

Computational Finance



Preliminaries

General Information

- My name is Simon Broda. You can find me at REC E4.27 (by appointment).
Email: s.a.broda@uva.nl (<mailto:s.a.broda@uva.nl>)
- Format of this course: 12 lectures of 2h each (2 per week), plus computer labs.
- Final grade based on a group assignment (groups of two; 40%), an individual assignment (35%), and a final exam (closed book, 2h; 25%).
- Additional exercises will be made available but not graded.

Material

- These lecture slides. Available on Blackboard (<https://blackboard.uva.nl>), Github (<https://github.com/s-broda/ComputationalFinance>), and Microsoft Azure (<https://notebooks.azure.com/s-broda/libraries/ComputationalFinance>).
- Book: Yves Hilpisch. Python for Finance: Analyze Big Financial Data. O'Reilly, 2014. ISBN 978-1-4919-4528-5 (603 pages, c. EUR 31). Code is available on Github (<https://github.com/yhilpisch/py4fi>).
- Further reading:
 - Python documentation (<https://docs.python.org/2/index.html>)
 - Yves Hilpisch. Derivatives Analytics with Python. Wiley, 2015. ISBN 978-1-119-03799-6 (374 pages, c. EUR 72). Code is available on Github (<https://github.com/yhilpisch/dawp>).
 - John C. Hull. Options, Futures and Other Derivatives. 8th Edition (or later), Prentice Hall, 2012. ISBN 978-0273759072 (847 pages, c. EUR 58).
 - Python for Data Analysis. 2nd Edition, O'Reilly, 2017. ISBN 978-1-4919-5766-0 (544 pages, c. EUR 34). Code is available on Github (<https://github.com/wesm/pydata-book>).

Outline and Reading List

- Unless noted otherwise, the reading list below refers to Hilpisch (2014).

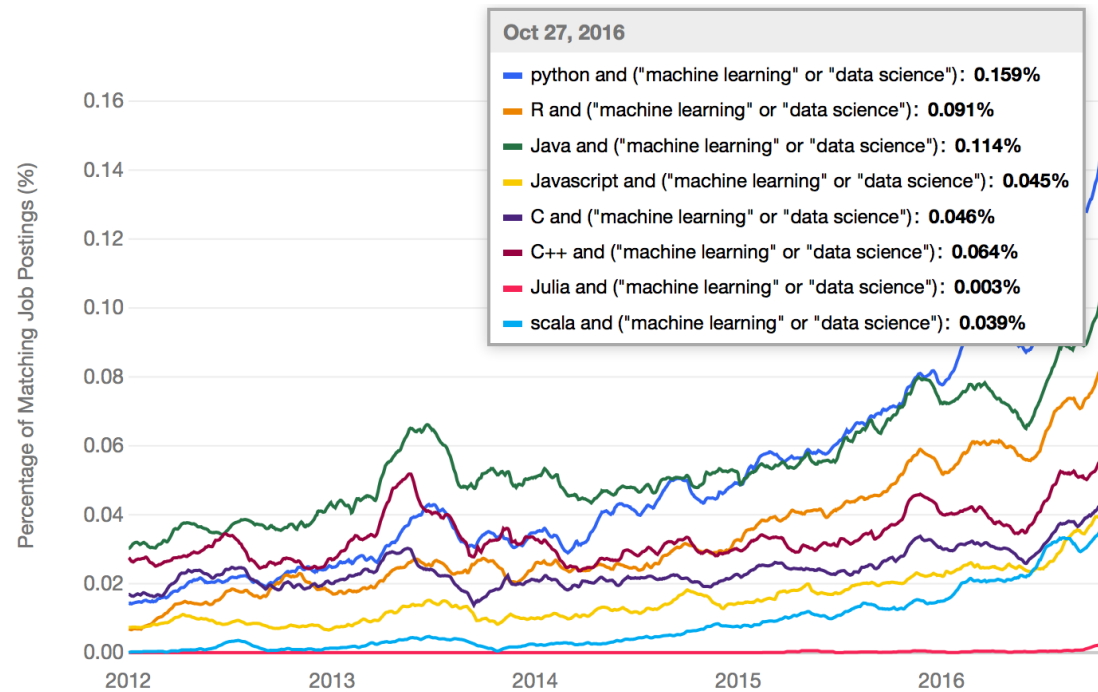
Week	Topic	Read
1	Introduction to Python	Ch. 2, Ch. 4 (pp. 79-95)
2	Dealing with Data	Ch. 4 (pp. 95-108), Ch. 6, App. C
3	Risk Measures; Plotting	Ch. 5, Ch. 10 (pp. 398-301), Ch. 11 (pp. 307-322)
4	Binomial Trees	Ch. 8 (pp. 218-223); Hull, Ch. 12, Ch. 20.1-20.5
5	Monte Carlo Methods	Ch. 10 (pp. 265-287, 290-294)
6	Variance Reduction Techniques	Ch. 10 (pp. 287-290)

Introduction to Python

Why Python?

