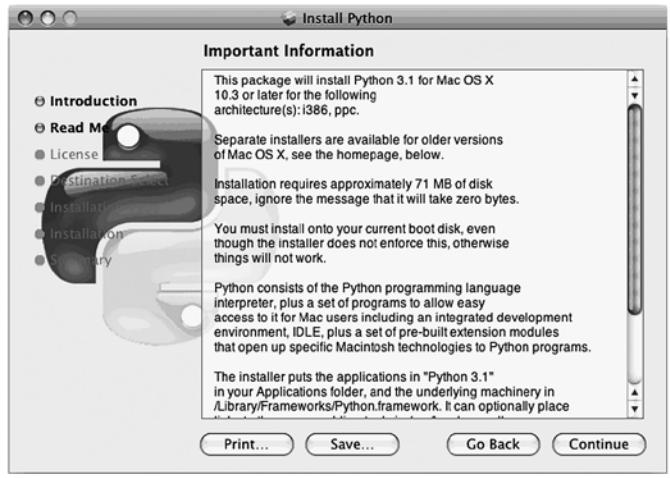


Na první stránce naleznete stručný popis jazyka Python a pro více detailů jste odkázáni na soubor `ReadMe.txt`. (...který jste nečetli. Nebo četli?)

Dál se posuneme stiskem tlačítka `Continue`.

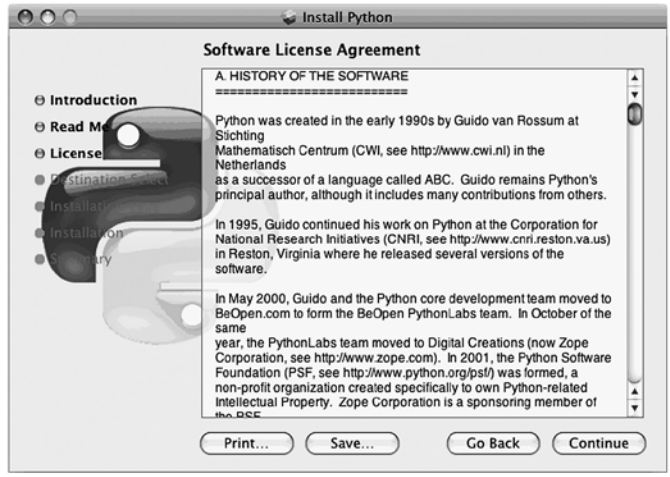
Následující stránka dialogu obsahuje některé důležité informace: Python vyžaduje Mac OS X 10.3 nebo novější. Pokud stále používáte Mac OS X 10.2, budete jej muset aktualizovat na vyšší verzi. Společnost Apple už pro váš operační systém neposkytuje bezpečnostní aktualizace a už při pouhém připojení na internet vystavujete svůj počítač riziku. A navíc nemůžete používat Python 3.

Pokračujeme stiskem tlačítka `Continue`.



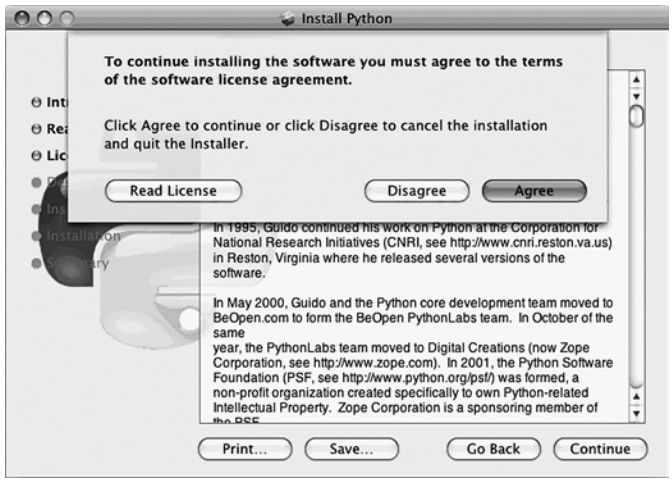
Tak jako všechny dobré instalátory, i ten pythonovský zobrazí licenční ujednání. Python je open source a jeho licence je schválena společností Open Source Initiative. Během historického vývoje měl Python řadu vlastníků a sponzorů. Každý z nich zanechal v jeho licenci svůj otisk. Ale konečný výsledek vypadá takto: Python je open source, můžete jej používat na libovolné platformě, pro libovolný účel, zdarma a bez závazku k protislužbě.

Stiskněte tlačítko `Continue` ještě jednou.



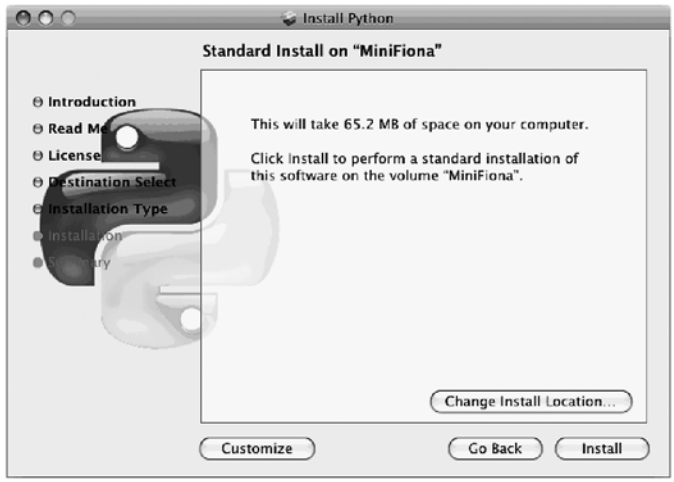
Abyste mohli instalaci dokončit, musíte kvůli manýrům v jádru applovského instalátoru projevit „souhlas“ se softwarovou licencí. Ale protože Python je open source, ve skutečnosti „souhlasíte“ s tím, že vám licence zaručuje práva navíc, než aby vás omezovala.

Pokračujeme stiskem tlačítka `Agree`.



Na další obrazovce můžete změnit umístění instalace. Python musíte instalovat na zaváděcí disk, ale kvůli omezením instalátoru to není vynuceno. Popravdě řečeno, nikdy jsem nepociťoval potřebu umístění instalace měnit.

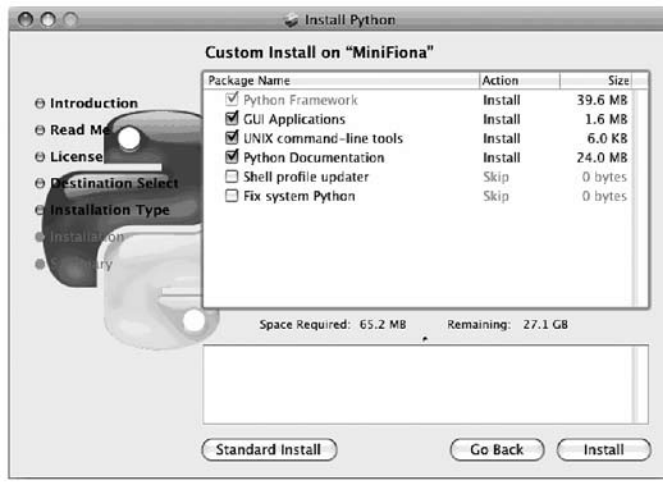
Na této obrazovce také můžete instalaci upravit vyloučením komponent, které nepotřebujete. Pokud tak chcete učinit, stiskněte tlačítko `Customize`. V opačném případě stiskněte tlačítko `Install`.



