Jmenné prostory a moduly v Pythonu

Petr Pošík

Katedra kybernetiky, FEL ČVUT v Praze

OI, B4B33RPH: řešení problémů a hry, 2016

Jmenné prostory

Co je to jmenný prostor?

- Podle Wikipedie (https://en.wikipedia.org/wiki/Namespace): Množina symbolů použitých k organizaci objektů různých typů tak, abychom se na ně mohli odkazovat jménem.
- Často má hierarchickou strukturu.
- Každé jméno musí být ve svém jmenném prostoru unikátní.
- Obecný koncept používaný nejen v ICT.
- Příklady:
  - Úplná poštovní adresa by měla jednoznačně určit člověka: jméno, příjmení, číslo domu, ulice, město, země.
  - Souborový systém jednoznačně přiřazuje jména souborům: `\users\bride\documents\private\wedding\application_form.doc`
  - URI přiřazují jednoznačná jména zdrojům dostupným (nejen) na internetu: `http:\\internal.fel.cvut.cz`

Proč používáme jmenné prostory?

- Abychom mohli používat jména jaká chceme, tj.
- abychom mohli použít jména, která před námi použil už někdo jiný,
  - funkce `open()` může být definována v mnoha různých jmenných prostorech jako např. `file`, `urllib`, `furniture.Suitcase`, a
- abychom předešli jmenným kolizím:
  - můžeme použít různé funkce se stejným jménem pomocí **plně kvalifikovaného jména**, např.

```
# Just an illustration, the code will not work
f = file.open('/some/file')
url = urllib.open('http://example.com')
sc = furniture.Suitcase('in bedroom').open()
```

Jmenné prostory v Pythonu

V Pythonu má vlastní jmenný prostor každý

- balík (kolekce souvisejících modulů),
- modul (soubor s koncovkou .py obsahující definice funkcí, tříd a proměnných), a
- třída.

Moduly

Opakované použití téhož kódu

- Jen zřídka píšete program sami, od začátku.
- Mnohem častější a produktivnější je využít již existující kód, který v minulosti už někdo (třeba vy sám) napsal.
- (Pro výukové účely po vás v tomto kurzu chceme vaši vlastní práci: můžete využívat standardní a vlastní moduly.)
- **Modul**:
  - Soubor typu .py.
  - Sada definic symbolů (funkcí, proměnných, tříd, ...) seskupených v jediném souboru, připravených k použití v jiných modulech.
- Modul obvykle obsahuje **vzájemně související** symboly:
  - např. modul `math` obsahuje definice matematických constant (`pi`, `e`) a funkcí (`sin`, `cos`, nebo `sqrt`).
- Ne všechny moduly musejí mít asociovaný .py soubor: některé moduly jsou vestavěny přímo do Pythonu (např. modul `sys`), některé mohou být implementované v jiných jazycích (např. v C). Jejich použití je ale stejné.

Dva typy modulů: programy a knihovny

- Všechno to jsou soubory .py, zde rozdíl není.
- Programy (skripty) jsou určeny primárně ke spouštění (představují vstupní bod nějaké aplikace).
- Knihovny jsou určeny k importu do jiných knihoven a programů
- Často ale můžeme tentýž modul využít
  - i jako skript (můžeme ho spustit a on udělá něco užitečného)
  - i jako knihovnu (můžeme ho importovat do jiných modulů).

Jak importovat moduly: příkaz `import`

- Přístup k symbolům definovaným v jiném modulu (tj. v jiném jmenném prostoru) získáme jejich `import`ováním.
- Příkaz `import` "nahraje" modul do paměti, vytvoří pro něj jméno v aktuálním jmenném protoru a sváže jej s tímto jménem.

In [1]:

```
import sys  # Import a single module
import math, collections  # Import more than one module (not PEP8 compliant)
import os.path as pth  # Import submodule and give it a name
```

Použití symbolů (funkcí, proměnných, ...) definovaných v modulu

Import modulu vytvoří proměnnou shodnou se jménem modulu.

In [2]:

```
import math
```

Tato proměnná odkazuje na objekt typu `module`.

