Προγραμματισμός σε C++ & Python & Εφαρμογές στη Ναυπηγική & Ναυτική Μηχανολογία

ΣΝΜΜ 2019

Μάθημα 3Α

Γεώργιος Παπαλάμπρου
Επίκουρος Καθηγητής ΕΜΠ
`george.papalambrou@lme.ntua.gr`

Εργαστήριο Ναυτικής Μηχανολογίας (Κτίριο Λ)
Σχολή Ναυπηγών Μηχανολόγων Μηχανικών
Εθνικό Μετσοβιο Πολυτεχνείο

March 15, 2019

Περιεχόμενα

Περιεχόμενο Μαθήματος

- Εβδομάδα 1. Α. Εισαγωγή. Η γλώσσα. Το περιβάλλον Linux. Command line. Python interpreter. Ιστοσελίδα μαθήματος. Βιβλιογραφία. Editors: Sublime, Spyder
  Β. Εισαγωγή στην γλώσσα Python. Hello World.
- Εβδομάδα 2. Α. Data types. Loops. Control. File I/O
  Β. Παραδείγματα.
- Εβδομάδα 3. Functions. Modules
- Εβδομάδα 4. OOP. Classes
- Εβδομάδα 5. Α. Βιβλιοθήκες NymPy, SciPy. Errors-Exceptions
  Β. Παραδείγματα: Γραμμική άλγεβρα, Γραφικά
- Εβδομάδα 6. Εφαρμογή: Hardware. Πλατφόρμες. Πρωτόκολλα. Βασικό I/O Εφαρμογή: Neural Networks. Machine Learning

Εισαγωγή

Η παράδοση προέρχεται από:

- An Introduction To Python For Scientific Computing, M. Scott Shell
  [Διαθέσιμο στο : http://www.freetechbooks.com/an-introduction-to-python-for-scientific-computing-t885.html]

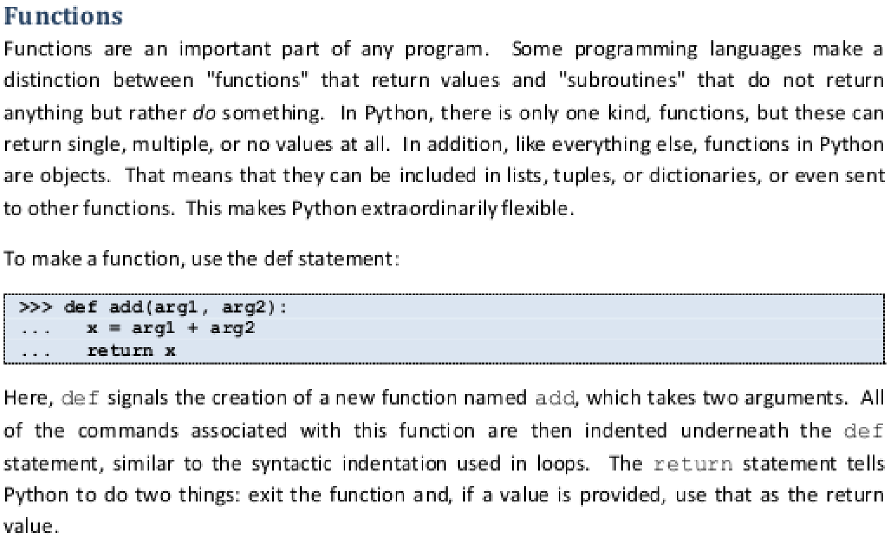
Για σήμερα:

- Functions, p. 38-44
- Modules, p. 44-48

Προυπόθεση (έχετε δει από Core Python Programming, Wesley Chun):

- Chapter 11. Functions
- Chapter 12. Modules

Functions



Functions

Functions are an important part of any program. Some programming languages make a distinction between "functions" that return values and "subroutines" that do not return anything but rather *do* something. In Python, there is only one kind, functions, but these can return single, multiple, or no values at all. In addition, like everything else, functions in Python are objects. That means that they can be included in lists, tuples, or dictionaries, or even sent to other functions. This makes Python extraordinarily flexible.