Pokud zvolíte uživatelskou úpravu instalace (`Custom Install`), nabídne vám instalátor následující seznam:

- **Python Framework.** Jde o jádro Pythonu. Proto je tato možnost předvolena a současně je zakázáno ji měnit. Tato část se nainstalovat musí.
- **GUI Applications** v sobě zahrnuje `IDLE`, což je grafický pythonovský shell. Budeme jej používat během celé knihy. Velmi doporučuji, abyste tuto volbu ponechali zapnutou.
- **UNIX command-line tools** v sobě obsahuje konzolovou aplikaci `python3`. Velmi doporučuji, abyste také tuto volbu ponechali zapnutou.
- **Python Documentation** obsahuje mnohé z informací uvedených na `docs.python.org`. Pokud máte omezený přístup k internetu nebo pokud používáte vytáčené připojení, doporučuji volbu ponechat zapnutou.
- **Shell profile updater** kontroluje, zda je nutné aktualizovat váš shellovský profil (použitý v `Terminal.app`) tak, aby bylo zajištěno, že umístění instalované verze Pythonu bude součástí prohledávaných cest. Tuto volbu pravděpodobně nebudete potřebovat měnit.
- Volbu **Fix system Python** byste měnit neměli. (Říká vašemu počítači, aby byl Python 3 použit jako preferovaný Python pro spouštění všech skriptů, včetně zabudovaných skriptů dodávaných firmou Apple. Dopadlo by to velmi špatně, protože většina těchto skriptů byla napsána pro Python 2 a pod verzí Python 3 by neběžely správně.)

Pokračujeme stiskem tlačítka `Install`.



Instalátor se vás zeptá na heslo správce, protože systémové binární soubory a nástroje se instalují do adresáře `/usr/local/bin/`. Bez administrátorských oprávnění Mac Python zkrátka nenainstalujete.

Stiskem tlačítka OK zahájíme instalaci.

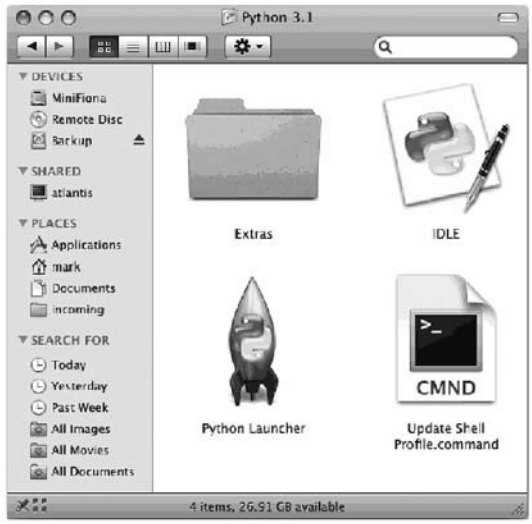
Během instalace částí, které jste si vybrali, instalátor indikuje postup instalace.

Pokud šlo všechno dobře, oznámí vám instalátor úspěšné dokončení instalace zobrazením zelené „fajfky“.

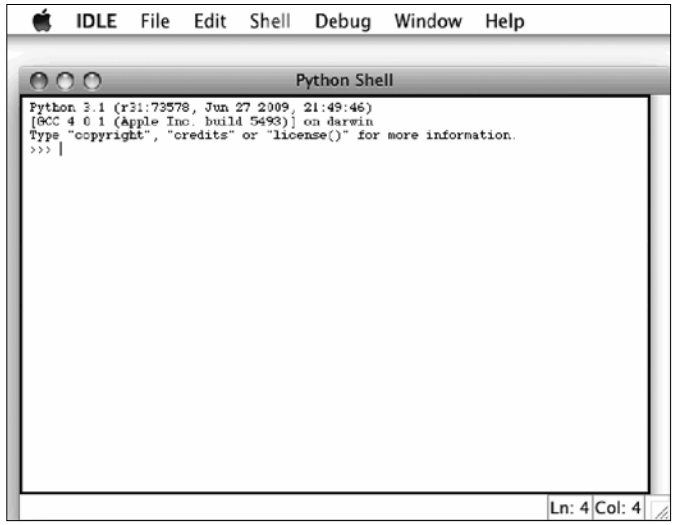
Stiskem tlačítka `Close` činnost instalátoru ukončíme.

Za předpokladu, že jste nezměnili umístění instalace, najdete nově nainstalované soubory v podadresáři `Python 3.1` uvnitř adresáře `/Applications`. Nejdůležitější součástí je zde grafický pythonovský shell zvaný idle.

Poklepejte na něj a pythonovský shell se spustí.

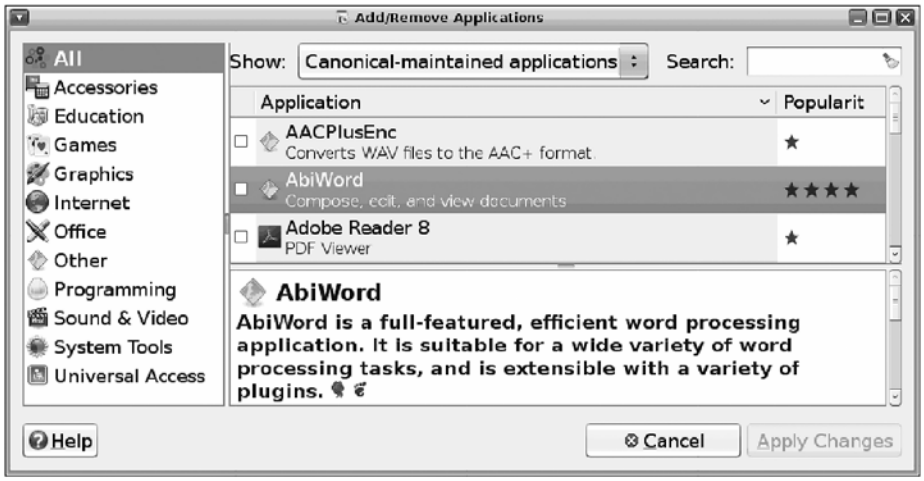


V pythonovském shellu strávíte při průzkumu jazyka Python nejvíce času. U příkladů budeme v této knize předpokládat, že se k pythonovskému shellu umíte dostat.