- General purpose programming language, unlike, e.g., Matlab®.
- High-level language with a simple syntax, interactive (*REPL*: read-eval-print loop). Hence ideal for rapid development.
- Vast array of libraries available, including for scientific computing and finance.
- Native Python is usually slower than compiled languages like C++. Alleviated by highly optimized libraries, e.g. NumPy for calculations with arrays.
- Free and open source software. Cross-platform.
- Python skills are a marketable asset: most popular language for data science.

Job Postings on Indeed.com



Source ([https://www.ibm.com/developerworks/community/blogs/ifp/entry/What Language Is Best For Machine Learning And Data Science?lang=en](https://www.ibm.com/developerworks/community/blogs/ifp/entry/What_Language_Is_Best_For_Machine_Learning_And_Data_Science?lang=en))

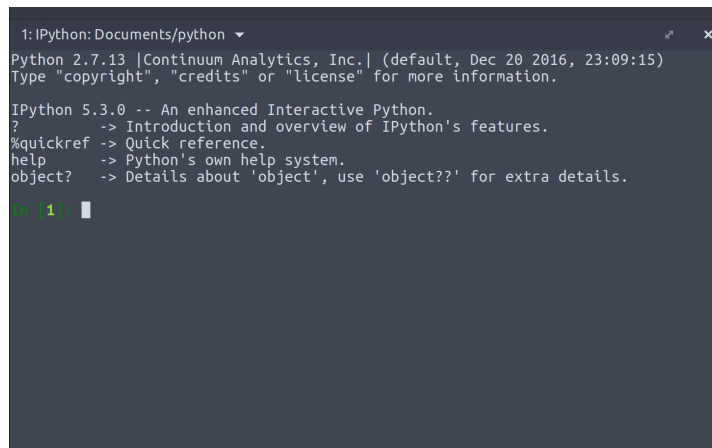
Obtaining Python

- Anaconda is a Python distribution, developed by Continuum Analytics, and specifically designed for scientific computing.
- Comes with its own package manager (conda). Many important packages (the *SciPy stack*) are pre-installed.
- Two versions: Python 2.7 and 3.6. Like the book, we will be using Python 2.7, which is still the industry standard. Most of our code should run on both with minimal adjustments.
- Obtain it from [here \(https://www.anaconda.com/download/\)](https://www.anaconda.com/download/). I recommend adding it to your PATH upon installation.
- Install the RISE plugin:

```
In [1]: #uncomment the next line to install. Note: "!" executes shell commands.  
        #!conda install -c damianavila82 rise
```

IPython Shell

- Python features a *read-eval-print loop* (REPL) which allows you to interact with it.
- The most bare-bones way to interact with it is via the *IPython shell*:

A screenshot of the IPython shell interface. The window title is "1: IPython: Documents/python". The text inside shows the Python version (2.7.13) and the IPython version (5.3.0). It lists several commands and their descriptions: "?", "%quickref", "help", and "object?". The prompt "In [1]: " is visible at the bottom left of the shell area.

```
1: IPython: Documents/python
Python 2.7.13 [Continuum Analytics, Inc.] (default, Dec 20 2016, 23:09:15)
Type "copyright", "credits" or "license()" for more information.

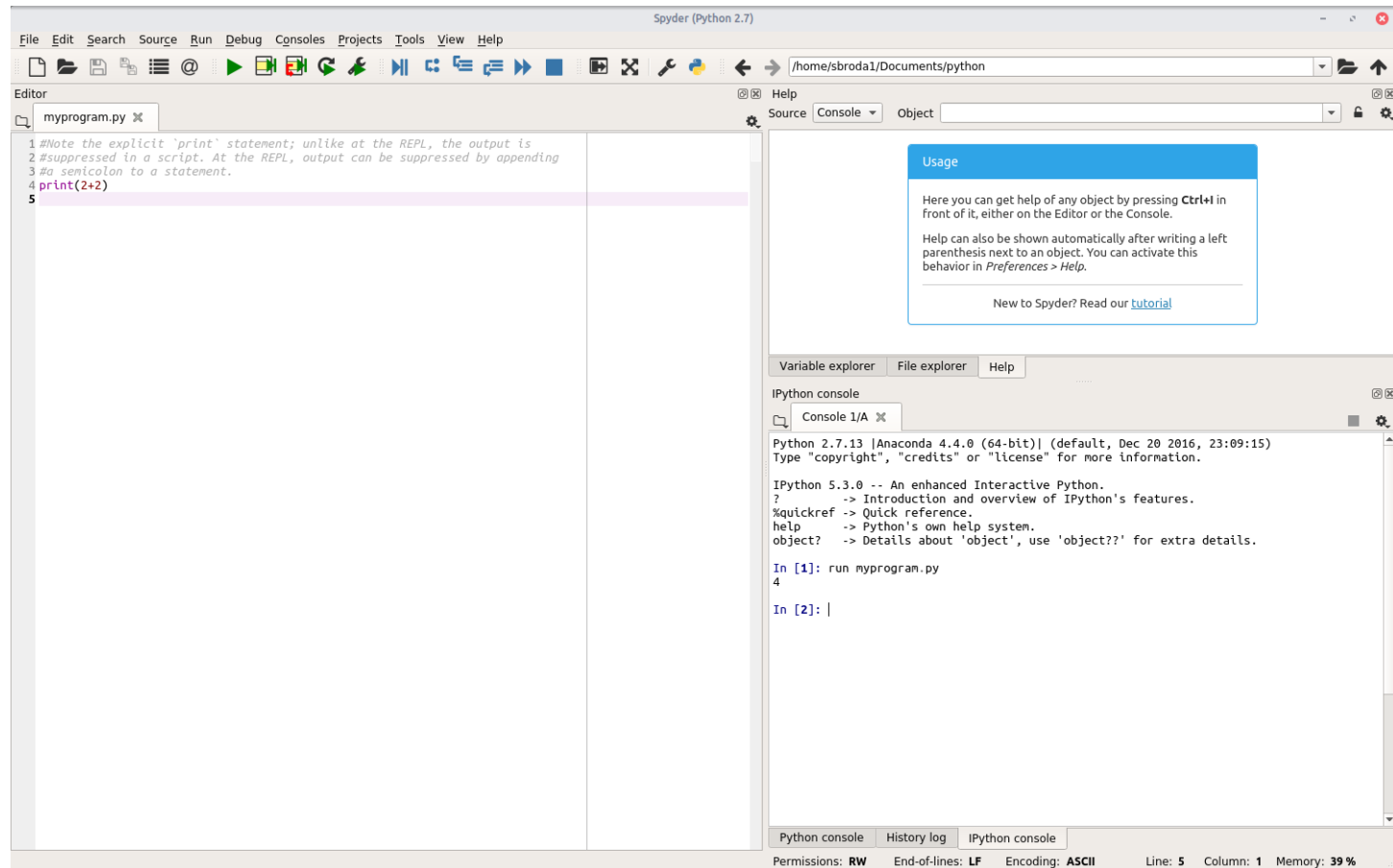
IPython 5.3.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]:
```

- For now you can treat it like a fancy calculator. Try entering `2+2`. Use `quit()` or `exit()` to quit, `help()` for Python's interactive help.

Writing Python Programs

- Apart from using it interactively, we can also write Python programs so we can rerun the code later.
- A Python program (called a *script* or a *module*) is just a text file, typically with the file extension `.py`.
- Contains Python commands and comments (introduced by the `#` character)
- To execute a program, do `run filename.py` in IPython (you may need to navigate to the right directory by using the `cd` command).
- While it's possible to code Python using just the REPL and a text editor, many people prefer to use an *integrated development environment* (IDE).
- Anaconda comes with an IDE called *Spyder* (Scientific PYthon Development EnviRonment), which integrates an editor, an IPython shell, and other useful tools.



Jupyter Notebooks

- Another option is the *Jupyter notebook* (JULia PYThon (e) R, formerly known as IPython notebook).
- Web app that allows you to create documents (*.ipynb) that contain text (formatted in Markdown (<https://daringfireball.net/projects/markdown>)), live code, and equations (formatted in $LAT_{E}X$).
- In fact these very slides are based on Jupyter notebooks. You can find them on my Github page (<https://github.com/s-broda/ComputationalFinance>).
- The slides are also available on my Microsoft Azure page (<https://notebooks.azure.com/s-broda/libraries/ComputationalFinance>), where you can also see them as a slide show and/or run them (after cloning them; requires a free Microsoft account).

