Namespaces and Modules in Python

Petr Pošík

Department of Cybernetics, FEE CTU in Prague

EECS, BE5B33PRG: Programming Essentials, 2015

Namespaces

What is a namespace?

- Set of symbols used to organize objects of various kinds so that we can refer to them by name.
- Often hierarchically structured.
- Each name must be unique in its namespace.
- A very general concept, related not only to computers and computing.
- Examples:
  - Complete postal address assigns names to persons: Name, Surname, Street, House number, City, Country
  - Email addresses assign names to post-boxes (usually belonging to a single person): `name.surname@fel.cvut.cz`
  - File systems assign names to files: `\users\bride\documents\private\wedding\application_form.doc`
  - Uniform resource identifier on the web: `http:\\internal.fel.cvut.cz`

Why namespaces

- Purpose:
  - to allow for name reuse:
    - function `open()` may be defined in namespaces like `file`, `urllib`, `furniture.Suitcase`
  - to prevent name clashes:
    - we can use different functions with the same name using **fully qualified names**, e.g.

```
# 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()
```

Namespaces in Python

In Python, namespaces are defined by

- packages (collections of related modules)
- modules (.py files containing definitions of functions, classes, variables, etc.)
- classes

Modules

Code reuse

- It is rare to write a program alone, from scratch. Much more common and productive: to use some of those millions of lines written by other authors (or you) before.
- **Module**:
  - A single .py file.
  - A collection of symbol definitions (functions, variables, classes, ...) grouped together in a single file, ready to be used by other modules.
- The module contents are usually related in some way.
  - E.g. module `math` contains definitions of variable (constant) `pi` and functions `sin()`, `cos()`, or `sqrt()`.
- Not all modules have associated .py file: some are built into Python (e.g. `sys` module), some are implemented in other languages (most often in C). Their usage is the same.

Difference between programs and modules

- Both are stored in .py files.
- Programs (scripts) are designed to be run (executed).
- Modules (libraries) are designed to be imported and used by other programs and other modules.
- Sometimes, a .py file is designed to be both a program and a module. It can be executed (and do a useful thing) and also imported (and provide functionality for other modules).

Importing modules: `import` statement

- To gain access to symbols defined in a module (i.e. in a different namespace), you have to `import` it.
- The statements below load a module, create a name in the current namespace, and bind the name to the loaded module.

In [1]:
```
import sys  # Import a single module
import math, collections  # Import more than one module
import os.path as pth  # Import submodule and give it a name
```

Using symbols (functions, variables, ...) defined in a module

Module import creates a variable with the name of the module.

In [2]:
```
import math
```

The variable refers to an object whose type is `module`.

In [3]:
```
type(math)
```

Out[3]:
```
module
```

Using the symbols defined in the imported module via **fully qualified names**:

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)
```

Importing modules: `from ... import ...`

The statements below load a module, but create and **bind a name in the current namespace** only for specific items in that module.

```
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.
```

Examples of `import`

How can I import and use function `dirname()` from module `os.path`?

Safe import of a high-level module. Forces the use of fully-qulified names.

```
In [7]: import os
        print(os.path.dirname('/courses/BE5B33PRG/lecture.ipynb'))
```

```
/courses/BE5B33PRG
```

Import of a submodule with new (shorter) name. Risk of collision with `pth`.

```
In [8]: import os.path as pth
        print(pth.dirname('/courses/BE5B33PRG/lecture.ipynb'))
```

```
/courses/BE5B33PRG
```

Direct import of a submodule. Risk of collision with `path`.

```
In [9]: from os import path
        print(path.dirname('/courses/BE5B33PRG/lecture.ipynb'))
```

```
/courses/BE5B33PRG
```

Import a specific symbol from a submodule. Risk of collision with `dirname`.