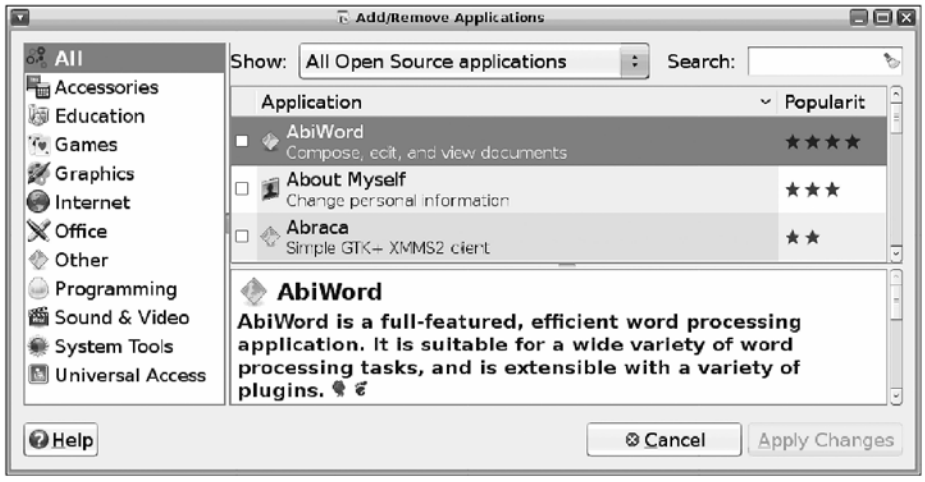
0.5. Instalace pod Ubuntu Linux

Moderní distribuce systému Linux jsou podepřeny ohromnými úložišti předkompilovaných aplikací, které jsou připraveny k okamžité instalaci. Detaily se pro konkrétní distribuce liší. Nejsnadnější způsob instalace Pythonu 3 pod Ubuntu Linux spočívá v použití nástroje `Add/Remove`, který najdete v menu `Applications`.



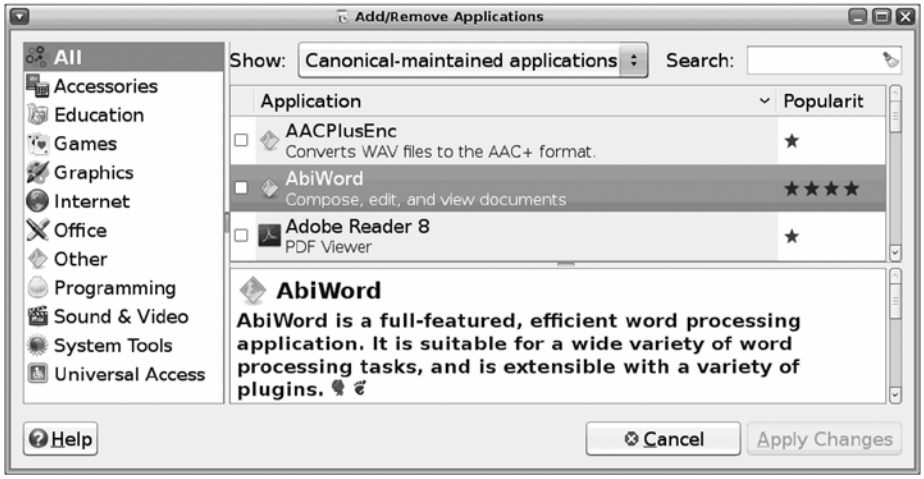
Když poprvé spustíte aplikaci `Add/Remove`, zobrazí vám seznam předvybraných aplikací v různých kategoriích. Některé z nich jsou již nainstalované, ale většina z nich ne. Protože úložiště obsahuje přes 10 tisíc aplikací, můžete pomocí různých filtrů omezit zobrazení jen na jeho malé části. Základem je filtr „Canonical-maintained applications“, což je malá podmnožina z celkového množství aplikací, které jsou oficiálně podporovány společností Canonical, která vytvořila a udržuje distribuci Ubuntu Linux.

Python 3 není společností Canonical udržován, takže jako první krok potlačíme činnost tohoto filtru a vybereme „All Open Source applications“ (všechny open source aplikace).



Jakmile změníte nastavení filtru tak, aby zahrnoval všechny open source aplikace, použijte k vyhledání `Pythonu 3` vyhledávací box nacházející se hned za nabídkou filtru.

V tom okamžiku se seznam aplikací zúží jen na ty, které souvisejí s Pythonem 3. Poté vybereme dva balíčky. Tím prvním je `Python (v3.0)`. Obsahuje vlastní interpret jazyka Python.



Druhý požadovaný balíček se nachází bezprostředně nad ním: `IDLE (using Python-3.0)`. Jde o grafický pythonovský shell, který budeme používat během celé knihy.

Po označení uvedených dvou balíčků pokračujte stiskem tlačítka `Apply Changes`.

Správce balíčků vás požádá o potvrzení, že chcete přidat jak `IDLE (using Python-3.0)`, tak `Python (v3.0)`. Pokračujeme stiskem tlačítka `Apply`.

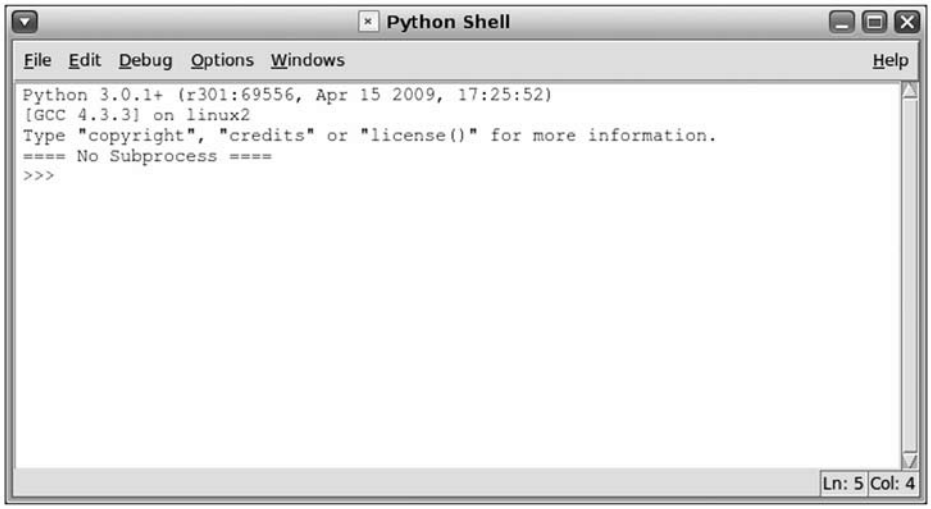
Během stahování potřebných balíčků z internetového úložiště společnosti Canonical zobrazuje správce balíčků indikátor postupu stahování.

Jakmile jsou balíčky staženy, zahájí správce balíčků automaticky jejich instalaci.

Pokud šlo všechno dobře, potvrdí správce balíčků, že byly oba úspěšně nainstalovány. V tomto okamžiku můžete poklepáním na `IDLE` spustit pythonovský shell, nebo můžete stiskem tlačítka `Close` ukončit činnost správce balíčků.

Pythonovský shell můžete spustit kdykoliv tím způsobem, že v menu `Applications` a v podmenu `Programming` vyberete `IDLE`.

V pythonovském shellu strávíte při průzkumu jazyka Python nejvíce času. U příkladů budeme v této knize předpokládat, že se k pythonovskému shellu umíte dostat.



0.6. Instalace na jiných platformách

Python 3 je dostupný pro řadu různých platforem. Abychom byli konkrétnější, je dostupný pro prakticky každou distribuci systému Linux, BSD a pro distribuce založené na systému Solaris. Takže například RedHat Linux používá správce balíčků yum. FreeBSD má svou sbírku ports and packages collection, SUSE má zypper a Solaris má pkgadd. Když zkusíte zběžně prohledat web při zadání `Python 3` + *váš operační systém*, dozvíte se, zda je balík s Pythonem 3 dostupný, a pokud ano, jak jej můžete nainstalovat.