Untitled

localhost:8888/notebooks/week1/Untitled.ipynb?kernel_name=python2

Search

☆

📄


⬇

🏠

📧


🔴

☰

 Jupyter

Untitled

Last Checkpoint: a minute ago (autosaved)

 Logout

File

Edit

View

Insert


Cell

Kernel


Widgets













Help

Trusted



Python 2



          Code  

A Jupyter Notebook ¶

Jupyter notebooks can contain live code:

```
In [3]: 2+2
```

```
Out[3]: 4
```

And \textit{TeX} equations:

$$a^2 + b^2 = c^2$$

Markdown

Text can be formatted using Markdown: *italics*, **bold**,

- * an
- * unnumbered
- * list

```
In [ ]: 
```

- A notebook consists of cells, each of which is either designated as Markdown (for text and equations), or as code.
- You should take a moment to familiarize yourself with the keyboard shortcuts. E.g., enter enters edit mode, esc enters command mode, ctrl-enter evaluates a cell, shift-enter evaluates a cell and selects the one below.
- Useful references:
 - Jupyter documentation (<http://jupyter-notebook.readthedocs.io/en/latest/index.html>)
 - Markdown cheat sheet (<https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>)
 - Latex math cheat sheet (<https://en.wikibooks.org/wiki/LaTeX/Mathematics>)

Python Basics

Variables

- A variable is a named memory location. It is assigned using "=" (technically, "=" binds the name on the LHS to the result of the expression on the RHS).

```
In [2]: a=2  
        a=a+1 #bind the value a to the result of the expression a+1  
        print(a) #show the result
```

3

```
In [3]: a+=1 #shorthand for a=a+1  
        print(a)
```

4

- Variable names can be made up from letters, numbers and the underscore. They may not start with a number. Python is case-sensitive: A is not the same as a.

Built-in Types

Attributes and Methods

- Any Python object has a *type*.
- One can use the `type` function to show the type of an object:

```
In [4]: type(a) #Functions take one or more inputs (in parens) and return an output.
```

```
Out[4]: int
```

- Objects can have attributes and methods associated with them:

```
In [5]: a.real #an attribute (internal variable stored inside an object)
```

```
Out[5]: 4
```

```
In [6]: a.bit_length() #a method (function that operates on objects of a particular type)
```

```
Out[6]: 3
```

Numeric Types

- Computers distinguish between integers and floating point numbers.
- Python integers can be arbitrary large (will use as many bits as necessary)
- Python floats are between $\pm 1.8 \cdot 10^{308}$, but are stored with just 64 bit precision.
- Hence floating point arithmetic is not exact:

```
In [7]: a=1.0;type(a) #Note that variables can change type: a was an int before
```

```
Out[7]: float
```

```
In [8]: a-0.9
```

```
Out[8]: 0.09999999999999998
```


Arithmetic

- The basic arithmetic operations are `+`, `-`, `*`, `/`, and `**` for exponentiation:

```
In [9]: 2*(3-1)**2
```

```
Out[9]: 8
```

- If any of the operands is `float`, then Python will convert the others to `float` too:

```
In [10]: 2*(3-1.0)**2
```

```
Out[10]: 8.0
```

- Note that `/` performs floor division in Python 2.7 (not 3.6) when both arguments are `ints`:

```
In [11]: c=3  
         c/2
```

```
Out[11]: 1
```

- Need to convert one argument to float to get the usual division:

```
In [12]: c/2.0
```

```
Out[12]: 1.5
```

```
In [13]: float(c)/2
```

```
Out[13]: 1.5
```

Booleans

- A `bool` can take one of two values: `True` or `False`.
- They are returned by *relational operators*: `<`, `<=`, `>`, `>=`, `==` (equality), `!=` (inequality)
- Can be combined using the *logical operators* `and`, `or`, and `not`.

```
In [14]: 1<=2<4
```

```
Out[14]: True
```

```
In [15]: 1<2 and 2<1
```

```
Out[15]: False
```

```
In [16]: not(1<2)
```

```
Out[16]: False
```

Sequence Types: Containers with Integer Indexing

Strings

- Strings hold text. Constructed using either single or double quotes.

```
In [17]: s1="Python"; s2=' is easy'; s1+s2 #Concatenation
```

```
Out[17]: 'Python is easy'
```