To make a function, use the def statement:

```
>>> def add(arg1, arg2):
...     x = arg1 + arg2
...     return x
```

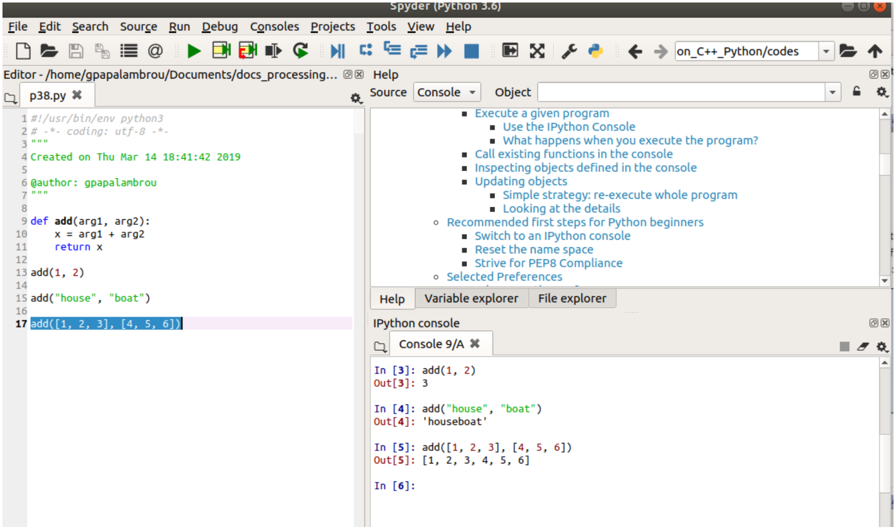
Here, `def` signals the creation of a new function named `add`, which takes two arguments. All of the commands associated with this function are then indented underneath the `def` statement, similar to the syntactic indentation used in loops. The `return` statement tells Python to do two things: exit the function and, if a value is provided, use that as the return value.

Functions

Unlike other programming languages, functions do not need to specify the types of the arguments sent to them. Python evaluates these at runtime every time the function is called. Using the above example, we could apply our function to many different types:

```
>>> add(1, 2)
3
>>> add("house", "boat")
'houseboat'
>>> add([1, 2, 3], [4, 5, 6])
[1, 2, 3, 4, 5, 6]
>>> add(1, "house")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Functions - Περιβάλλον Spyder [p38.py]


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
on_C++_Python/codes
Editor - /home/gpapalambrou/Documents/docs_processing...
p38.py
Help
Source Console Object
Execute a given program
Use the IPython Console
What happens when you execute the program?
Call existing functions in the console
Inspecting objects defined in the console
Updating objects
Simple strategy: re-execute whole program
Looking at the details
Recommended first steps for Python beginners
Switch to an IPython console
Reset the name space
Strive for PEP8 Compliance
Selected Preferences
Help Variable explorer File explorer
IPython console
Console 9/A

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 14 18:41:42 2019

@author: gpapalambrou
"""

def add(arg1, arg2):
    x = arg1 + arg2
    return x

add(1, 2)

add("house", "boat")

add([1, 2, 3], [4, 5, 6])
```

```
In [3]: add(1, 2)
Out[3]: 3

In [4]: add("house", "boat")
Out[4]: 'houseboat'

In [5]: add([1, 2, 3], [4, 5, 6])
Out[5]: [1, 2, 3, 4, 5, 6]

In [6]:
```

Functions

The return statement can be called from anywhere within a function:

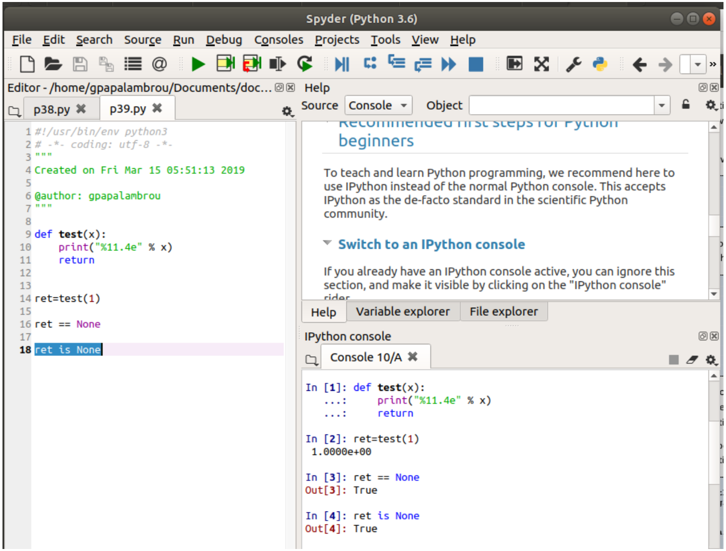
```
>>> def power(x, y):
...   if x <= 0:
...     return 0.
...   else:
...     return x**y
```

If no return statement is present within a function, or if the return statement is used without a return value, Python automatically returns the special value None:

```
>>> def test(x):
...   print "%11.4e" % x
...   return
...  <hit return>
>>> ret = test(1)
1.0000e+000
>>> ret == None
True
>>> ret is None
True
```

None is a reserved, special object in Python, similar to True and False. It essentially means *nothing*, and will not appear using the print statement. However, as seen in the above example, one can test for the None value using conditional equality or the is statement.

Functions - Spyder [p39.py]


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - /home/gpapalambrou/Documents/doc...
p38.py
p39.py
Help
Source Console Object
Help Variable explorer File explorer
IPython console
Console 10/A

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 15 05:51:13 2019

@author: gpapalambrou
"""

def test(x):
    print("%11.4e" % x)
    return


ret=test(1)

ret == None

ret is None
```

Recommended first steps for Python beginners

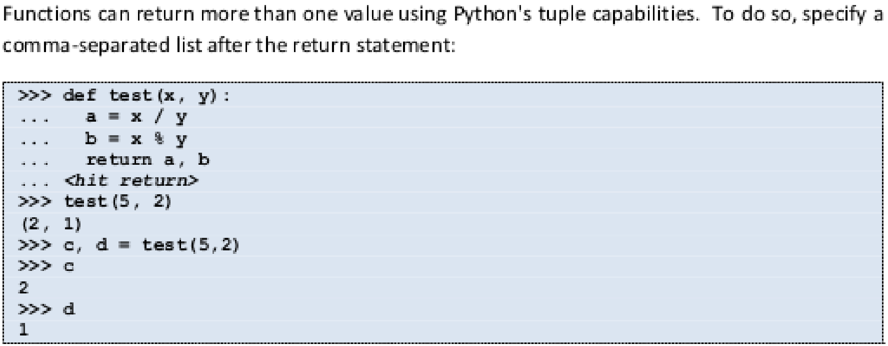
To teach and learn Python programming, we recommend here to use IPython instead of the normal Python console. This accepts IPython as the de-facto standard in the scientific Python community.

Switch to an IPython console

If you already have an IPython console active, you can ignore this section, and make it visible by clicking on the "IPython console" rider