0.7. Použití Python Shell

Python Shell (kvůli skloňování a zobecnění pohledu mu budeme říkat také *pythonovský shell*) bude nástrojem pro studium syntaxe jazyka Python, zdrojem interaktivní nápovědy k příkazům a prostředkem pro ladění krátkých programů. Grafický pythonovský shell (pojmenovaný `IDLE`) obsahuje navíc ucházející textový editor, který podporuje barevné zvýrazňování syntaxe a zajišťuje spolupráci s (konzolovým) pythonovským shellem. Pokud již nemáte nějaký svůj oblíbený textový editor, měli byste si `IDLE` vyzkoušet.

Ale proberme nejdříve hlavní věci. Samotný Python Shell je úžasné interaktivní prostředí, se kterým si vyhrajete. V celé knize se budete setkávat s příklady, jako je tento:

```
>>> 1 + 1
2
```

Tři úhlové závorky (`>>>`) jsou vyzývacím řetězcem pythonovského shellu. Tuto část neopisujte. Vyjadřuji tím to, že byste si příklad měli vyzkoušet v pythonovském shellu.

Vy budete psát pouze část `1 + 1`. V pythonovském shellu můžete napsat jakýkoliv platný pythonovský výraz nebo příkaz. Nestyďte se! Nekousne vás to! Přinejhorším se stane to, že se vám zobrazí chybové hlášení. Příkazy se provádějí okamžitě (jakmile stisknete `ENTER`). Také výrazy jsou vyhodnoceny okamžitě a pythonovský shell vytiskne jejich výsledek.

Takže zobrazená část `2` je výsledkem vyhodnocení předchozího výrazu. Protože se tak stalo, je `1 + 1` zjevně platným pythonovským výrazem. Jeho výsledek je samozřejmě `2`.

Vyzkoušejme něco dalšího.

```
>>> print('Hello world!')
Hello world!
```

Docela jednoduché, že? Ale v pythonovském shellu toho můžete dělat mnohem víc. Když se někdy zadrhnete — když si nemůžete vzpomenout na nějaký příkaz nebo si nemůžete vzpomenout na správné argumenty předávané nějaké funkci —, můžete se v pythonovském shellu dostat k interaktivní nápovědě. Napište prostě `help` a stiskněte `ENTER`.

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

Nápovědu můžeme používat ve dvou režimech. Můžeme získat nápovědu pro jeden objekt. Vytiskne se prostě jeho dokumentace a vrátíte se na vyzývací řádek pythonovského shellu. Nebo můžeme vstoupit do *režimu nápovědy*, ve kterém místo vyhodnocování pythonovských výrazů píšeme klíčová slova nebo jména příkazů a Python zobrazuje vše, co o těchto příkazech ví.

Pro vstup do interaktivního režimu nápovědy napište `help()` a stiskněte ENTER.

```
>>> help()
Welcome to Python 3.0!  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://docs.python.org/tutorial/.

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>
```

Všimněte si, že se vyzývací řetězec změnil z `>>>` na `help>`. Má vám to připomenout, že se nacházíte v interaktivním režimu nápovědy. V tomto okamžiku můžete napsat libovolné klíčové slovo, příkaz, jméno modulu, jméno funkce — v podstatě cokoliv, čemu Python rozumí — a přečtete si k tomu zobrazenou dokumentaci.

```
help> print                                                              [1]
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.

help> PapayaWhip                                                         [2]
no Python documentation found for 'PapayaWhip'

help> quit                                                               [3]

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.
>>>                                                                    [4]
```

[1] Abyste dostali dokumentaci k funkci `print()`, napište print a stiskněte ENTER. V interaktivním režimu nápovědy se zobrazí něco podobného jako manovská stránka: jméno funkce, stručný popis, argumenty funkce a jejich přednastavené hodnoty a tak dále. Pokud se vám zdá obsah dokumentace nejasný, nepropadejte panice. V následujících několika kapitolách se o těchto věcech dozvíte více.

[2] V interaktivním režimu nápovědy se samozřejmě nedozvíte všechno. Pokud zde napíšete něco, co není pythonovským příkazem, modulem, funkcí nebo nějakým zabudovaným klíčovým slovem, režim interaktivní nápovědy prostě pokrčí svými virtuálními rameny.

[3] Interaktivní režim nápovědy ukončíte tím, že napíšete `quit` a stisknete ENTER.

[4] Vyzývací řádek se změní zpět na `>>>`, čímž se dozvíte, že jste opustili režim interaktivní nápovědy a vrátili jste se do pythonovského shellu.

Grafický pythonovský shell IDLE navíc obsahuje textový editor šitý na míru jazyku Python.

0.8. Editory a vývojová prostředí pro Python

Pokud jde o psaní programů v jazyce Python, nepředstavuje idle jedinou možnost. Jakkoliv může být užitečný při seznamování se s jazykem jako takovým, mnozí vývojáři dávají přednost jiným textovým editorům nebo integrovaným vývojovým prostředím (Integrated Development Environment, čili IDE). Nebudu se zde jimi zabývat, ale komunita uživatelů jazyka Python udržuje seznam editorů podporujících jazyk Python, který pokrývá široké rozpětí podporovaných platforem a softwarových licencí.

Možná chcete nahlédnout i do seznamu ide podporujících jazyk Python, i když zatím pouze nemnohé z nich podporují Python 3. Jedním z těch, které jej podporují, je PyDev, zásuvný modul pro Eclipse, který změní Eclipse na plnohodnotné pythonovské integrované vývojové prostředí. Jak Eclipse, tak PyDev jsou multiplatformní a open source.

Z komerčních produktů jmenujme Komodo IDE společnosti ActiveState. Licence je vázána na uživatele. Studenti mohou získat slevu a k dispozici je i zkušební, časově omezená verze.

V jazyce Python programuji už devět let. Své programy edituji v prostředí GNU Emacs a ladím je v konzolovém pythonovském shellu. Při vývoji v jazyce Python není žádná cesta správnější nebo vyloženě špatná. Najděte si způsob, který vyhovuje právě vám!

> **“Don’t bury your burden in saintly silence. You have a problem? Great. Rejoice, dive in, and investigate.”**
> (Neutápějte své břímě ve svatém mlčení. Máte problém? Paráda. Radujte se, ponořte se do něj, bádejte.)
>
> — Ven. Henepola Gunaratana

1. Váš první pythonovský program

1.1. Ponořme se

Konvence nám diktuje, že bych vás teď měl otravovat základními stavebními kameny, které s programováním souvisejí. A z nich bychom pak měli pomalu budovat něco užitečného. Přeskočme to. Tady máte úplný a funkční pythonovský program. Pravděpodobně vám bude zcela nepochopitelný. Žádné strachy. Rozpitváme ho řádek po řádku. Ale nejdříve si jej celý přečtěte a zjistěte, co z něj chápete (pokud vůbec něco).

