Python

Libraries

A function is a way to turn a bunch of related statements into a single "chunk"

A function is a way to turn a bunch of related statements into a single "chunk"

– Avoid duplication

A function is a way to turn a bunch of related statements into a single "chunk"

– Avoid duplication
– Make code easier to read

A function is a way to turn a bunch of related statements into a single "chunk"

– Avoid duplication

– Make code easier to read

A *library* does the same thing for related functions

A function is a way to turn a bunch of related statements into a single "chunk"

– Avoid duplication

– Make code easier to read

A *library* does the same thing for related functions

Hierarchical organization

A function is a way to turn a bunch of related statements into a single "chunk"

- Avoid duplication
- Make code easier to read

A *library* does the same thing for related functions

Hierarchical organization

library

function

statement

Every Python file can be used as a library

Every Python file can be used as a library

Use import to load it

Every Python file can be used as a library

Use import to load it

```
# halman.py
def threshold(signal):
  return 1.0 / sum(signal)
```

Every Python file can be used as a library

Use import to load it

```
# halman.py
def threshold(signal):
  return 1.0 / sum(signal)
```

```
# program.py
import halman
readings = [0.1, 0.4, 0.2]
print 'signal threshold is', halman.threshold(readings)
```

Every Python file can be used as a library

Use import to load it

```
# halman.py
def threshold(signal):
  return 1.0 / sum(signal)
```

```
# program.py
import halman
readings = [0.1, 0.4, 0.2]
print 'signal threshold is', halman.threshold(readings)
```

```
$ python program.py
signal threshold is 1.42857
```

When a module is imported, Python:

When a module is imported, Python:

1. Executes the statements it contains

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

```
# noisy.py
print 'is this module being loaded?'
NOISE_LEVEL = 1./3.
```

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

```
# noisy.py
print 'is this module being loaded?'
NOISE_LEVEL = 1./3.
```

```
>>> import noisy
is this module being loaded?
```

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

```
# noisy.py
print 'is this module being loaded?'
NOISE_LEVEL = 1./3.
```

```
>>> import noisy
is this module being loaded?
>>> print noisy.NOISE_LEVEL
0.33333333
```

Each module is a *namespace*

Each module is a *namespace*

function

Each module is a *namespace*

$$\begin{tikzcd}
\text{module} & \\
 & \text{function} \arrow[ul, bend right]
\end{tikzcd}$$

Each module is a *namespace*

$$\begin{tikzcd}
\text{global} & & \\
& \text{module} \arrow[ul, bend right] & \\
& & \text{function} \arrow[ul, bend right]
\end{tikzcd}$$

Each module is a *namespace*

$$\begin{tikzcd}
\text{global} & & \\
& \text{module} \arrow[ul, bend right] & \\
& & \text{function} \arrow[ul, bend right]
\end{tikzcd}$$

```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

Each module is a *namespace*

$$\begin{tikzcd} \text{global} & & \\ & \text{module} \arrow[ul, bend right] & \\ & & \text{function} \arrow[ul, bend right] \end{tikzcd}$$

```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

```
>>> NAME = 'Hamunaptra'
```

Each module is a *namespace*

$$\begin{tikzcd}
\text{global} & & \\
& \text{module} \arrow[ul, bend right] & \\
& & \text{function} \arrow[ul, bend right]
\end{tikzcd}$$

```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

```
>>> NAME = 'Hamunaptra'
>>> import module
```

Each module is a *namespace*

$$\begin{tikzcd} \text{global} & & \\ & \text{module} \arrow[ul, bend right] & \\ & & \text{function} \arrow[ul, bend right] \end{tikzcd}$$

```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

```
>>> NAME = 'Hamunaptra'
>>> import module
>>> print module.func('!!!')
Transylvania !!!
```

Python comes with many standard libraries

Python comes with many standard libraries

```
>>> import math
```

Python comes with many standard libraries

```
>>> import math
>>> print math.sqrt(2)
1.4142135623730951
```

Python comes with many standard libraries

```
>>> import math
>>> print math.sqrt(2)
1.4142135623730951
>>> print math.hypot(2, 3)  # sqrt(x**2 + y**2)
3.6055512754639891
```