```
In [1]: def test(x):
   ...:     print("%11.4e" % x)
   ...:     return

In [2]: ret=test(1)
 1.0000e+00

In [3]: ret == None
Out[3]: True

In [4]: ret is None
Out[4]: True
```

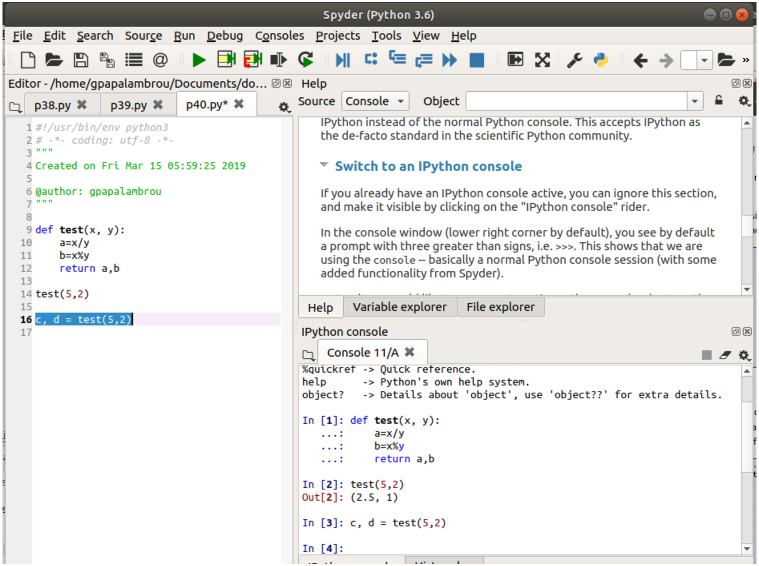
Functions


Functions can return more than one value using Python's tuple capabilities. To do so, specify a comma-separated list after the return statement:

```
>>> def test(x, y):
...     a = x / y
...     b = x % y
...     return a, b
... <hit return>
>>> test(5, 2)
(2, 1)
>>> c, d = test(5,2)
>>> c
2
>>> d
1
```

x/y: integer divisions will truncate instead of becoming a floating point number.
x%y: modulo

Functions - Spyder [p40.py]



Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - /home/gpapalambrou/Documents/do...
p38.py
p39.py
p40.py*
Help
Source Console Object
Help Variable explorer File explorer
IPython console
Console 11/A

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 15 05:59:25 2019

@author: gpapalambrou
"""

def test(x, y):
    a=x/y
    b=x%y
    return a,b

test(5,2)

c, d = test(5,2)

```

IPython instead of the normal Python console. This accepts IPython as the de-facto standard in the scientific Python community.

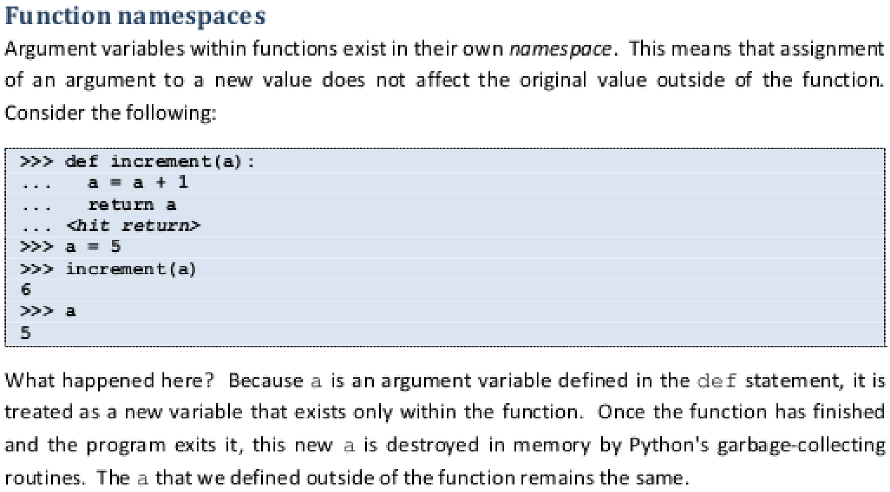
Switch to an IPython console

If you already have an IPython console active, you can ignore this section, and make it visible by clicking on the "IPython console" rider.

In the console window (lower right corner by default), you see by default a prompt with three greater than signs, i.e. >>>. This shows that we are using the `console` -- basically a normal Python console session (with some added functionality from Spyder).

```
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: def test(x, y):
   ...:     a=x/y
   ...:     b=x%y
   ...:     return a,b

In [2]: test(5,2)
Out[2]: (2.5, 1)

In [3]: c, d = test(5,2)

In [4]:
```