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
    if size < 0:
        raise ValueError('number must be non-negative')

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('number too large')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

Spusťme program z příkazového řádku. Pod Windows to bude vypadat nějak takto:

```
c:\home\diveintopython3\examples> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

Pod Mac OS X nebo pod Linuxem to bude vypadat zase takhle:

```
you@localhost:~/diveintopython3/examples$ python3 humansize.py
1.0 TB
931.3 GiB
```

Co se to vlastně stalo? Spustili jste svůj první pythonovský program. Z příkazového řádku jste zavolali interpret jazyka Python a předali jste mu jméno skriptu, který měl být proveden. Uvedený skript definuje jedinou funkci, `approximate_size()`, která přebírá přesnou velikost souboru v bajtech a vypočítá velikost „v hezčím tvaru“ (ale přibližnou). (Pravděpodobně už jste něco podobného viděli v Průzkumníku Windows, v okně Finder na Mac OS X nebo v aplikacích Nautilus nebo Dolphin nebo Thunar na Linuxu. Když si necháte složku s dokumenty zobrazit v podobě vícesloupcového seznamu, uvidíte v tabulce ikonu dokumentu, jméno dokumentu, velikost, typ, datum poslední změny a tak dále. Pokud složka obsahuje soubor se jménem `TODO` a s velikostí 1093 bajtů, nezobrazí váš správce souborů `TODO 1093 bytes`. Místo toho se ukáže něco jako `TODO 1 KB`. A právě tohle dělá funkce `approximate_size()`.)

Podívejte se na konec skriptu a uvidíte dva řádky s voláním `print(approximate_size(argumenty))`. Jde o volání funkcí. Nejdříve se volá funkce `approximate_size()` a předávají se jí argumenty. Její návratová hodnota se předává přímo funkci `print()`. Funkce `print()` patří mezi zabudované (built-in). Její deklaraci nikdy neuvidíte. Můžete ji ale používat — kdykoliv a kdekoliv. (Zabudovaných funkcí existuje celá řada. A ještě mnohem více se jich nachází v různých *modulech*. Jen klid...)

Takže proč vlastně spuštěním skriptu z příkazového řádku získáme pokaždé stejný výstup? K tomu se ještě dostaneme. Nejdříve se podíváme na funkci `approximate_size()`.

1.2. Deklarace funkcí

Python pracuje s funkcemi podobně jako většina dalších jazyků, ale neodděluje hlavičkové soubory jako `c++` nebo sekce `rozhraní/implementace` jako Pascal. Pokud potřebujete nějakou funkci, prostě ji deklarujete, jako třeba zde:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

Deklarace funkce začíná klíčovým slovem `def`. Následuje jméno funkce a v závorce pak argumenty. Více argumentů se odděluje čárkami.

Pokud potřebujete nějakou funkci, prostě ji deklarujte.

Všimněte si, že funkce nedefinuje typ návratové hodnoty. Funkce v jazyce Python neurčují datový typ návratové hodnoty. Neurčují dokonce ani to, jestli vracejí hodnotu nebo ne. (Ve skutečnosti každá pythonovská funkce vrací hodnotu. Pokud funkce provede příkaz `return`, vrátí v něm uvedenou

hodnotu. V ostatních případech vrací `None`, což je pythonovský ekvivalent hodnoty null, nil, nic, žádná hodnota.)

> V některých jazycích funkce (které vracejí hodnotu) začínají slovem `function` a podprogramy (které nevracejí hodnotu) začínají slovem `sub`. Jazyk Python žádné podprogramy nezná. Vše jsou funkce, všechny funkce vracejí hodnotu (i když někdy je to `None`) a všechny funkce začínají slovem `def`.

Funkce `approximate_size()` přebírá dva argumenty — `size` a `a_kilobyte_is_1024_bytes` —, ale u žádného z nich není určen datový typ. V jazyce Python nemají proměnné explicitně určen typ nikdy. Python zjistí, jakého typu proměnná je, a vnitřně si to eviduje.

> V jazyce Java a v dalších jazycích se statickými datovými typy musíme určovat datový typ návratové hodnoty funkce a každého argumentu funkce. V jazyce Python nikdy explicitně neurčujeme datový typ čehokoliv. Python vnitřně sleduje datový typ podle toho, jakou hodnotu jsme přiřadili.

1.2.1. Nepovinné a pojmenované argumenty

Python umožňuje nastavit argumentům funkce implicitní hodnotu. Pokud funkci zavoláme bez zadání argumentu, získá argument svou implicitní hodnotu. Pokud použijeme pojmenované argumenty, můžeme je navíc (při volání funkce) zadat v libovolném pořadí.

Teď se na deklaraci funkce `approximate_size()` podíváme ještě jednou:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

U druhého argumentu, `a_kilobyte_is_1024_bytes`, je uvedena implicitní hodnota `True`. To znamená, že tento argument je *nepovinný*. Funkci můžeme zavolat, aniž bychom ho zadali. Python se bude chovat, jako kdybychom při volání funkce zadali na místě druhého argumentu hodnotu `True`.

Teď se podívejte na konec skriptu:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))               [1]
    print(approximate_size(1000000000000))                      [2]
```

[1] Zde se funkce `approximate_size()` volá s dvěma argumenty. Protože jsme druhému argumentu explicitně předali hodnotu `False`, nabývá `a_kilobyte_is_1024_bytes` uvnitř funkce `approximate_size()` hodnotu `False`.

[2] Zde se funkce `approximate_size()` volá pouze s jedním argumentem. Ale je to v pořádku, protože druhý argument je volitelný! A protože ho volající neurčil, nabývá druhý argument implicitní hodnoty `True` — přesně jak bylo určeno v deklaraci funkce.

Hodnotu argumentu můžeme do funkce předat také jako pojmenovanou.

```
>>> from humansize import approximate_size
>>> approximate_size(4000, a_kilobyte_is_1024_bytes=False)       [1]
'4.0 KB'
>>> approximate_size(size=4000, a_kilobyte_is_1024_bytes=False)  [2]
'4.0 KB'
>>> approximate_size(a_kilobyte_is_1024_bytes=False, size=4000)  [3]
'4.0 KB'
>>> approximate_size(a_kilobyte_is_1024_bytes=False, 4000)       [4]
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> approximate_size(size=4000, False)                           [5]
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

[1] Zde se funkce `approximate_size()` volá s hodnotou prvního argumentu `4000` (`size`) a s hodnotou `False` pro pojmenovaný argument `a_kilobyte_is_1024_bytes`. (Shodou okolností je to druhý argument, ale na tom nezáleží — jak uvidíte o chvíli později.)

[1] Zde se funkce `approximate_size()` volá s hodnotou `4000` pro pojmenovaný argument `size` a s hodnotou `False` pro pojmenovaný argument `a_kilobyte_is_1024_bytes`. (Pojmenované argumenty jsou zde shodou okolností uvedeny ve stejném pořadí, v jakém jsou uvedeny v deklaraci funkce, ale na tom rovněž nezáleží.)

[3] Zde se funkce `approximate_size()` volá s hodnotou `False` pro pojmenovaný argument `a_kilobyte_is_1024_bytes` a s hodnotou `4000` pro pojmenovaný argument `size`. (Vidíte? Já jsem vám říkal, že na pořadí nezáleží.)

[4] Toto volání selhalo, protože jsme použili pojmenovaný argument a teprve po něm následoval nepojmenovaný (poziční) argument. Tohle nefunguje nikdy. Při čtení seznamu argumentů zleva doprava se po použití prvního pojmenovaného argumentu musí všechny následující argumenty uvést také jako pojmenované.

[5] Toto volání rovněž selhává — ze stejného důvodu jako předchozí volání. Je to tak překvapivé? Když se to tak vezme, předáváme hodnotu `4000` pro pojmenovaný argument `size` a je „zřejmé“, že hodnota `False` byla myšlena jako hodnota argumentu `a_kilobyte_is_1024_bytes`. Ale Python tímto způsobem nefunguje. Jakmile použijeme pojmenovaný argument, všechny argumenty uvedené napravo od něj musí být také pojmenované.

1.3. Psaní čitelného kódu

Nebudu vás zde nudit dlouhým proslovem o důležitosti dokumentování vašeho kódu. Jen si uvědomte, že kód se píše jednou, ale čte se mnohokrát. A nejdůležitějším čtenářem vašeho zdrojového textu budete vy sami — šest měsíců poté, co jste jej napsali (to znamená poté, co už jste o něm všechno zapomněli a máte v něm něco opravit). V jazyce Python se čitelný kód píše snadno, takže toho využijte. Za šest měsíců mi poděkujete.

1.3.1. Dokumentační řetězce

Pythonovskou funkci můžete zdokumentovat tím, že jí přidělíte dokumentační řetězec (zkráceně `docstring`). V našem programu je u funkce `approximate_size()` dokumentační řetězec uveden:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
```