Python comes with many standard libraries

```
>>> import math
>>> print math.sqrt(2)
1.4142135623730951
>>> print math.hypot(2, 3)  # sqrt(x**2 + y**2)
3.6055512754639891
>>> print math.e, math.pi   # as accurate as possible
2.7182818284590451 3.1415926535897931
```

Python also provides a help function

Python also provides a help function

```
>>> import math
>>> help(math)
Help on module math:
NAME
    math
FILE
    /usr/lib/python2.5/lib-dynload/math.so
MODULE DOCS
    http://www.python.org/doc/current/lib/module-math.html
DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.
FUNCTIONS
    acos(...)
        acos(x)
        Return the arc cosine (measured in radians) of x.
⋮
```

And some nicer ways to do imports

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
```

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
>>> from math import hypot as euclid
>>> euclid(3, 4)
5.0
```

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
>>> from math import hypot as euclid
>>> euclid(3, 4)
5.0
>>> from math import *
>>> sin(pi)
1.2246063538223773e-16
>>>
```

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
>>> from math import hypot as euclid
>>> euclid(3, 4)
5.0
>>> from math import *
>>> sin(pi)
1.2246063538223773e-16
>>>
```

$\longleftarrow$ Generally a bad idea

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
>>> from math import hypot as euclid
>>> euclid(3, 4)
5.0
>>> from math import *
>>> sin(pi)
1.2246063538223773e-16
>>>
```

$\leftarrow$ Generally a bad idea

Someone could add to the library after you start using it

Almost every program uses the `sys` library

Almost every program uses the `sys` library

```
>>> import sys
```

Almost every program uses the `sys` library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
```

Almost every program uses the `sys` library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
>>> print sys.platform
win32
```

Almost every program uses the `sys` library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
>>> print sys.platform
win32
>>> print sys.maxint
2147483647
```

Almost every program uses the `sys` library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
>>> print sys.platform
win32
>>> print sys.maxint
2147483647
>>> print sys.path
['',
 'C:\\WINDOWS\\system32\\python27.zip',
 'C:\\Python27\\DLLs', 'C:\\Python27\\lib',
 'C:\\Python27\\lib\\plat-win',
 'C:\\Python27', 'C:\\Python27\\lib\\site-packages']
```

`sys.argv` holds command-line arguments

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

```
# echo.py
import sys
for i in range(len(sys.argv)):
   print i, " " + sys.argv[i] + " "
```

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

```
# echo.py
import sys
for i in range(len(sys.argv)):
   print i, " " + sys.argv[i] + " "
```

```
$ python echo.py
0 echo.py
$
```

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

```
# echo.py
import sys
for i in range(len(sys.argv)):
    print i, " " + sys.argv[i] + " "
```

```
$ python echo.py
0 echo.py
$ python echo.py first second
0 echo.py
1 first
2 second
$
```

`sys.stdin` is *standard input* (e.g., the keyboard)

`sys.stdin` is *standard input* (e.g., the keyboard)

`sys.stdout` is *standard output* (e.g., the screen)

`sys.stdin` is *standard input* (e.g., the keyboard)

`sys.stdout` is *standard output* (e.g., the screen)

`sys.stderr` is *standard error* (usually also the screen)

`sys.stdin` is *standard input* (e.g., the keyboard)

`sys.stdout` is *standard output* (e.g., the screen)

`sys.stderr` is *standard error* (usually also the screen)

See the Unix shell lecture for more information

Picking up changes in external libraries ("reload")

In some scenarios you will want to keep a python session running whilst modifying an external module.

Picking up changes in external libraries ("reload")

In some scenarios you will want to keep a python session running whilst modifying an external module. E.g...

```
>>> import mylib
>>> print mylib.x
33.8
>>> # change "mylib.py" now and get new x
```

Let's look in detail

```
>>> import mylib
>>> print mylib.x
33.8
```

Let's look in detail

```
>>> import mylib
>>> print mylib.x
33.8
```

Change `"mylib.py"` so that `x` is set to "hello" - and save the module.

```
>>> import mylib
>>> print mylib.x
33.8
```

Let's look in detail

```
>>> import mylib
>>> print mylib.x
33.8
```

Change `"mylib.py"` so that `x` is set to "hello" - and save the module.

