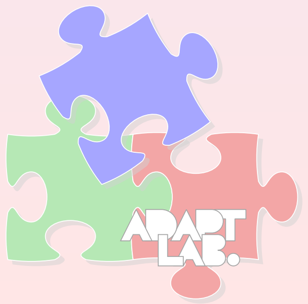
# Modularizing Python
## Modules & Packages

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola



## Modules
### Basics on Modules

A module is a simple text file of Python's statements.

```
import «module name»
```

Lets a client (importer) fetch a module as a whole.

```
from «module name» import «name list»|*
```

Allows clients to fetch particular names from a module.

```
imp.reload «module name»
```

Provides a way to reload a module's code without stopping the Python interpreter.

## Modules
### How Imports Work

Imports are run-time operations that:

1. find the module's file

   ```
   import rectangle
   ```

   This looks for `rectangle.py` through a standard module search path.

2. load module's bytecode (from a `.pyc` file named after the module)
   - if a source file newer than the `.pyc` is found or
   - no bytecode is found the module source file is compiled

   compilation occurs when the module is imported
   - so only imported modules will leave a `.pyc` file on your system.

3. run the module's code to build the objects it defines.

# Modules
## Python's Module Search Path

Python looks for module in:

1. The current directory
2. `PYTHONPATH` directories (if set)
3. Standard library directories
4. The contents of any `.pth` files

The concatenation of these four components becomes `sys.path`.

```
[DING!]cazzola@hymir:~/esercizi-pa>python3
>>> import sys
>>> sys.path
['', '/usr/lib64/python34.zip', '/usr/lib64/python3.4',
 '/usr/lib64/python3.4/plat-linux', '/usr/lib64/python3.4/lib-dynload',
 '/usr/lib64/python3.4/site-packages', '/usr/lib/python3.4/site-packages']
```

## Modules
### Imports Happen Only Once

Modules are imported only once, so, code is executed just at import time.

Let us consider

```
print('hello')
spam = 1          # Initialize variable
```

```
[16:45]cazzola@hymir:~/esercizi-pa>python3
>>> import simple   # First import: loads and runs file's code
hello
>>> simple.spam     # Assignment makes an attribute
1
>>> simple.spam = 2 # Change attribute in module
>>> import simple   # Just fetches already loaded module
>>> simple.spam     # Code wasn't rerun: attribute unchanged
2
```

- the module `simple` is imported just the first time
- the assignment to `spam` in the module is only executed the first time.

## Modules

### import and from Are Assignments

**import** and **from** are statements not compile-time declarations.

- they may be used in statements, in function definition, ...;
- they are not resolved or run until the execution flow reaches them.

**import** and **from** are assignments:

```
x = 1
y = [1, 2]
```

- **import** assigns an entire module object to a single name

```
[23:10]cazzola@hymir:~/esercizi-pa>python3
>>> import small
>>> small
<module ’small’ from ’small.py’>
```

- **from** assigns new names to homonyms objects of another module.

```
>>> from small import x, y
>>> x = 42
>>> y[0] = 42
>>> import small
>>> small.x
1
>>> small.y
[42, 2]
```

## Modules
### "Import" and "From" Equivalence

The following

```
from small import x,y    # Copy these two names out (only)
```

is equivalent to

```
import small             # Fetch the module object
x = small.x              # Copy names out by assignment
y = small.y
del small                # Get rid of the module name
```

```
[9:03]cazzola@hymir:~/esercizi-pa>python3
>>> from small import x,y
>>> small
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'small' is not defined
>>> import small
>>> small
<module 'small' from 'small.py'>
>>> x = small.x
>>> y = small.y
>>> del small
>>> small
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'small' is not defined
>>> x
1
```

# Modules
## Module Namespaces

### Files generate namespaces

- module statements run once at the first import
- every name that is assigned to a value at the top level of a module file becomes one of its attributes.
- module namespaces can be accessed via `__dict__` or `dir(module)`
- module are single scope (i.e., local is global)

```python
print('starting to load...')
import sys
name = 42
def func(): pass
print('done loading')
```

```
[23:37]cazzola@hymir:~/esercizi-pa>python3
>>> import module2
starting to load...
done loading
>>> module2.sys
<module 'sys' (built-in)>
>>> module2.name
42
>>> module2.func
<function func at 0xb7a0cbac>
>>> list(module2.__dict__.keys())
['name','__builtins__','__file__','__package__','sys','func','__name__','__doc__']
```

# Modules
## Module Reload

The `imp.reload` function forces an already loaded module's code to be reloaded and rerun.