Každá funkce si zaslouží decentní docstring.

Tři apostrofy uvozují víceřádkový řetězec. Vše mezi počátečními a koncovými apostrofy (nebo uvozovkami) se stává součástí jediného řetězce, včetně konců řádků, úvodních bílých znaků a jednoduchých apostrofů. Víceřádkové řetězce můžete použít kdekoliv, ale nejčastěji se s nimi setkáte při zápisech dokumentačních řetězců.

> Použití ztrojených apostrofů představuje rovněž jednoduchý způsob pro zápis řetězců, ve kterých se vyskytují jak apostrofy, tak uvozovky. Chovají se jako zápis `qq/.../` v jazyce Perl 5.

Vše, co se nachází mezi ztrojenými apostrofy, je dokumentační řetězec, který popisuje, co funkce dělá. Pokud `docstring` existuje, pak to musí být první věc, která se v těle funkce objeví. (To znamená, že musí být uveden na řádku následujícím za deklarací funkce.) Z technického pohledu není nutné `docstring` funkci vůbec přidělovat, ale prakticky byste to měli udělat vždy. Já vím, že jste o tom slyšeli v každém kurzu programování, který jste navštěvovali. Ale u jazyka Python máme jeden motivační faktor navíc: `docstring` je dostupný za běhu programu v podobě atributu (vlastnosti) funkce.

> Mnohá pythonovská integrovaná vývojová prostředí používají docstring pro účely kontextově citlivé nápovědy. To znamená, že po napsání jména funkce se její docstring zobrazí v podobě tooltipu (tj. malého informačního okénka zobrazovaného poblíž daného místa). Může to být velmi užitečné, ale bude to dobré jen tak, jak dobře napíšete dokumentační řetězce.

1.4. Vyhledávací cesta pro import

Než půjdeme dál, chtěl bych se stručně zmínit o vyhledávací cestě pro knihovny (library search path). Když se pokoušíte importovat modul, hledá jej Python na několika místech. Přesněji řečeno, hledá jej ve všech adresářích, které jsou definovány proměnnou sys.path. Jde o běžný seznam a jeho obsah můžete snadno zobrazit nebo měnit prostřednictvím standardních metod seznamu. (O seznamech se dozvíme více v kapitole Přirozené datové typy.)

```
>>> import sys                                                  [1]
>>> sys.path                                                    [2]
['',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
>>> sys                                                         [3]
<module 'sys' (built-in)>
>>> sys.path.insert(0, '/home/mark/diveintopython3/examples')   [4]
>>> sys.path                                                    [5]
['/home/mark/diveintopython3/examples',
 '',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
```

[1] Importováním modulu sys zpřístupníme všechny jeho funkce a atributy.

[2] sys.path je seznam adresářů, které tvoří aktuální vyhledávací cestu. (U vás to bude vypadat jinak v závislosti na vašem operačním systému, na verzi Pythonu, který používáte, a na tom, kam byl nainstalován.) Pokud se pokoušíte o import, hledá Python soubor s daným jménem a příponou .py právě v těchto adresářích (v uvedeném pořadí).

[3] No, ve skutečnosti jsem trochu zalhal. Pravda je o něco komplikovanější, protože ne všechny moduly jsou uloženy v podobě souborů s příponou .py. U některých jde o *zabudované (built-in)*

moduly. Ve skutečnosti jsou součástí programu Python. Zabudované moduly se chovají úplně stejně jako běžné moduly, ale není k nim k dispozici pythonovský zdrojový kód, protože nejsou napsány v jazyce Python! Zabudované moduly jsou napsány v jazyce C, stejně jako samotný Python.

[4] K pythonovské vyhledávací cestě můžete za běhu přidat nový adresář tím, že jeho jméno přidáte do sys.path. Kdykoliv se od toho okamžiku pokusíte importovat nějaký modul, Python bude prohledávat i tento adresář. Efekt trvá tak dlouho, dokud Python běží.

[5] Použitím příkazu `sys.path.insert(0, new_path)` jsme vložili nový adresář jako první položku seznamu `sys.path`, což znamená, že se ocitla na začátku pythonovské vyhledávací cesty. Většinou potřebujeme právě tohle. V případě konfliktu jmen (například když se Python dodává s konkrétní knihovnou verze 2, ale my chceme použít tutéž knihovnu ve verzi 3) uvedeným obratem zajistíme, že námi požadované moduly budou nalezeny dříve než moduly dodané s Pythonem.

1.5. Všechno je objekt

Pokud vám to náhodou uniklo, řekli jsme si, že pythonovské funkce mají atributy a tyto atributy jsou přístupné za běhu programu. Funkce, stejně jako všechno ostatní v Pythonu, je objektem.

Spusťme interaktivní pythonovský shell a vyzkoušejme si:

```
>>> import humansize                                [1]
>>> print(humansize.approximate_size(4096, True))  [2]
4.0 KiB
>>> print(humansize.approximate_size.__doc__)      [3]
Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string
```

[1] Na prvním řádku importujeme program `humansize` jako modul — kus kódu, který můžeme používat interaktivně nebo z většího pythonovského programu. Jakmile je import modulu proveden, můžeme se odkazovat na jeho veřejné funkce, třídy nebo atributy. Moduly mohou dělat totéž, čímž si zpřístupňují funkčnost z jiných modulů. A my to můžeme udělat v interaktivním pythonovském shellu také. Tato koncepce je důležitá a v knize se s ní potkáme ještě mnohokrát.

[2] Pokud chceme použít funkce definované v importovaných modulech, musíme uvést i jméno modulu. Takže nestačí napsat jen `approximate_size`. Musíme uvést `humansize.approximate_size`. Pokud jste používali třídy v jazyce Java, mělo by vám to něco připomínat.

[3] Zde se místo očekávaného volání funkce ptáme na jeden z jejích atributů, který je nazván `__doc__`.

> Pythonovský příkaz `import` se podobá příkazu `require` v jazyce Perl. Jakmile provedeme `import` pythonovského modulu, vyjadřujeme přístup k jeho funkcím zápisem `modul.funkce`. Jakmile v jazyce Perl provedeme příkaz `require`, dostaneme se na jeho funkce zápisem `modul::funkce`.