```
In [10]: from os.path import dirname
         print(dirname('/courses/BE5B33PRG/lecture.ipynb'))
```

```
/courses/BE5B33PRG
```

Import of all non-private symbols from a submodule. **Risk of many name collisions!**

```
In [11]: from os.path import *
         print(dirname('/courses/BE5B33PRG/lecture.ipynb'))
```

```
/courses/BE5B33PRG
```

Where does Python look for imported modules?

- A list named `path` in module `sys` contains all the locations (and can be modified).
- The first item on the list is the directory containing the program itself.

In [12]:
```
import sys
print(sys.path)
```

```
['', 'c:\\miniconda3\\envs\\py34\\scripts\\src\\traitlets', 'c:\\miniconda3\\envs\\py
34\\scripts\\src\\jupyter-core', 'c:\\miniconda3\\envs\\py34\\scripts\\src\\nbformat'
, 'c:\\miniconda3\\envs\\py34\\scripts\\src\\jupyter-client', 'c:\\miniconda3\\envs\\
py34\\scripts\\src\\ipython', 'c:\\miniconda3\\envs\\py34\\scripts\\src\\ipykernel',
'c:\\miniconda3\\envs\\py34\\scripts\\src\\nbconvert', 'C:\\Program Files\\Mosek\\7\\
tools\\platform\\win64x86\\python\\2', 'C:\\Miniconda3\\envs\\py341\\python34.zip', '
C:\\Miniconda3\\envs\\py341\\DLLs', 'C:\\Miniconda3\\envs\\py341\\lib', 'C:\\Minicond
a3\\envs\\py341', 'c:\\miniconda3\\envs\\py341\\lib\\site-packages\\setuptools-18.4-p
y3.4.egg', 'C:\\Miniconda3\\envs\\py341\\lib\\site-packages', 'C:\\Miniconda3\\envs\\
py341\\lib\\site-packages\\win32', 'C:\\Miniconda3\\envs\\py341\\lib\\site-packages\\
win32\\lib', 'C:\\Miniconda3\\envs\\py341\\lib\\site-packages\\Pythonwin', 'c:\\minic
onda3\\envs\\py34\\scripts\\src\\ipython\\IPython\\extensions', 'c:\\users\\petr\\.ip
ython']
```

What happens when Python imports a module?

- Python searches the module among already imported modules in `sys.modules`.
- If the module is not found in `sys.modules`, Python searches locations in `sys.path`, **executes the module** once it is found, and records that the module has already been imported.
- Python creates names in local namespace for the imported module, or for all the variables, functions, etc. imported from the module.

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
```

Where to place `import` statements

- An `import` statement can be anywhere in a .py file.
- Common practice:
  - Place all imports in the beginning of a .py file, after shebang and after the module documentation.
  - Import standard library modules first, then third-party modules, and finally your own modules.

Custom modules

Creating your own modules

Module is just a .py file, simply create it!

In [15]:
```
%%writefile mygreatmodule.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 mygreatmodule.py
```

Using your own modules

Import them as needed! (When we import the same module again, the module is not executed again.)

In [16]:
```
from mygreatmodule import greet
import mygreatmodule  # Importing the same module for the second time, not executed aga
in
```

```
This is a message from the module
```

And we can use the function from the module.

In [17]:
```
greet('John')
```

```
Hello, John
```

And ask for the documentation of the module:

In [18]:
```
help(mygreatmodule)
```

```
Help on module mygreatmodule:

NAME
    mygreatmodule - 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\programming essentials\prg-notes\mygreatmodule.py
```

Ask for a documentation of the function:

In [19]:
```
help(greet)
```

```
Help on function greet in module mygreatmodule:

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

Modules that can be both run and imported

- Special variable called `__name__`: inside each module it contains
  - the name of the module (of the .py file) when the module is imported, or
  - a string `"__main__"` when the .py file is run as a program (script).

In [20]:
```
%%writefile testmodulename.py
print("The module's __name__ is", __name__)
```

```
Overwriting testmodulename.py
```

Importing this module results in:

In [21]:
```
import testmodulename
```

```
The module's __name__ is testmodulename
```

Running this module results in:

In [22]:
```
# To run the module from command-line, you need to call something like
# > python testname.py
# The following line is the IPython way of running a script inside the code cell.
%run testmodulename.py
```

```
The module's __name__ is __main__
```

Variable `__name__` is defined in both the calling namespace and in the namespace of the module:

```
In [23]: print('__name__ is', __name__)
         print('testmodulename.__name__ is', testmodulename.__name__)
```

```
__name__ is __main__
testmodulename.__name__ is testmodulename
```

This is usually used to create "runable" and "importable" modules like this:

```
In [24]: %%writefile examplemodule.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 definitions of functions, 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 examplemodule.py
```

Let us test it.