– Assignments in the file's new code change the existing module object in-place.

```python
# changer.py
message = "First version"
def printer():
  print(message)
```

```python
# changer.py after the editing
message = "After editing"
def printer():
    print('reloaded:', message)
```

```
>>> import changer
>>> changer.printer()
First version
>>> ^Z
Suspended
[9:57]cazzola@hymir:~/esercizi-pa>gvim changer.py
[9:58]cazzola@hymir:~/esercizi-pa>fg
python3
```

```
>>> import changer
>>> changer.printer()
First version
>>> from imp import reload
>>> reload(changer)
<module 'changer' from 'changer.py'>
>>> changer.printer()
reloaded: After editing
```

## Packages
### Basics on Python's Packages

An **import** can name a directory path.

- A directory of Python code is said to be a package, so such imports are known as package imports.
- A package import turns a directory into another Python namespace, with attributes corresponding to the subdirectories and module files that the directory contains.

Packages are organized in directories, e.g., $\texttt{dir}_0\texttt{/dir}_1\texttt{/mod}_0$

- **import**s are independent of the file system conventions, i.e., **import** $\texttt{dir}_0\texttt{.dir}_1\texttt{.mod}_0$ loads $\texttt{dir}_0\texttt{/dir}_1\texttt{/mod}_0$;
- the package must be reachable via the Python's search path mechanism.

## Packages
### Package __init__.py files

Each directory named within the path of a package import statement must contain a file named `__init__.py`

- They contain standard python code
- They provide a hook for package-initialization-time actions, generate a module namespace for a directory, and support the **from** * when used in combination with package imports.

#### Package Initialization

The first time Python imports through a directory, it automatically runs all the code in the directory's `__init__.py` file.

#### Package Namespace Initialization

In the package import model, the directory paths in your script become real nested object paths after an import.

#### From * Statement Behavior

`__all__` lists in `__init__.py` files can be used to define what is exported when a directory is imported with the **from** * statement form.

## Packages
### Package Example

```
# dir0/__init__.py
print('dir0 init')
x = 1
```

```
# dir0/dir1/__init__.py
print('dir1 init')
y = 2
```

```
# dir0/dir1/mod.py
print('in mod.py')
z = 3
```

```
[11:08]cazzola@hymir:~/esercizi-pa>python3
>>> import dir0.dir1.mod
dir0 init
dir1 init
in mod.py
>>> from imp import reload
>>> reload(dir0)
dir0 init
<module 'dir0' from 'dir0/__init__.py'>
>>> reload(dir0.dir1)
dir1 init
<module 'dir0.dir1' from 'dir0/dir1/__init__.py'>
>>> dir0.dir1
<module 'dir0.dir1' from 'dir0/dir1/__init__.py'>
>>> dir0.dir1.mod
<module 'dir0.dir1.mod' from 'dir0/dir1/mod.py'>
>>> dir0.x,dir0.dir1.y,dir0.dir1.mod.z
(1, 2, 3)
>>> from dir0.dir1.mod import z
>>> z
3
>>> import dir0.dir1.mod as mod
>>> mod.z
3
```

## Packages
### Absolute vs Relative Imports

**imports** in packages have a slightly different behavior

- they are absolute with respect to the Python's search path
- to look for modules in the package you have to use the relative path search statement **from .**

```python
# mypkg/spam.py
from . import eggs
print(eggs.X)
```

```python
# mypkg/eggs.py
X = 99999
import string
print(string)
```

```
[11:33]cazzola@hymir:~/esercizi-pa>python3
>>> import mypkg.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mypkg/spam.py", line 2, in <module>
    import eggs
ImportError: No module named eggs
>>> import mypkg.spam
<module 'string' from '/usr/lib/python3.1/string.py'>
99999
>>>
```

## Advances on Packages & Modules
### Data Hiding & Future Extensions

Data hiding in Python is only a convention

- to prefix a name with a '_' will prevent the **from** * statement to import such a name.
- to assign a list of strings to the `__all__` will force the **from** * statement to import only the listed names.

### Enabling Future Language Features

Python permits a gradual introduction of new concepts in the language

```
from __future__ import featurename
```

This permits to turn on a novel featured disabled by default

- this is particularly useful for backwards compatibility.

## References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
  The Pragmatic Bookshelf, second edition, 2009.
- Mark Pilgrim.
  Dive into Python 3.
  Apress*, 2009.
- Mark Summerfield.
  Programming in Python 3: A Complete Introduction to the Python Language.
  Addison-Wesley, October 2009.