1.5.1. Co to vlastně je objekt?

V Pythonu je objektem všechno. A vše může mít atributy a metody. Všechny funkce mají zabudovaný atribut `__doc__`, který vrací dokumentační řetězec funkce definovaný ve zdrojovém souboru. Modul `sys` je objekt, který (mimo jiné) má atribut zvaný `path`. A tak dále.

Tím ale stále neodpovídáme na základnější otázku: Co je to vlastně objekt? Různé programovací jazyky definují „objekt“ různým způsobem. V některých jazycích to znamená, že *všechny* objekty *musí* mít atributy a metody. V jiných jazycích to znamená, že všechny objekty lze rozdělit do tříd. Jazyk Python definuje objekt volněji. Některé objekty nemusí mít ani atributy ani metody, *ale mohou je mít*. Ne všechny objekty mají svou třídu. Ale vše je objektem v tom smyslu, že to může být přiřazeno do proměnné nebo předáno jako argument funkce.

V jiných souvislostech s programováním jste už možná slyšeli pojem „prvotřídní objekt“ („first-class object“). Kvůli lepší srozumitelnosti mu říkejme (opisem) *plnohodnotný objekt*. V jazyce Python je *plnohodnotným objektem* i funkce. Funkci můžeme předat jako argument jiné funkci. Moduly jsou rovněž *plnohodnotnými objekty*. Funkci můžeme předat jako argument celý modul. Třídy jsou také plnohodnotné objekty a jednotlivé instance třídy jsou rovněž plnohodnotnými objekty.

To je velmi důležité, takže pro případ, že by vám to na začátku párkrát uteklo, zopakuji znovu: *V jazyce Python je všechno objektem.* Řetězce jsou objekty. Seznamy jsou objekty. Funkce jsou objekty. Třídy jsou objekty. Instance tříd jsou objekty. Dokonce moduly jsou objekty.

1.6. Odsazování kódu

V jazyce Python se pro označování míst, kde kód funkce začíná a kde končí, nepoužívají slova `begin` a `end` a ani žádné složené závorky. Jediným oddělovačem těla je dvojtečka (`:`) a odsazení kódu.

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):  [1]
    if size < 0:                                            [2]
        raise ValueError('number must be non-negative')     [3]
                                                            [4]
```

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
 for suffix in SUFFIXES[multiple]:                          [5]
     size /= multiple
     if size < multiple:
         return '{0:.1f} {1}'.format(size, suffix)

 raise ValueError('number too large')
```

[1] Bloky kódu (bloky zdrojového textu) jsou určeny jejich odsazením. „Blokem kódu“ zde rozumím volání funkcí, příkazy `if`, cykly `for`, cykly `while` a další. Blok je zahájen odsazením (odskočením řádku vpravo) a končí předsazením (odskočením následujícího řádku vlevo). Nenajdeme zde žádné explicitní závorky nebo klíčová slova. To ale znamená, že používání bílých znaků má svůj význam a že je musíme užívat důsledně. V tomto příkladu je kód funkce odsazen o čtyři mezery. Nemusí to být zrovna čtyři mezery, ale musíme použít stejné odsazení. První řádek, který není odsazený, označuje konec funkce.

[2] V Pythonu za příkazem `if` následuje blok kódu. Pokud výraz za `if` nabývá hodnoty `true`, provede se následující odsazený blok. V opačném případě se provede blok za `else` (pokud je uveden). Povšimněte si, že kolem výrazu chybí závorky.

[3] Tento řádek se nachází v bloku kódu, který je uvnitř příkazu `if`. Příkaz `raise` vyvolá výjimku (typu `ValueError`), ale jen v případě, kdy platí `size < 0`.

[4] Zde ještě není konec funkce. Zcela prázdné řádky se nepočítají. Díky nim může být kód čitelnější, ale nepovažují se za oddělovače bloků kódu. Na dalším řádku funkce pokračuje.

[5] Rovněž příkaz cyklu `for` zahajuje blok kódu. Bloky kódu se mohou skládat z mnoha řádků, ale všechny musí být odsazeny stejně. Tento cyklus `for` má blok s třemi řádky kódu. Pro víceřádkové bloky kódu se nepoužívá žádná jiná zvláštní syntaxe. Prostě odsadíme a jedeme dál.

Po počátečních protestech a sarkastických přirovnáních k Fortranu si na to zvyknete a zjistíte, jaké to má výhody. Jedna z největších výhod spočívá v tom, že všechny pythonovské programy vypadají podobně, protože odsazování je vynuceno samotným jazykem a není jen věcí stylu. Pythonovský kód napsaný někým jiným se proto snadněji čte a je srozumitelnější.

> Python používá k oddělování příkazů konec řádku. Oddělení bloku kódu se vyjadřuje dvojtečkou a odsazením. Jazyky `c++` a Java používají k oddělování příkazů středník a k oddělování bloku kódu složené závorky.

1.7. Výjimky

V jazyce Python najdete výjimky všude. Používá je prakticky každý modul standardní pythonovské knihovny a samotný Python je vyvolává při mnoha různých okolnostech. V celé této knize se s nimi budete opakovaně setkávat.

Co to vlastně je výjimka? Obvykle jde o projev nějaké chyby. Vyjadřuje, že něco nedopadlo dobře. (Ne všechny výjimky jsou vyjádřením chyby. Ale v tomto okamžiku na tom nezáleží.) V některých programovacích jazycích jsme vedeni k používání návratových chybových kódů, které pak *kontrolujeme*. Python nás vede k používání výjimek, které pak *obsluhujeme*.

Když se v pythonovském shellu objeví chyba, vypíše nějaké podrobnosti o výjimce a jak k ní došlo. A to je právě ono. Říkáme tomu *neobsloužená* výjimka. V okamžiku vyvolání výjimky se v okolí nenacházel žádný kód, který by si toho všímal a který by se jí zabýval. Takže výjimka probublala zpět až do horních úrovní pythonovského shellu. Ten vyplivnul nějaké ladicí informace a považoval to za vyřešené. Pokud se to stane při práci v shellu, není to žádná pohroma. Ale pokud by se to stalo u vašeho skutečného pythonovského programu, pak by za předpokladu, že výjimku nic neobsloužilo, došlo ke skřípavému zastavení jeho běhu. Možná by vám to vyhovovalo, možná ne.

> V Pythonu nemusí funkce deklarovat, jaké výjimky mohou vyvolat — na rozdíl od jazyka Java. Rozhodnutí o tom, jaké možné výjimky potřebujete odchytávat, záleží zcela na vás.

Ale výjimka nemusí vést k úplnému krachu programu. Výjimky mohou být *obslouženy*. Někdy je výjimka opravdu důsledkem chyby ve vašem programu (když se například pokoušíte použít proměnnou, která neexistuje), ale někdy je výjimka výsledkem něčeho, co se dalo předvídat. Když otvíráte soubor, nemusí třeba existovat. Když importujete modul, nemusel být nainstalován. Když se připojujete k databázi, může být nedostupná nebo k ní nemůžete přistupovat kvůli nedostatečným bezpečnostním oprávněním. Pokud víte, že na nějakém řádku může vzniknout výjimka, měli byste ji obsloužit pomocí konstrukce `try...except`.