In [3]:

```
type(math)
```

Out[3]:

```
module
```

Symboly definované v modulu lze použít pomocí **plně kvalifikovaných jmen**:

In [4]:

```
math.e, math.log(math.e)
```

Out[4]:

```
(2.718281828459045, 1.0)
```

In [5]:

```
math.pi, math.cos(math.pi), math.sin(math.pi)
```

Out[5]:

```
(3.141592653589793, -1.0, 1.2246467991473532e-16)
```

Import modulů: `from ... import ...`

Níže uvedené příkazy nahrají modul do paměti, ale vytvoří jména v aktuálním jmenném prostoru jen pro vybrané prvky modulu!

In [6]:

```
from sys import path  # Where does Python look for modules?
from sys import modules as loaded_modules  # Which modules are already loaded?
from math import pi, e  # Import math constants
from math import (sin, cos, floor,
    sqrt, log)  # Import split to several lines
from math import *   # Import all non-private names defined in module math. Use wisely.
```

Příkaz `import`: příklady

Jakými různými způsoby mohu importovat funkci `dirname()` z modulu `os.path`?

Bezpečný import celého modulu os. Musíte používat plně kvalifikovaná jména.

In [7]:

```
import os
print(os.path.dirname('/courses/B4B33RPH/lecture.ipynb'))
```

```
/courses/B4B33RPH
```

Import submodulu a použití jiného (třeba kratšího) jména. Nebezpečí kolize s `pth`.

In [8]:

```
import os.path as pth
print(pth.dirname('/courses/B4B33RPH/lecture.ipynb'))
```

```
/courses/B4B33RPH
```

Přímý import submodulu. Nebezpečí kolize s `path`.

In [9]:

```
from os import path
print(path.dirname('/courses/B4B33RPH/lecture.ipynb'))
```

```
/courses/B4B33RPH
```

Import specifického symbolu ze submodulu. Nebezpečí kolize s `dirname`.

In [10]:

```
from os.path import dirname
print(dirname('/courses/B4B33RPH/lecture.ipynb'))
```

```
/courses/B4B33RPH
```

Import všech veřejných symbolů ze submodulu. **Nebezpečí mnoha jmenných kolizí! Nedoporučuje se!**

In [11]:

```
from os.path import *
print(dirname('/courses/B4B33RPH/lecture.ipynb'))
```

```
/courses/B4B33RPH
```

Jak Python ví, kde má moduly hledat?

- Seznam `path` v modulu `sys` obsahuje všechna místa (adresáře), kde Python moduly hledá. Seznam se dá měnit.
- První položkou v seznamu je adresář, v němž je uložen hlavní program, který Python právě vykonává.

In [12]:

```
import sys
print(sys.path)
```

```
['', 'C:\\Program Files\\Mosek\\7\\tools\\platform\\win64x86\\python\\2',
 'C:\\Miniconda3\\envs\\py35\\python35.zip', 'C:\\Miniconda3\\envs\\py35
\\DLLs', 'C:\\Miniconda3\\envs\\py35\\lib', 'C:\\Miniconda3\\envs\\py35',
 'C:\\Miniconda3\\envs\\py35\\lib\\site-packages', 'c:\\p\\sr4rl\\xffx',
 'c:\\p\\code\\python\\pyprshell', 'C:\\Miniconda3\\envs\\py35\\lib\\site-
packages\\setuptools-27.2.0-py3.5.egg', 'C:\\Miniconda3\\envs\\py35\\lib
\\site-packages\\IPython\\extensions', 'c:\\users\\petr\\.ipython']
```

Co se děje, když Python importuje modul?

- Python hledá daný modul mezi těmi, které už jsou importované (seznam `sys.modules`).
- Pokud modul nenajde v `sys.modules`, Python začne modul hledat na místech v `sys.path`, a jakmile jej najde, Python tento **modul spustí (provede)**, a zaznamená, že tento modul už byl importován, do `sys.modules`.
- Python vytvoří pro modul jméno v aktuálním lokálním jmenném prostoru (a/nebo pro všechny proměnné funkce, atd., importované z modulu).