Function Namespaces



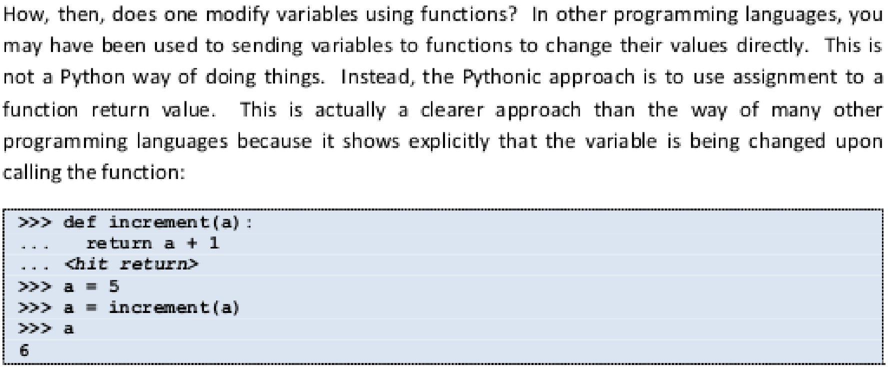
Function namespaces

Argument variables within functions exist in their own *namespace*. This means that assignment of an argument to a new value does not affect the original value outside of the function. Consider the following:

```
>>> def increment(a):
...     a = a + 1
...     return a
... <hit return>
>>> a = 5
>>> increment(a)
6
>>> a
5
```

What happened here? Because `a` is an argument variable defined in the `def` statement, it is treated as a new variable that exists only within the function. Once the function has finished and the program exits it, this new `a` is destroyed in memory by Python's garbage-collecting routines. The `a` that we defined outside of the function remains the same.

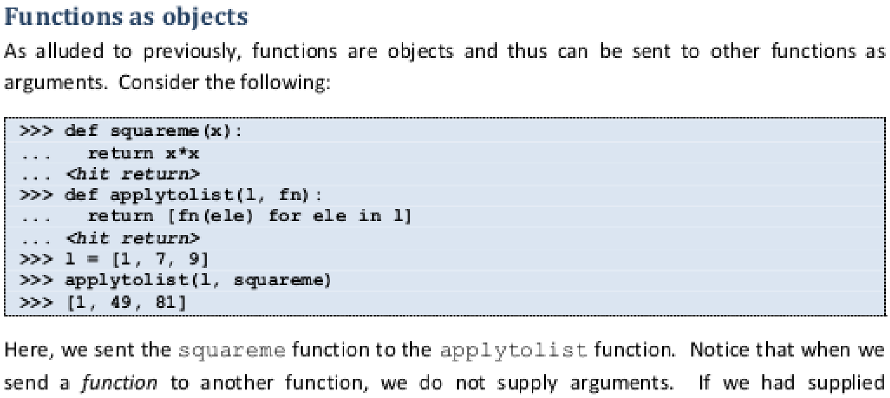
Function Namespaces



How, then, does one modify variables using functions? In other programming languages, you may have been used to sending variables to functions to change their values directly. This is not a Python way of doing things. Instead, the Pythonic approach is to use assignment to a function return value. This is actually a clearer approach than the way of many other programming languages because it shows explicitly that the variable is being changed upon calling the function:

```
>>> def increment(a):
...     return a + 1
... <hit return>
>>> a = 5
>>> a = increment(a)
>>> a
6
```

