České vysoké učení technické v Praze FIT

Programování v Pythonu

Jiří Znamenáček

Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.

Praha & EU: Investujeme do vaší budoucnosti

Python – Moduly

Úvod

S moduly jsme se nejviditelněji už několikrát setkali v souvislosti s příkazem `import`:

```
import math
import sys.argv
from pprint import pprint
```

Tento slouží k zavedení jakýchsi „doplňkových“, do té doby nedostupných částí jazyka (knihoven, konstant, funkcí…), do aktuálního pracovního paměťového prostoru. Co však moduly konkrétně jsou?

I. Ve skutečnosti je modulem každý soubor **.py*. V jeho rámci je společná globální úroveň viditelnosti pro proměnné, funkce apod. v něm definované.

II. Skutečná struktura modulu však může (a samozřejmě často bývá) mnohem složitější – obsahuje nejen více souborů, ale většinou jsou tyto soubory i uspořádány do hierarchické struktury vhodně pojmenovaných adresářů.

Modul v nejjednodušším případě importujeme *pod jeho jménem*. Pak můžeme používat v něm definované objekty (funkce, proměnné atd.). Přitom obsahuje-li modul proveditelné příkazy, ty se **provedou při jeho prvním importu**. Typicky tak slouží třeba k jeho inicializaci a podobně.

> → PS: Naimportované moduly můžete poslat zpátky do křemíkového nebe pomocí příkazu `del MODUL`, ale kromě vyčištění globálního prostoru jmen (*namespace*) nic nezískáte – v rámci stejného běhu interpretru při příštím importu totiž (z důvodů efektivity) „obživne“ předchozí použitý modul, nikoli jeho případná novější verze. Skutečné znovunačtení modulu řeší metoda `imp.reload(MODUL)` z balíčku *imp*.

Import

Moduly a objekty z nich můžete naimportovat a používat několika různými způsoby:

- `import MODUL` – zpřístupní objekty modulu MODUL pomocí „tečkové“ notace (tj. například *math.sin()*, *math.pi* apod.)

  → Toto je nejklasičtější, nejukecanější, ale zároveň i nejbezpečnější způsob importu modulů. Všechny vlastnosti importovaného modulu jsou totiž dostupné pouze pod jeho jménem, tedy v konkrétním prostoru jmen (*namespace*), takže se do ničeho nikde jinde „nemotají“.

- `from MODUL import OBJEKT` – OBJEKT bude dostupný v aktuálním kontextu jako globální, tj. přímo pod svým názvem

  → Na první pohled vypadá tento import nevinně. Ovšem už na ten druhý vám dojde, že pokud ve vašem globálním kontextu existoval jiný objekt pod stejným názvem, nebude už nadále po provedení importu dostupný – jméno OBJEKT bude nyní odkazovat na nově zpřístupněný objekt z modulu MODUL.

- `from MODUL import OBJEKT as NÁZEV` – OBJEKT bude dostupný v aktuálním kontextu jako globální, ale pod názvem NÁZEV

  → Nic vám samozřejmě nebrání vyrobit si později vlastní „zkratku“ typu *msin* = *math.sin*. Ale provedení téhož přímo v rámci importu je řekl bych „čistější“, čitelnější. Poznámka o skrývání objektů stejných jmen z předchozího bodu zde samozřejmě platí také.

sys.path

otázka: Kde Python hledá moduly?
odpověď: Na místech vyjmenovaných v proměnné `sys.path`.

Sama proměnná *sys.path* se skládá z místa umístění aktuálního spouštěného skriptu (tj. adresáře „tečka“), dále obsahu proměnné prostředí *PYTHONPATH* a základní cesty závislé na instalaci (typicky adresář *Lib/site-packages* instalace Python'u):

```
>>> import sys
>>> sys.path
['',
 'C:\\Program Files\\DreamPie\\share\\dreampie',
 'c:\\home\\DJANGO\\Django-1.0.2',
 'c:\\home\\DJANGO\\django-evolution',
 'C:\\WINDOWS\\system32\\python31.zip',
 'c:\\Program Files\\Python31\\DLLs',
 'c:\\Program Files\\Python31\\lib',
 'c:\\Program Files\\Python31\\lib\\plat-win',
 'c:\\Program Files\\Python31',
 'c:\\Program Files\\Python31\\lib\\site-packages',
 'c:\\Program Files\\Python31\\lib\\site-packages\\win32',
 'c:\\Program Files\\Python31\\lib\\site-packages\\win32\\lib',
 'c:\\Program Files\\Python31\\lib\\site-packages\\Pythonwin',
 'c:\\Program Files\\Python31\\lib\\site-packages\\setuptools-0.6c
```

..přičemž při dotazu jsou prohledávána všechna vyjmenovaná umístění popořadě. Za zmínku stojí zvláště dvě věci:

- Aktuální adresář je na prvním místě, z čehož vyplývá, že moduly vhodných jmen v aktuálním adresáři mohou zcela „zakrýt“ moduly vestavěné.
- Je to obyčejný seznam, a proto si ho můžete upravit přidáním vlastních cest na vhodná místa.