In [13]:

```
>>> import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In [14]:

```
>>> import this     # Nothing happens for the second time
```

Kam umísťovat `importy`

- Z hlediska Pythonu může být příkaz `import` v modulu kdekoliv.
- Zažitá (a doporučená) praxe (podle PEP8):
  - Umístěte všechny importy na začátek modulu (hned za dokumentaci modulu).
  - Importujte nejprve standardní moduly, pak moduly třetích stran, nakonec své vlastní moduly.

Vlastní moduly

Jak vytvořit váš vlastní modul?

Modul je jen soubor typu .py, prostě jej vytvořte!

In [15]:

```
%%writefile my_great_module.py
"""
This is a great module doing a great thing.

And this is a detailed documentation how to use it.
"""
# You probably do not want to print anything like this in a module
# Here we use it just to show when the module was executed
print('This is a message from the module')

def greet(person):
    """Print a greeting of a person"""
    print('Hello,', person)
```

```
Overwriting my_great_module.py
```

Jak použít váš vlastní modul?

Prostě ho importujte! (Když modul importujeme podruhé, znovu se nepustí.)

In [16]:

```
print('1')
from my_great_module import greet  # Import specific symbol from module
print('2')
import my_great_module  # Import the module as a whole (not executed again)
print('3')
```

```
1
This is a message from the module
2
3
```

A nyní můžeme použít funkci definovanou v modulu.

In [17]:

```
greet('John')
```

```
Hello, John
```

Je libo dokumentaci k modulu?

In [18]:

```
help(my_great_module)
```

```
Help on module my_great_module:

NAME
    my_great_module - This is a great module doing a great thing.

DESCRIPTION
    And this is a detailed documentation how to use it.

FUNCTIONS
    greet(person)
        Print a greeting of a person

FILE
    c:\p\0teaching\a4b99rph-new\repos\rph-lectures\modules\my_great_modul
e.py
```

Nebo dokumentaci k funkci z modulu?

In [19]:

```
help(greet)
```

```
Help on function greet in module my_great_module:

greet(person)
    Print a greeting of a person
```

Moduly určené ke spouštění i importu

Ve jmenném prostoru každého modulu je definována zvláštní proměnná `__name__`, v níž je uložen buď

- název modulu (souboru .py), pokud je modul importován, nebo
- řetězec "`__main__`", pokud je modul spuštěn jako hlavní program.

In [20]:

```
%%writefile module_name_test.py
print("The value of variable __name__ is", __name__)
```

```
Overwriting module_name_test.py
```

Když tento modul importujeme:

In [21]:

```
import module_name_test
```

```
The value of variable __name__ is module_name_test
```

Když tento modul spustíme:

In [22]:

```
# To run the module from command-line, you need to call something like
# $ python module_name_test.py
# The following line is the Jupyter/IPython way of running a script inside the code cel
l.
%run module_name_test.py
```

```
The value of variable __name__ is __main__
```

Proměnná __name__ je definována v obou jmenných prostorech, jak zde, odkud vše voláme, tak uvnitř modulu:

In [23]:

```
print('The value of __name__ is', __name__)
print('The value of module_name_test.__name__ is', module_name_test.__name__)
```

```
The value of __name__ is __main__
The value of module_name_test.__name__ is module_name_test
```

Toho se často využívá k vytvoření "spustitelných" a zároveň "importovatelných" modulů:

In [24]:

```
%%writefile example_module.py
def add1(number):
    return number + 1

# All the above code is executed no matter if the module is imported or executed.
# It usually contains only definitions of functions, classes, variables, etc.

if __name__ == "__main__":

    # All the code below is executed only when the file is run as a script.
    # It can be used e.g. to test the defined functions, etc.
    print(add1(10))
```

```
Overwriting example_module.py
```

Zkusme modul importovat.

In [25]:

```
import example_module
```

Během importu se zdánlivě nic nestalo. Ale není to pravda. Nespustil se blok kódu uvnitř `if`, ale byl nahrán modul `example_module`, který má funkci `add1()` ve svém jmenném prostoru. Funkci můžeme použít:

In [26]:

```
example_module.add1(31)
```

Out[26]:

```
32
```

Ale když modul spustíme, nejen, že se nadefinuje funkce `add1()`, ale také se vykoná blok kódu v podmínce `if` (a vytiskne se výsledek jednoduchého testu).

In [27]:

```
%run example_module.py
```

```
11
```

Moduly: Příklad

V modulu `triangle.py` implementujte funkci `generate(n)`, která vrátí seznam n 2D bodů, ležících uvnitř trojúhelníka s vrcholy (0, 0), (1, 0), and (0.5, 1).

Řekněme, že jsme se rozhodli použít následující algoritmus:

- Mějme 3 body ve 2D prostoru (v rovině).
- Jeden z vrcholů vyber jako aktuální bod.
- Opakuj n-krát:
    - Vygeneruj nový bod jako průměr aktuálního bodu a náhodně zvoleného vrcholu.

In [28]:

```
%%writefile triangle.py
import random

vertices = ((0, 0),
            (1, 0),
            (0.5, 1))

def generate(n):
    """Generate n data points in triangle."""
    points = []
    p = random.choice(vertices)
    for i in range(n):
        v = random.choice(vertices)
        p = tuple((a + b) / 2 for a, b in zip(p, v))
        points.append(p)
    return points
```

```
Overwriting triangle.py
```

Nyní použijeme náš modul `triangle` a knihovnu `matplotlib` ve skriptu, který vytvoří graf vygenerovaných bodů.

In [29]:

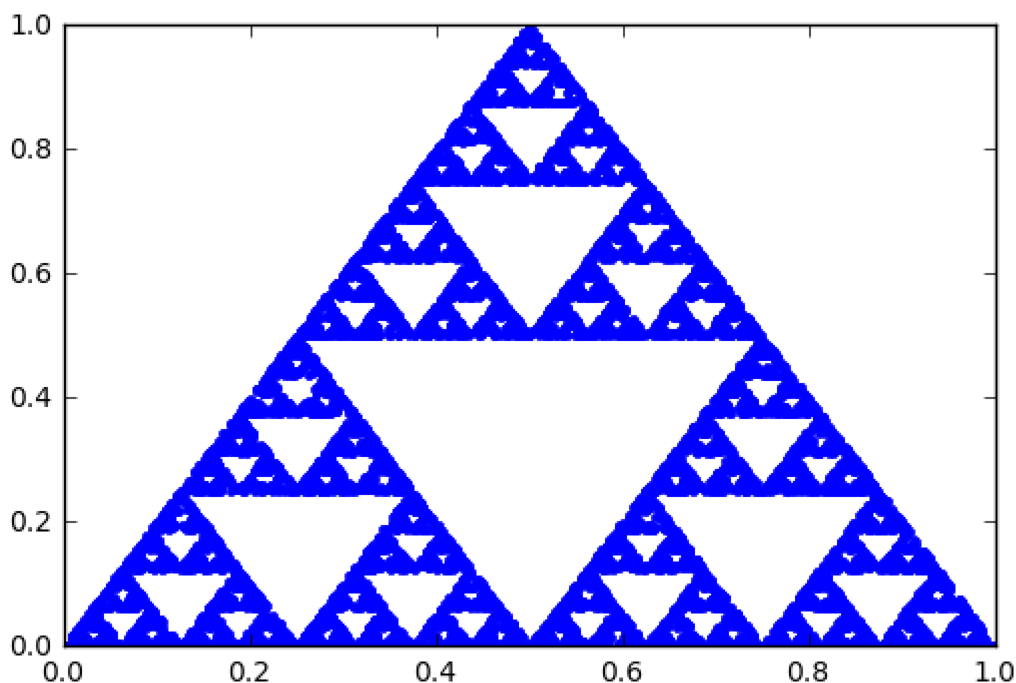
```
%%writefile plot_triangle.py
import matplotlib.pyplot as plt
from triangle import generate

# Generate n points from the triangle
pts = generate(10000)
# Separate their x and y coordinates
x = [x for x, y in pts]
y = [y for x, y in pts]
# Plot the points
plt.plot(x, y, '.')
plt.show()
```

```
Overwriting plot_triangle.py
```

Pokud máme nainstalovanou knihovnu `matplotlib`, spuštěním skriptu bychom měli dostat následující graf.

In [30]:

```
%matplotlib inline
%run plot_triangle.py
```



Souhrn

- Moduly umožňují opakované použití kódu (funkcí, tříd, ...):
  - Naprogramuj jednou, použij kolikrát chceš.
  - Symboly definované unvitř modulu lze `import`ovat a použít v mnoha jiných modulech/skriptech.
- Moduly v Pythonu definují jmenné prostory.