Functions as objects



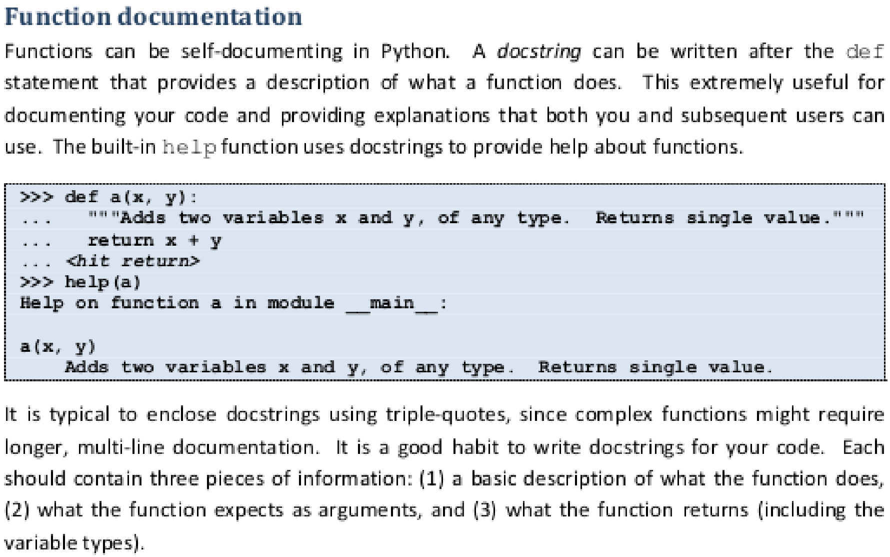
Functions as objects

As alluded to previously, functions are objects and thus can be sent to other functions as arguments. Consider the following:

```
>>> def squareme(x):
...   return x*x
... <hit return>
>>> def applytolist(l, fn):
...   return [fn(ele) for ele in l]
... <hit return>
>>> l = [1, 7, 9]
>>> applytolist(l, squareme)
>>> [1, 49, 81]
```

Here, we sent the `squareme` function to the `applytolist` function. Notice that when we send a *function* to another function, we do not supply arguments. If we had supplied

Function docs



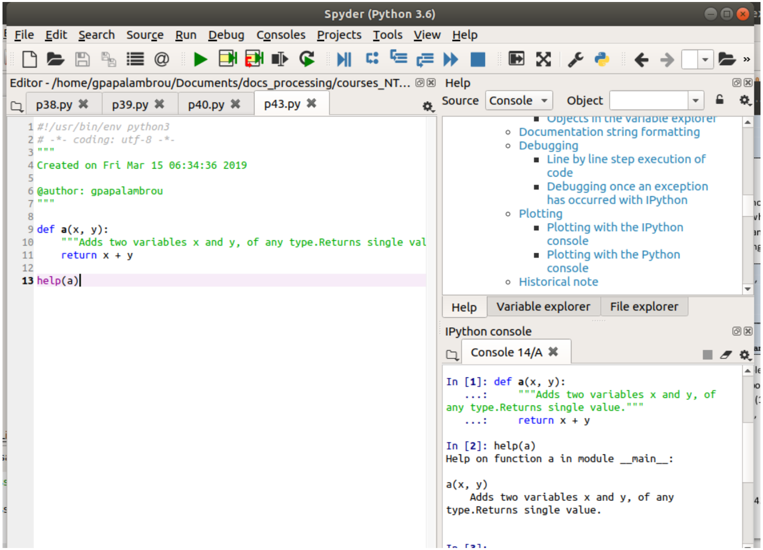
Function documentation

Functions can be self-documenting in Python. A *docstring* can be written after the `def` statement that provides a description of what a function does. This extremely useful for documenting your code and providing explanations that both you and subsequent users can use. The built-in `help` function uses docstrings to provide help about functions.

```
>>> def a(x, y):
...     """Adds two variables x and y, of any type.  Returns single value."""
...     return x + y
... <hit return>
>>> help(a)
Help on function a in module __main__:

a(x, y)
    Adds two variables x and y, of any type.  Returns single value.
```

It is typical to enclose docstrings using triple-quotes, since complex functions might require longer, multi-line documentation. It is a good habit to write docstrings for your code. Each should contain three pieces of information: (1) a basic description of what the function does, (2) what the function expects as arguments, and (3) what the function returns (including the variable types).