```
>>> import mylib
>>> print mylib.x
33.8
```

Oh No! Python has ignored my changes.

We need to "reload"!!!

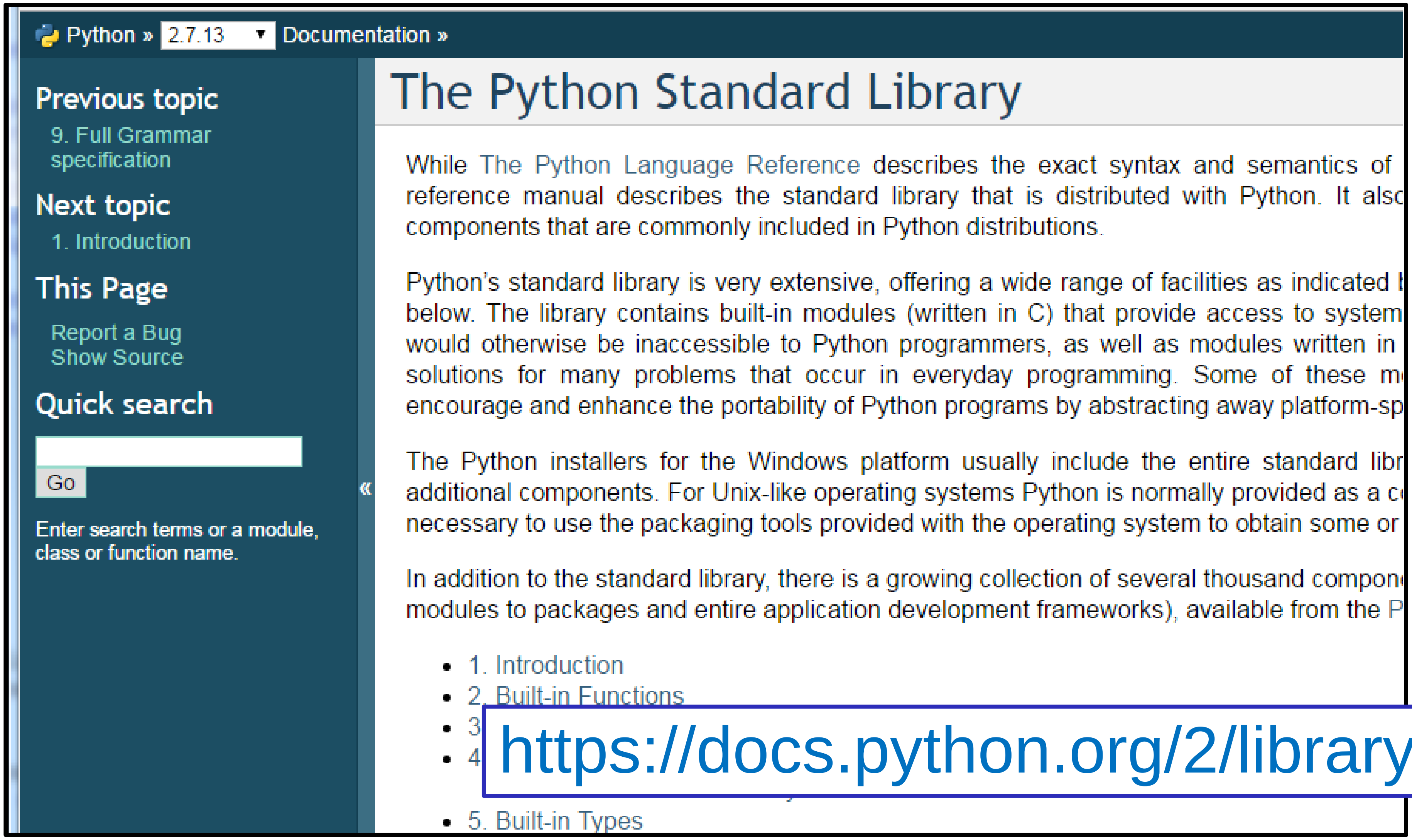
```
>>> import mylib
>>> print mylib.x
33.8
```

Change "`mylib.py`" so that `x` is set to "hello" - and save the module.