> Python používá bloky `try...except` k obsluze výjimek. Příkaz `raise` používá k jejich generování. Jazyky Java a c++ používají k obsloužení výjimek bloky `try...catch`. K jejich generování používají příkaz `throw`.

Funkce `approximate_size()` vyvolává výjimky ve dvou různých případech: když je zadaná velikost (`size`) větší, než pro jakou byla funkce navržena, nebo když je zadaná velikost menší než nula.

```
if size < 0:
    raise ValueError('number must be non-negative')
```

Syntaxe pro vyvolání výjimky je poměrně jednoduchá. Použijeme příkaz `raise`, za kterým uvedeme jméno výjimky a nepovinný, pro člověka srozumitelný řetězec usnadňující ladění. Zápis se podobá volání funkce. (Ve skutečnosti jsou výjimky implementovány jako třídy. Příkaz raise zde vytváří instanci třídy `ValueError` a její inicializační metodě předává řetězec `'number must be non-negative'` (číslo nesmí být záporné). Ale nepředbíhejme!)

> Výjimka nemusí být obsloužena ve funkci, která ji vyvolala. Pokud ji jedna funkce neobslouží, výjimka bude předána volající funkci, pak funkci, která vyvolala zase ji a tak dále, „nahoru po zásobníku“. Pokud není výjimka obsloužena vůbec, program zhavaruje a Python vypíše „traceback“

(trasovací výpis) na standardní chybový výstup a tím to končí. Znovu opakuji, možná takové chování požadujeme. Záleží to na tom, k čemu je náš program určen.

1.7.1. Obsluha chyb importu

Jednou ze zabudovaných výjimek jazyka Python je `ImportError`. Ta je vyvolána v okamžiku, kdy se pokoušíme o import modulu a tato operace selže. Může k tomu dojít z různých důvodů, ale v nejjednodušším případě modul nebyl nalezen ve vaší vyhledávací cestě pro import. Toho můžete využít pro zabudování nepovinných vlastností svého programu. Tak například knihovna `chardet` umožňuje autodetekci znakového kódování. Možná byste chtěli, aby váš program tuto knihovnu využil *v případě, že existuje.* Pokud ji uživatel nemá nainstalovanou, měl by program bez mrknutí oka pokračovat. Můžeme toho dosáhnout použitím bloku `try..except`.

```
try:
  import chardet
except ImportError:
  chardet = None
```

Později můžete otestovat, zda je modul `chardet` přítomen — jednoduše, příkazem `if`:

```
if chardet:
  # do something
else:
  # continue anyway
```

Další běžný případ použití výjimky `ImportError` souvisí se situací, kdy dva moduly implementují společné aplikační programové rozhraní (`API`), ale jeden z nich chceme používat přednostně. (Možná je rychlejší nebo používá méně paměti.) Můžeme zkusit importovat jeden modul, ale pokud import selže, vezmeme zavděk tím druhým. Tak například kapitola o XML pojednává o dvou modulech, které implementují společné rozhraní zvané `ElementTree`. Prvním z nich je lxml, což je modul třetí strany, který si musíte sami stáhnout a nainstalovat. Tím druhým je xml.etree.ElementTree, který je sice pomalejší, ale je součástí standardní knihovny jazyka Python 3.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

Na konci bloku `try..except` máte zpřístupněný některý z těchto modulů a máte jej pojmenovaný `etree`. Protože oba moduly implementují stejné rozhraní (`API`), nemusíte ve zbytku svého kódu neustále testovat, který modul se vlastně naimportoval. A protože se modul, který se opravdu naimportoval, vždy jmenuje `etree`, nemusí být zbytek vašeho kódu zaneřáděný příkazy `if`, ve kterých se volají různě pojmenované moduly.

1.8. Volné proměnné

Podívejme se znovu na následující řádek kódu funkce `approximate_size()`:

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

Proměnnou `multiple` (násobek) jsme nikde nedeklarovali. Pouze jsme do ní přiřadili hodnotu. To je v pořádku, protože Python vám tohle dovolí. Co už vám ale Python *nedovolí*, je pokus o odkaz na proměnnou, které nebyla nikdy přiřazena hodnota. Pokud se o to pokusíme, bude vyvolána výjimka `NameError`.

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

Jednoho dne za to Pythonu poděkujete.

1.9. Vše je citlivé na velikost písmen

V jazyce Python je zápis všech jmen citlivý na velikost písmen. Týká se to jmen proměnných, jmen funkcí, jmen tříd, jmen modulů, jmen výjimek. Pokud to můžete zpřístupnit, nastavit, zavolat, importovat nebo to vyvolat, je to citlivé na velikost písmen.

```
>>> an_integer = 1
>>> an_integer
1
>>> AN_INTEGER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'AN_INTEGER' is not defined
>>> An_Integer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'An_Integer' is not defined
>>> an_inteGer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'an_inteGer' is not defined
```

A tak dále.

1.10. Spouštění skriptů

V Pythonu je objektem všechno.

V Pythonu je objektem i modul a moduly definují několik užitečných atributů. Při psaní vašich modulů toho můžeme využít k jejich snadnému testování. Vložíme do nich speciální blok kódu, který se provede v případě, kdy pythonovský soubor spustíte z příkazového řádku. Podívejte se na poslední řádky v souboru `humansize.py`:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

> Python — stejně jako jazyk `c` — používá == pro porovnání a = pro přiřazení. Na rozdíl od jazyka `C` ale Python nepodporuje přiřazovací výraz, takže odpadá možnost nechtěného přiřazení hodnoty v situaci, kdy jste měli na mysli test na rovnost.

Takže čím je vlastně tento příkaz `if` zvláštní? Tak tedy, moduly jsou objekty a všechny moduly mají zabudovaný atribut `__name__`. Jeho hodnota závisí na tom, jakým způsobem modul používáte. Pokud provádíte import modulu, pak je v atributu `__name__` zachyceno jméno jeho souboru bez cesty do adresáře a bez přípony.

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

Ale modul můžete spustit také přímo, jako samostatný program. V takovém případě bude `__name__` nabývat speciální přednastavené hodnoty `__main__`. Python tuto skutečnost otestuje příkazem `if`, zjistí, že výraz platí, a provede blok kódu uvnitř `if`. V našem případě se vytisknou dvě hodnoty.

```
c:\home\diveintopython3> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

A tohle všechno dělá váš první pythonovský program!

1.11. Přečtěte si

- **PEP 257: Docstring Conventions.** Najdete zde vysvětlení, čím se liší dobrý `docstring` od vynikajícího `docstringu`.
  (http://www.python.org/dev/peps/pep-0257/)
- **Python Tutorial: Documentation Strings** — dotýká se stejného tématu.
  (http://docs.python.org/py3k/tutorial/controlflow.html)
- **PEP 8: Style Guide for Python Code** pojednává o vhodných způsobech odsazování.
  (http://www.python.org/dev/peps/pep-0008/)
- **Python Reference Manual** vysvětluje co to znamená, když se řekne, že vše v Pythonu je objekt, protože někteří lidé jsou puntičkáři a rádi o takových věcech dlouze diskutují.
  (http://docs.python.org/py3k/reference/)