- Strings can be indexed into:

```
In [18]: s1[0] #Note zero-based indexing
```

```
Out[18]: 'P'
```

```
In [19]: s1[-1] #Negative indexes count from the right:
```

```
Out[19]: 'n'
```

- Can also pick out several elements ("slicing"). Works for all *sequence types* (lists, NumPy arrays, ...)

```
In [20]: s1[0:2] #Elements 0 and 1; left endpoint is included, right endpoint excluded.
```

```
Out[20]: 'Py'
```

```
In [21]: s1[0:6:2] #start:stop:step
```

```
Out[21]: 'Pto'
```

```
In [22]: s1[::-1] #start and stop can be omitted; default to 0 and len(str)
```

```
Out[22]: 'nohtyP'
```

- Strings are *immutable*:

```
In [23]: #Wrapping this in a try block so the error doesn't break `Run all` in Jupyter  
try:  
    s1[0]="C" #This errors  
except TypeError as e:  
    print(e)
```

```
'str' object does not support item assignment
```

- Python has many useful methods for strings:

```
In [24]: print(', '.join(filter(lambda m: callable(getattr(s1, m)) and not m.startswith("_"), dir(s1))))
```

```
capitalize, center, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill
```

```
In [25]: help(s1.upper)
```

Help on built-in function upper:

```
upper(...)
    S.upper() -> string
```

Return a copy of the string S converted to uppercase.

```
In [26]: (s1+s2).replace('easy', 'hard').upper()
```

```
Out[26]: 'PYTHON IS HARD'
```

Lists

- Lists are indexable collections of arbitrary (though usually homogenous) things:

```
In [27]: list1=[1, 2., 'hi']; print(list1)  
[1, 2.0, 'hi']
```

- The function `len` returns the length of a list (or any other sequence):

```
In [28]: len(list1)
```

```
Out[28]: 3
```

- Like strings, they support indexing, but unlike strings, they are *mutable*:

```
In [29]: list1[2]=42; print(list1)  
[1, 2.0, 42]
```

- Note the following:

```
In [30]: list2=list1 #bind the name list2 to the object list1. This does not create a copy:  
list2[0]=13  
print(list1) #list2 and list1 are the _same_ object!  
  
[13, 2.0, 42]
```

```
In [31]: list3=list1[:] #This creates a copy  
list3==list1 #Tests if all elements are equal
```

```
Out[31]: True
```

```
In [32]: list3 is list1 #Tests if list3 and list1 refer to the same object
```

```
Out[32]: False
```

```
In [33]: list2 is list1
```

```
Out[33]: True
```


- Lists of integers can be constructed using the range function:

```
In [34]: range(1,11,2) #start,stop[,step]
```

```
Out[34]: [1, 3, 5, 7, 9]
```

```
In [35]: range(5) #start and step can be omitted
```

```
Out[35]: [0, 1, 2, 3, 4]
```

- *List comprehensions* allow creating lists programmatically:

```
In [36]: [x**2 for x in range(1,10) if x>3 and x<7]
```

```
Out[36]: [16, 25, 36]
```

- The `for` and `if` statements will be discussed in more detail later.

- Methods for lists:

```
In [37]: print(' ', '.join(filter(lambda m: callable(getattr(list1, m)) and not m.startswith("_"), dir(list1))))
```

append, count, extend, index, insert, pop, remove, reverse, sort

```
In [38]: list1.append(13); #append 13 to the list `l1`  
print(list1)
```

[13, 2.0, 42, 13]

```
In [39]: list1.remove(13) #remove first occurrence of 13 from l1  
print(list1)
```

[2.0, 42, 13]

- Note: Table 4-2 in the book incorrectly states that `remove[i]` removes the element at index `i`. For that, use

```
In [40]: del(list1[0]); print(list1)
```

[42, 13]

- `del` can also be used to delete variables (technically, unbind the variable name)

xranges

- an xrange is similar to a list created with range, but they are more memory efficient because the list elements are created on demand (*lazily*).

```
In [41]: xrange(1,10,2)[3]
```

```
Out[41]: 7
```

Tuples

- Tuples are immutable sequences. They are created with round brackets:

```
In [42]: (1, 2., 'hi')
```

```
Out[42]: (1, 2.0, 'hi')
```

Other built-in datatypes

- Other built-in datatypes include `sets` (unordered collections) and `dicts` (collections of key-value pairs). See Hilpisch (2014), pp. 92-94.