  → Pro *neglobální* ovlivnění prohledávané cesty vyjmenujte (existující) adresáře do **.pth* souboru. Umístění jeho nadřazeného adresáře určuje proměnná `site.USER_SITE`.

__name__

Jelikož vlastně každý pythonovský zdrojový soubor může sloužit zároveň jako modul i jako skript, potřebujeme mezi těmito použitími rozlišit. K tomu slouží globální proměnná aktuálního modulu/skriptu `__name__`, která obsahuje:

- řetězec `__main__` – v případě skriptu; tj. pokud jsme soubor **.py* spustili jako „hlavní“, např. z příkazové řádky
- *jméno souboru* (bez přípony) – v případě modulu; tj. pokud jsme soubor **.py* naimportovali pomocí příkazu `import SOUBOR`

Chceme-li tedy, aby se modul při použití jako skript (tj. tehdy, když není *import*ován) nějak rozumně choval, stačí na jeho konec přidat kód:

```
if __name__ == "__main__":
    # kód pro užití modulu jako skriptu
```

- → Typicky se tak výkonnový kód modulu uzavře do několika funkcí, z nichž se ta „hlavní“ v tomto bloku zavolá.
- → Nezapomeňte, že veškerý kód, který se může vykonat, se při prvním importu modulu také vykoná. Jinými slovy – vše, co není uzavřeno ve funkcích nebo podobně, bude při importu provedeno (pokud tedy samozřejmě ty funkce nebudete rovnou i volat, že ;).

dir()

I. Pro ukázání „obsahu“ modulu, tj. objektů dostupných v rámci příslušného modulu, slouží globální funkce `dir()`:

- `dir(MODUL)` – vrátí seřazený seznam řetězců představujících jména objektů dostupných v modulu MODUL
- `dir()` – vypíše jména aktuálně definovaných objektů (tj. například v rámci daného sezení)

Příklad čerstvě spuštěného pythoního shellu:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']

>>> xs = [1, 2, 3,]
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'xs']

>>> import sys
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys', 'xs'

>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__pa
```

II. Z tohoto seznamu získáme pouze jméno objektu (jako řetězec). Objekt samotný pak dostaneme pomocí globální funkce *getattr()* takto: `objekt = getattr(MODUL, JMÉNO_JEHO_OBJEKTU)`

Příklad navazující na předchozí:

```
>>> argvs = getattr(sys, 'argv')

>>> type(argvs)
<class 'list'>

>>> argvs
['']
```

*.pyc, *.pyo

I. Úspěšně naimportované moduly jsou Python'em bajtově zkompilovány, a to buď do souborů `*.pyc` (standardně) nebo `*.pyo` (byl-li interpretr spuštěn s parametrem pro optimalizaci). Jejich výhodou je, že se rychleji načítají => při příštím použití budou dříve k dispozici.

II. Python pozná, může-li použít některý ze zkompilovaných souborů nebo byl-li jejich zdrojový `*.py` soubor mezitím modifikován.

> → Tyto zkompilované soubory jsou (kupodivu) platformně nezávislé. Proto, když na to přijde, můžete váš program distribuovat přímo ve formě těchto *byte-compiled* souborů, tj. bez zdrojáků – o dost hůře se čtou :)

III. **.pyo* vzniknou vyvoláním interpretru s přepínačem `-O`. V dané době to znamená pouze jediné – z kódu jsou odstraněny všechny příkazy `assert`.

Poznámky

Python má zabudovanou ochranu proti cyklickým importům. Pokusíte-li se například naimportovat „sami sebe“..

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# soubor: test.py
import test
print("Ahoj!")
```

..vypíše se nápis *Ahoj!* pouze dvakrát.

> → Dejte si pozor, abyste svým pojmenováním modulu neznemožnili jeho funkčnost na jiné platformě. A stejně tak, abyste „nezastínili“ nějaký globální pythonovský modul! Pokud to ovšem není zrovna váš úmysl – moduly z aktuálního adresáře se totiž importují jako první.