```
In [25]: import examplemodule
```

During the import, seemingly nothing happend. Actually, this is not true. The code in the `if`-block was not executed, but the function `add1()` was defined, so we can use it:

```
In [26]: examplemodule.add1(31)
```

```
Out[26]: 32
```

But when we run the module, function `add1()` is defined, the `if` branch is also executed (printing the result of a simple test).

```
In [27]: %run examplemodule.py
```

```
11
```

Example: Modules

In module `triangle.py` implement function `generate(n)` that will produce a list of 2D points lying inside a triangle given by vertices (0, 0), (1, 0), and (0.5, 1). The function shall use the following algorithm:

- Define 3 points in 2D plane.
- Choose one of the points as starting.
- Repeat n-times:
  - Generate new point as the average of the current point and one of the 3 vertices.

In [28]:
```
%%writefile triangle.py
import random

def generate(n):
    """Generate n data points in triangle."""
    vertices = ((0, 0),
                (1, 0),
                (0.5, 1))
    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
```

Now, we will use our module `triangle` and 3rd party library `matplotlib` in a script to plot the generated points.

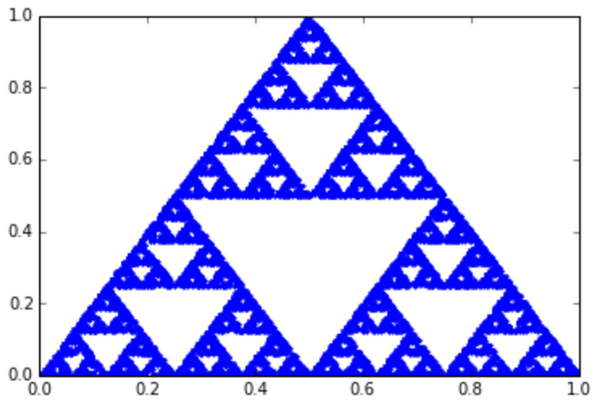
In [29]:
```
%%writefile plot_triangle.py
import matplotlib.pyplot as plt
from triangle import generate

# 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
```

If matplotlib is installed, running the script shall produce the following graph:

In [30]:
```
%matplotlib inline
%run plot_triangle.py
```



Summary

- Modules allow for code reuse:
    - Symbols defined inside a module can be `import`ed and used in many other modules/scripts.
- Modules in Python define namespaces.

Notebook config

Some setup follows. Ignore it.

In [31]:
```
from notebook.services.config import ConfigManager
cm = ConfigManager()
cm.update('livereveal', {
              'theme': 'Simple',
              'transition': 'slide',
              'start_slideshow_at': 'selected',
              'width': 1268,
              'height': 768,
              'minScale': 1.0
})
```

Out[31]:
```
{'height': 768,
 'minScale': 1.0,
 'start_slideshow_at': 'selected',
 'theme': 'Simple',
 'transition': 'slide',
 'width': 1268}
```

In [32]:
```
%%HTML
<style>
.reveal #notebook-container { width: 90% !important; }
.CodeMirror { max-width: 100% !important; }
pre, code, .CodeMirror-code, .reveal pre, .reveal code {
    font-family: "Consolas", "Source Code Pro", "Courier New", Courier, monospace;
}
pre, code, .CodeMirror-code {
    font-size: inherit !important;
}
.reveal .code_cell {
    font-size: 130% !important;
    line-height: 130% !important;
}
</style>
```