```
>>> reload(mylib)
>>> print mylib.x
'hello'
```

It worked!

Free stuff - the Python Standard Library



Python » 2.7.13 Documentation »

Previous topic
9. Full Grammar specification

Next topic
1. Introduction

This Page
Report a Bug
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

The Python Standard Library

While The Python Language Reference describes the exact syntax and semantics of reference manual describes the standard library that is distributed with Python. It also components that are commonly included in Python distributions.

Python's standard library is very extensive, offering a wide range of facilities as indicated below. The library contains built-in modules (written in C) that provide access to system would otherwise be inaccessible to Python programmers, as well as modules written in solutions for many problems that occur in everyday programming. Some of these m encourage and enhance the portability of Python programs by abstracting away platform-sp

The Python installers for the Windows platform usually include the entire standard libr additional components. For Unix-like operating systems Python is normally provided as a c necessary to use the packaging tools provided with the operating system to obtain some or

In addition to the standard library, there is a growing collection of several thousand compon modules to packages and entire application development frameworks), available from the P

- 1. Introduction
- 2. Built-in Functions
- 3
- 4
- 5. Built-in Types

https://docs.python.org/2/library/

More examples from the **Python Standard Library**

datetime:

```
>>> from datetime import date, timedelta
>>> today = date.today()
>>> print today
2017-02-25
>>> print today - timedelta(days=365)
2016-02-26
```

random:

```
random.random() # Random float x, 0 <= x < 1
0.37444887175646646
random.uniform(1, 10) # Random float x, 1 <= x < 10
1.1800146073117523
random.randint(1, 10) # Integer from 1 to 10, endpoints
included
7
```

https://docs.python.org/2/library/

urllib:

```
>>> import urllib2
>>> response = urllib2.urlopen('http://python.org/')
>>> print response.readlines()[:3]
['<!doctype html>\n', '<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->\n', '<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]-->\n']
```

https://docs.python.org/2/library/

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
$ python count.py < a.txt
48
$
```

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
$ python count.py < a.txt
48
$ python count.py b.txt
227
$
```

The more polite way

```
'''Count lines in files.  If no filename arguments given,
read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

The more polite way

```
'''Count lines in files.  If no filename arguments given,
read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

The more polite way

```
'''Count lines in files.  If no filename arguments given,
read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

If the first statement in a module or function is a string, it is saved as a *docstring*

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

```
>>> import adder
>>> help(adder)
NAME
    adder - Addition utilities.
FUNCTIONS
    add(a, b)
        Add arguments.
>>>
```

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

```
>>> import adder
>>> help(adder)
NAME
    adder - Addition utilities.
FUNCTIONS
    add(a, b)
        Add arguments.
>>> help(adder.add)
add(a, b)
        Add arguments.
>>>
```

When Python loads a module, it assigns a value

to the module-level variable __name__

When Python loads a module, it assigns a value to the module-level variable __name__

main program
'__main__'

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

$\leftarrow$ Always executed

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

← Always executed

← Only executed when file run directly

```
# stats.py
'''Useful statistical tools.'''

def average(values):
  '''Return average of values or None if no data.'''
  if values:
    return sum(values) / len(values)
  else:
    return None

if __name__ == '__main__':
  print 'test 1 should be None:', average([])
  print 'test 2 should be 1:', average([1])
  print 'test 3 should be 2:', average([1, 2, 3])
```

```
# test-stats.py
from stats import average
print 'test 4 should be None:', average(set())
print 'test 5 should be -1:', average({0, -1, -2})
```

```
# test-stats.py
from stats import average
print 'test 4 should be None:', average(set())
print 'test 5 should be -1:', average({0, -1, -2})
```

```
$ python stats.py
test 1 should be None: None
test 2 should be 1: 1
test 3 should be 2: 2
$
```

```
# test-stats.py
from stats import average
print 'test 4 should be None:', average(set())
print 'test 5 should be -1:', average({0, -1, -2})
```

```
$ python stats.py
test 1 should be None: None
test 2 should be 1: 1
test 3 should be 2: 2
$ python test-stats.py
test 4 should be None: None
test 5 should be -1: -1
$
```

created by

Greg Wilson

October 2010