Functions

Functions break up your code into reusable chunks
They take *arguments* of any type, and can return values

Example:

```
>>> def my_quadratic(x):
...     q = -x**2 + 2.*x + 3.
...     return q
...
>>> my_quadratic(0.)
>>> 3.0
>>> my_quadratic(-0.5)
>>> 1.75
>>> 3. * my_quadratic(1.)
>>> 12.0
```

Functions

Want to return more than one object? Use a tuple.

Example:

```
>>> def roots(a, b, c):
...     rD = math.sqrt(b**2 - 4.*a*c)
...     root1 = (-b + rD) / 2. / a
...     root2 = (-b - rD) / 2. / a
...     return (root1, root2)
...
>>> roots(1., 5., -14.)
>>> (2.0, -7.0)
```

i the brackets are optional

⚠ Warning! For various reasons (see later), this isn't always a very good way of calculating the roots of quadratic equations...

Functions and Scope

Variables defined inside a function are only accessible within that function
Variables outside the function are accessible from within it

```
>>> def my_func():
...     a = 4
...     c = 2
...     print( (a, b, c) )
...
>>> a = 3; b = 7
>>> my_func()
(4, 7, 2)
>>> a
3
>>> c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

i the function's a is the one used; `my_func()` also has access to b

⚠ oops - the function's c is 'out of scope'

Recursive Functions

Example: to calculate the factorial of a number,

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

factorial.py

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

print(factorial(10))
```

```
$ python factorial.py
3628800
```

Functions - Default Arguments

A function may be defined with default arguments to be used if they are not specified upon calling the function.
Example:

```
>>> def greet(name, title='Dr'):
...     print('Greetings, %s %s' % (title, name))
...
>>> greet('Smith', 'Professor')
Greetings, Professor Smith
>>> greet('Brown')
Greetings, Dr Brown
```

Functions - Keyword Arguments

The arguments to a function may be specified in any order using keywords:

```
>>> def func(a, b=4, c=3):
...     print('(a, b, c) =', (a, b, c))
...
>>> func(1,2)
(a, b, c) = (1, 2, 3)
>>> func(c=7, a=1)
(a, b, c) = (1, 4, 7)
>>> func(1, c=7)
(a, b, c) = (1, 4, 7)
>>> func(a=1, 5)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

arguments passed in any order

this is OK: b has a default value

oops - is 5 supposed to be b or c?

More on Functions

Arguments passed to Python functions are 'references, passed by value'. Some examples:

```
>>> def func1(a):
...     a = a + 1
...     print('a has been incremented to', a)
...
>>> a = 5
>>> func1(a)
a has been incremented to 6
>>> a
5
```

WTF?

a is an integer, an *immutable* type. The line `a = a + 1` creates a new object (the integer 6), local to `func1`. The outside variable `a` is not affected.

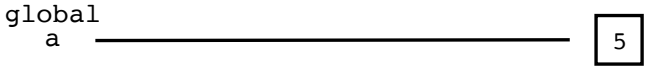
More on Functions

Passing *mutable* types, on the other hand...

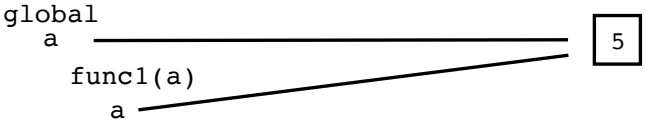
```
>>> def func2(b):
...     b[1] = 0
...     print('b is now', b)
...
>>> b = [1, 2, 3]
>>> func2(b)
b is now [1, 0, 3]
>>> b
[1, 0, 3]
```

`lists` are *mutable* objects: the line `b[1] = 0` changes the second item in the list referenced by the object `b` (a new list is not created). So the change appears outside the function too.

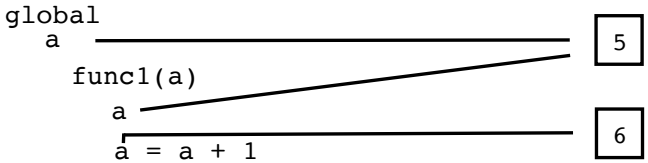
More on Functions



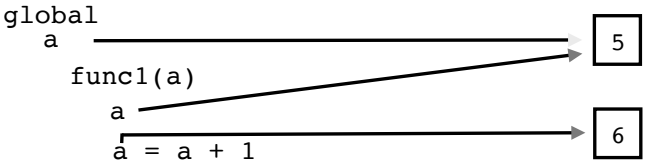
More on Functions



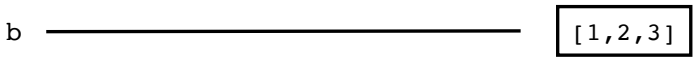
More on Functions



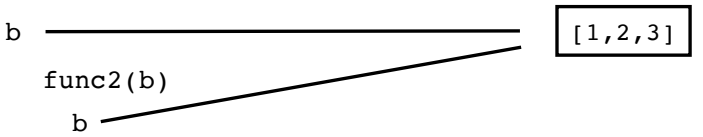
More on Functions



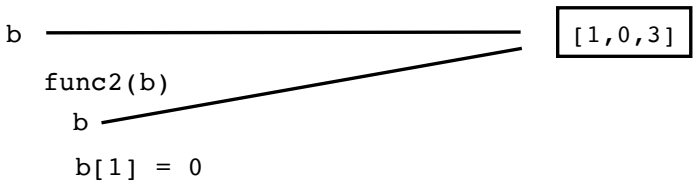
More on Functions



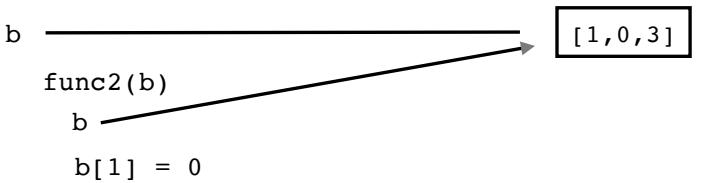
More on Functions



More on Functions



More on Functions



Modules

Longer programs are best split up into separate files (and even separate directories of files).

Each file, containing definitions of variables and functions is a *module*

Modules are *imported* into a Python program with the `import` statment

The Standard Python Library consists of modules (and packages of modules)

Modules

Some common Python modules:

`math`	Mathematical functions
`cmath`	Mathematical functions for complex arithmetic
`sys`	System-specific parameters and functions
`os`	Miscellaneous operating system interfaces
`random`	Generate (pseudo-)random numbers
`zlib`	A compression library (compatible with gzip)
`argparse`	Command-line argument parsing
`HTMLParser`	Simple HTML parsing
`urllib`	Open and access resources across the internet by URL
`datetime`	Basic date and time types
`re`	Regular expressions

and the packages `numpy`, `scipy`, and `matplotlib` ... see later.

The `math` module

Just some of the mathematical functions provided:

`sqrt(x)`	
`exp(x)`	
`log(x)`	natural logarithm
`log(x, base)`	
`log10(x)`	
`sin(x), etc.`	
`asin(x), etc.`	
`sinh(x), etc.`	
`hypot(x, y)`	`sqrt(x*x + y*y)`, the Euclidean norm
`erf(x)`	the error function (Python 2.7+ only)
`pi`	3.141592... to available precision
`e`	2.718281... to available precision

The `sys` module

System methods and constants, of which the most useful are:

`argv` The list of command line arguments passed to a Python script. `argv[0]` is the script name. e.g.