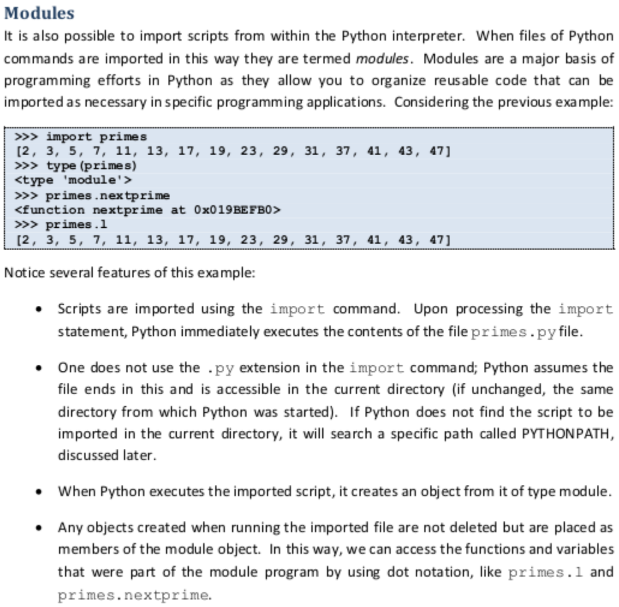
Functions - Spyder [p43.py]


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - /home/gpapalambrou/Documents/docs_processing/courses_NT...
p38.py p39.py p40.py p43.py
Help
Source Console Object
Documentation string formatting
Debugging
Line by line step execution of code
Debugging once an exception has occurred with IPython
Plotting
Plotting with the IPython console
Plotting with the Python console
Historical note
Help Variable explorer File explorer
IPython console
Console 14/A

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 15 06:34:36 2019

@author: gpapalambrou
"""

def a(x, y):
    """Adds two variables x and y, of any type.Returns single val
    return x + y

help(a)
```

```
In [1]: def a(x, y):
   ...:     """Adds two variables x and y, of
any type.Returns single value."""
   ...:     return x + y

In [2]: help(a)
Help on function a in module __main__:

a(x, y)
    Adds two variables x and y, of any
type.Returns single value.

In [3]:
```

Modules



Modules

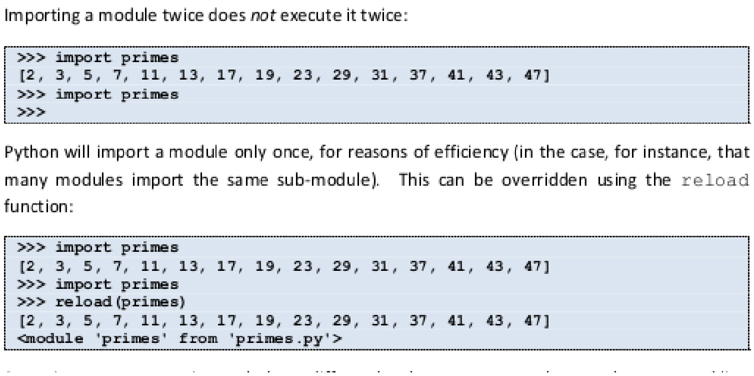
It is also possible to import scripts from within the Python interpreter. When files of Python commands are imported in this way they are termed *modules*. Modules are a major basis of programming efforts in Python as they allow you to organize reusable code that can be imported as necessary in specific programming applications. Considering the previous example:

```
>>> import primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> type(primes)
<type 'module'>
>>> primes.nextprime
<function nextprime at 0x019BEFB0>
>>> primes.l
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Notice several features of this example:

- Scripts are imported using the `import` command. Upon processing the `import` statement, Python immediately executes the contents of the file `primes.py` file.
- One does not use the `.py` extension in the `import` command; Python assumes the file ends in this and is accessible in the current directory (if unchanged, the same directory from which Python was started). If Python does not find the script to be imported in the current directory, it will search a specific path called PYTHONPATH, discussed later.
- When Python executes the imported script, it creates an object from it of type module.
- Any objects created when running the imported file are not deleted but are placed as members of the module object. In this way, we can access the functions and variables that were part of the module program by using dot notation, like `primes.l` and `primes.nextprime`.