„Balíčky“ (packages)

Definovat všechny potřebné objekty v rámci jednoho modulu je při čemkoliv jen trošku větším velmi nešikovné. Python má pro tento případ k dispozici velmi jednoduchý a elegantní „balíčkovací“ mechanizmus:

> **Rozdělte objekty do jednotlivých modulů a ty do jednotlivých podadresářů hlavního adresáře (tj. balíčku, *package*) podle jejich zaměření a hierarchie.**

Jednotlivé moduly a jejich objekty jsou pak dostupné pomocí klasické tečkové notace, jak ukazuje následující příklad:

→ Zjednodušeno z příkladu v http://www.python.org/doc/3.0.1/tutorial/modules.html

```
sound/                  # top-level package
    __init__.py         # inicializace balíčku sound
    formats/            # subpackage
        __init__.py
        wavread.py
        wavwrite.py
        ...
    effects/            # subpackage
        __init__.py
        echo.py
        reverse.py
        ...
    filters/            # subpackage
        __init__.py
        equalizer.py
        ...
```

I. Moduly z balíčku se pak importují naprosto standardně a průhledně (a absolutně), například..

```
import sound.effects.echo
```

..a podobně i pomocí všech ostatních dříve uvedených způsobů importu.

II. Na posledním místě importu tedy může být buď modul nebo nějaký jeho objekt (funkce, proměnná...).

→ Při importu je nejdříve otestováno, zda je požadovaná „cesta“ prvkem příslušného modulu a pokud ano, je tento prvek načten. Není-li prvkem, předpokládá se, že je to modul, a systém se pokusí o jeho načtení. Neprojde-li to, je vyhozena výjimka *ImportError*.

III. Podobným způsobem na sebe mohou odkazovat moduly uvnitř balíčku. Jde to buď absolutně (tj. od kořene balíčku)..

```
# Nechť se právě nacházíme v modulu sound.filters.equalizer:
from sound.effects import echo
```

..nebo relativně (tj. vůči aktuální pozici):

```
# Nechť se právě nacházíme v modulu sound.effects.reverse:
from . import echo
from .. import formats
from ..filters import equalizer
```

→ Relativní cesty začínají tečkou (počet teček značí počet přeskoků na vyšší úroveň) a je pro ně přípustná pouze forma `from .MODULE import NĚCO`.

__init__.py

Soubor `__init__.py` je **vyžadován**, aby byl adresář rozpoznán jako balíček! Je to mimo jiné z toho důvodu, aby se náhodným importem čehokoliv nezastínily například systémové balíčky Python'u (obecně balíčky na pozdějších místech *sys.path*).

Ve většině případů tedy stačí, aby soubor *__init__.py* byl prázdný. Jelikož je ale čten (a vykonán) při importu balíčku (tedy právě jednou), může též s výhodou obsahovat například jeho inicializační kód (je-li nějaký potřeba).

Jednoduchý balíček tak bude vypadat přibližně takto:

```
balíček/                # top-level package
    __init__.py         # inicializace balíčku "balíček"
    soubor1.py          # modul "balíček.soubor1"
    soubor2.py          # modul "balíček.soubor2"
    ...
```

__main__.py

Jako přítomnost souboru `__init__.py` zajišťuje, že je balíček vůbec pokládán za balíček, tak přítomnost souboru `__main__.py` zajišťuje, že je balíček pokládán za *spustitelný* z příkazové řádky pod svým jménem.

I. Mějme kupříkladu balíček *foo* s následující strukturou:

```
foo/
    __init__.py   # 'foo' je možnou naimportovat
    __main__.py   # 'foo' je možno spustit
    bar.py
```

Přitom soubor *__main__.py* obsahuje..

```
import bar
bar.fce('MAIN')
```

..a soubor *bar.py* zase:

```
def fce( txt="SVĚTE" ):
    s = "Ahoj, " + txt + "!"
    print(s)

if __name__ == "__main__":
    import sys
    try:
        arg = sys.argv[1]
    except:
        arg = "KDOSI"
    fce( arg )
```

Budeme-li stát na úrovni adresáře (a tedy balíčku) *foo*, příslušná volání budou mít následující efekty:

```
# python3 foo
Ahoj, MAIN!

# python foo/bar.py
Ahoj, KDOSI!

# python foo/bar.py Pavle
Ahoj, Pavle!
```

II. Jde to dokonce ještě dál – z adresáře *foo* můžete vyrobit zip-archív (resp. ze souborů v něm obsažených; adresář bude efektivně nahrazen archívem)..

```
foo.zip/
    __main__.py
    bar.py
```

a chovat se k němu velmi podobně:

```
# python foo.zip
Ahoj, MAIN!
```

→ Dokonce není ani potřeba, aby se archív jmenoval zrovna *foo.zip*. Narazil jsem na doporučení (zřejmě po vzoru **.pyc* a **.pyo*) používat koncovku *pyz*, tedy v našem případě

foo.pyz.

Poznámky

Vyhledávání metod (v rámci balíčku) je vyhodnocováno za běhu. Proto bude (o něco málo) rychlejší, ale hlavně asi také čitelnější, místo..

```
modul1.podmodul11.funkce1()
modul1.podmodul11.funkce2()
```

..použít..

```
pracovni_modul = modul1.podmodul11
pracovni_modul.funkce1()
pracovni_modul.funkce2()
```

Tím spíš, bude-li podobný kód uvnitř smyčky.