```
$ python my_script.py hello 4
```

results in

```
sys.argv[0] = 'my_script.py'
sys.argv[1] = 'hello'
sys.argv[2] = '4'
```

note that the argument appears as a string

`exit([n])` Exit from Python. If not provided, `n` defaults to `0`, indicating normal termination. Different non-zero values are used to indicate to the shell various errors.

The sys module

Example:

reciprocal.py

```
import sys

try:
    x = float(sys.argv[1])
except (IndexError, ValueError):
    sys.exit("I need a number, please.")
try:
    rx = 1./x
except ZeroDivisionError:
    sys.exit("You can't divide by zero!")
print(rx)
```

```
$ python reciprocal.py 0
You can't divide by zero!
```

The os module

Miscellaneous operating system interfaces

`path`	Manipulate file and directory names
`environ`	A mapping object representing the string environment. e.g. `os.environ['HOME']` is the pathname of your home directory (on some systems)
`remove`	Delete a file
`rename`	Rename a file
`stat`	'stat' a file (get information about its read/write permissions, last time of modification, etc.
`listdir`	Return a list of the entries in a directory

The os module

Example: get a list of all the entries in your home directory

```
>>> import os
>>> HOME = os.environ['HOME']
>>> print(os.listdir(HOME))
['.bash_history', '.bash_profile', 'Desktop',
'Documents, 'Downloads', 'Library', 'research',
'teaching', ... ]
```

i Python has a host of other methods for dealing with files and directories, including the modules `glob` and `shutil` (see the Python documentation)

The Joy of Sets

A Python **set** is an *unordered* collection of *unique*, *immutable* objects. Example:

```
>>> a = set()
>>> a.add(3)
>>> a.add(4)
>>> a.add('hello')
>>> a.add(4)
>>> a
set([3, 4, 'hello'])
```

i sets can be initialized with any iterable object:

```
>>> b = set([1,1,0,3,7,4,3,0,1,3,-1])
>>> b.add(6)
set([0, 1, 3, 4, 6, 7, -1])
```

i note that our 6 could pop up anywhere in the set - sets are not ordered.

The Joy of Sets

Some more `set` methods:

```
>>> a = set((0, 1, 2, 3))
>>> b = set((2, 3, 4, 5))
>>> a & b
set([2, 3])
```

intersection of sets

```
>>> a | b
set([0, 1, 2, 3, 4, 5])
>>> a.remove(0)
>>> a
set([1, 2, 3])
>>> c = set([1,2])
>>> c.issubset(a)
True
```

union of sets

subsets

... and many more (see documentation)

Dictionaries

A Python *dictionary*, `dict`, is an *unordered* collection of objects (*values*) referenced by *keys*, which can be a large range of immutable Python objects (integers, strings, tuples, etc.)

Example:

```
>>> heights = {'Eiffel Tower': 324.,
               'The Shard': 308.,
               'Fernsehturm': 368.,
              }
>>> heights['The Shard']
308.
```

```
>>> for name in heights:
...     print(name, heights[name])
Fernsehturm 368.
Eiffel Tower 324.
The Shard 308.
```

`dicts` are iterable

Dictionaries have no particular order

Dictionaries

More dictionary methods and examples

```
>>> a = {}
>>> a['A'] = 65
>>> a[0] = 'nought'
>>> a[(1,2,3)] = 3.14
```

an empty `dict`

```
>>> list(a.keys())
['A', 0, (1, 2, 3)]
```

a list of a's keys

```
>>> list(a.values())
[65, 'nought', 3.14]
```

a list of a's values

```
>>> list(a.items())
[('A', 65), (0, 'nought'), ((1, 2, 3),
3.14)]
```

a list of (key, value) tuples

Dictionaries

The `get` dictionary method.

```
>>> d = {'Cu': 8.96, 'Fe': 7.87,
         'Ni': 8.91, 'Pt': 21.45}

>>> d['Zn']
...
KeyError: 'Zn'
```

No such key

```
>>> print(d.get('Zn'))
None
```

Default return value is `None`

```
>>> d.get('Zn', 0)
0
```

Can set the default return value

Dictionaries

More dictionary methods and examples

```
>>> a = {'March': 'mars', 'April': 'avril',
May': 'mai', 'June': 'juin'}
>>> for m_en in sorted(a.keys()):
...     print(m_en, a[m_en])
April avril
June juin
March mars
May mai
```

note the use of the `sorted()` method to produce an ordered list of a's keys