Modules



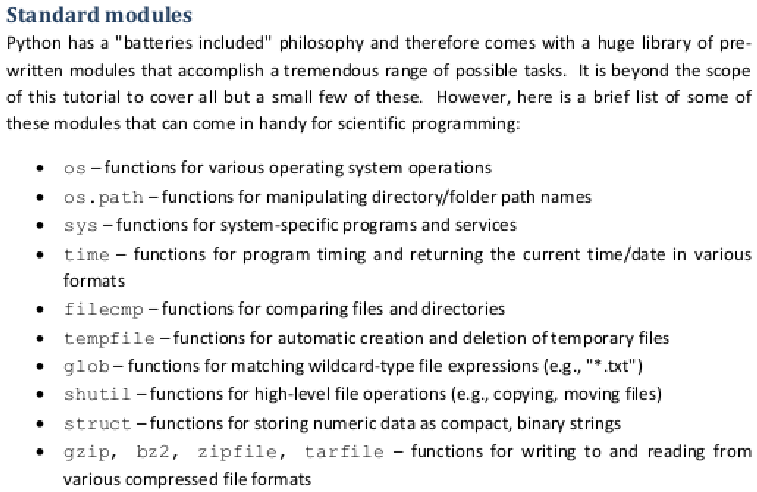
Importing a module twice does *not* execute it twice:

```
>>> import primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> import primes
>>>
```

Python will import a module only once, for reasons of efficiency (in the case, for instance, that many modules import the same sub-module). This can be overridden using the `reload` function:

```
>>> import primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> import primes
>>> reload(primes)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
<module 'primes' from 'primes.py'>
```

Modules

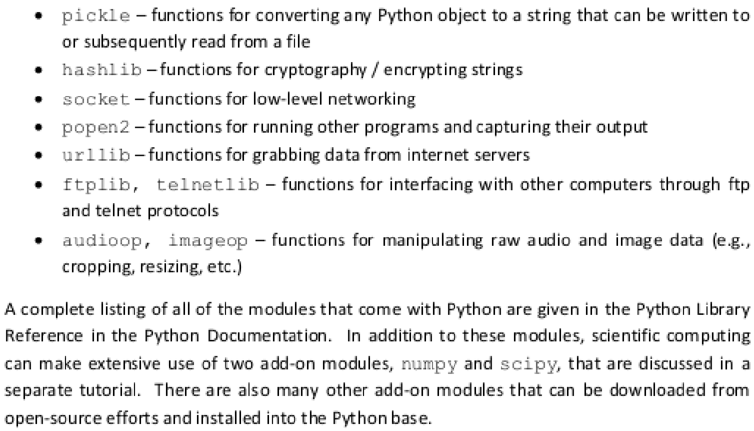


Standard modules

Python has a "batteries included" philosophy and therefore comes with a huge library of pre-written modules that accomplish a tremendous range of possible tasks. It is beyond the scope of this tutorial to cover all but a small few of these. However, here is a brief list of some of these modules that can come in handy for scientific programming:

- `os` – functions for various operating system operations
- `os.path` – functions for manipulating directory/folder path names
- `sys` – functions for system-specific programs and services
- `time` – functions for program timing and returning the current time/date in various formats
- `filecmp` – functions for comparing files and directories
- `tempfile` – functions for automatic creation and deletion of temporary files
- `glob` – functions for matching wildcard-type file expressions (e.g., "*.txt")
- `shutil` – functions for high-level file operations (e.g., copying, moving files)
- `struct` – functions for storing numeric data as compact, binary strings
- `gzip`, `bz2`, `zipfile`, `tarfile` – functions for writing to and reading from various compressed file formats

Modules



- `pickle` – functions for converting any Python object to a string that can be written to or subsequently read from a file
- `hashlib` – functions for cryptography / encrypting strings
- `socket` – functions for low-level networking
- `popen2` – functions for running other programs and capturing their output
- `urllib` – functions for grabbing data from internet servers
- `ftplib`, `telnetlib` – functions for interfacing with other computers through ftp and telnet protocols
- `audioop`, `imageop` – functions for manipulating raw audio and image data (e.g., cropping, resizing, etc.)

A complete listing of all of the modules that come with Python are given in the Python Library Reference in the Python Documentation. In addition to these modules, scientific computing can make extensive use of two add-on modules, `numpy` and `scipy`, that are discussed in a separate tutorial. There are also many other add-on modules that can be downloaded from open-source efforts and installed into the Python base.