Control Flow

- Control flow refers to the order in which commands are executed within a program.
- Often we would like to alter the linear way in which commands are executed.

Examples:

1. *Conditional branch*: Code that is only evaluated if some condition is true.
2. *Loop*: Code that is evaluated more than once.

Conditional Branch: The `if-else` statement

```
In [43]: x=3#uncomment the next line for interactive use
#x=int(raw_input("Enter a number between 0 and 9: ")) #`raw_input` returns a string
. `int` converts to integer.
if x<0:
    print("You entered a negative number.")
elif x>9:
    print("You entered a number greater than 9.")
else:
    print("Thank you. You entered %s." %x) #String interpolation.
```

Thank you. You entered 3.

- Notes:
 1. Code blocks are introduced by colons and *have* to be indented.
 2. The `if` block is executed only if the first condition is true
 3. The optional `elif` (short for 'else if') block is executed only if the first condition is false and the second one is true. There could be more than one.
 4. The optional `else` block is executed when none of the others is.

While loops

- Similar to `if`, but jumps back to the `while` statement after the `while` block has finished.
- The `else` block is executed when the condition becomes false (not if the loop is exited through a `break` statement; see next).

```
In [44]: x=1#set this to -1 to run
while x<0 or x>9:
    x=int(raw_input("Enter a number between 0 and 9: "))
    if x<0:
        print("You entered a negative number.")
    elif x>9:
        print("You entered a number greater than 9.")
else:
    print("Thank you. You entered %s." %x)
```

Thank you. You entered 1.

- Alternative implementation:

```
In [45]: while False: #Change to True to run
          x=int(raw_input("Enter a number between 0 and 9: "))
          if x<0:
              print("You entered a negative number.")
              continue #skip remainder of loop body, go back to `while`
          if x>9:
              print("You entered a number greater than 9.")
              continue
          print("Thank you. You entered %s." %x)
          break #exit innermost enclosing loop
```


For Loops

- A for loop iterates over the elements of a sequence (e.g., a list):

```
In [46]: for letter in "Python":  
         print(letter)
```

```
P  
y  
t  
h  
o  
n
```

- `letter` is called the loop variable. Every time the loop body is executed, it will in turn assume the value of each element of the sequence.

- For loops are typically used to execute a block of code a pre-specified number of times; range and xrange are often used in that case:

```
In [47]: squares=[]  
         for i in xrange(10):  
             squares.append(i**2)  
         print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Question: What does the following compute?

```
In [48]: n=7  
         f=1  
         for i in xrange(n):  
             f*=i+1
```

Modules

- Python's functionality is organized in *Modules*.
- Some of these are part of Python's *standard library* (e.g., `math`). Others are part of *packages*, many of which come preinstalled with Anaconda (e.g., `numpy`).
- Modules need to be imported in order to make them available:

```
In [49]: import math  
         math.factorial(7)
```

```
Out[49]: 5040
```

- Can use *tab completion* to discover functions defined by `math`: after importing, enter `math.` and press the Tab key. Alternatively, use `dir(math)`:

```
In [50]: print(' ', '.join(filter(lambda m: not m.startswith("_"), dir(math)))) #just so the o  
        utput fits on the slide
```

```
acos, acosh, asin, asinh, atan, atan2, atanh, ceil, copysign, cos, cosh, degrees  
, e, erf, erfc, exp, expm1, fabs, factorial, floor, fmod, frexp, fsum, gamma, hy  
pot, isinf, isnan, ldexp, lgamma, log, log10, loglp, modf, pi, pow, radians, sin  
, sinh, sqrt, tan, tanh, trunc
```

- Note that importing the module does not bring the functions into the *global namespace*: they need to be called as `module.function()`
- A function can be imported into the global namespace like so:

```
In [51]: from math import factorial  
factorial(7)
```

```
Out[51]: 5040
```

- It is possible to import all functions from the module into the global namespace using `from math import *`, but this is frowned upon; it pollutes the namespace, which may lead to name collisions.
- Packages can contain several modules. They are imported the same way:

```
In [52]: import numpy  
numpy.random.rand()
```

```
Out[52]: 0.6734649180161542
```

- Optionally, you can specify a shorthand name for the imported package/module:

```
In [53]: import numpy as np  
np.sqrt(2.0) #Note that this is not the same function as math.sqrt
```

```
Out[53]: 1.4142135623730951
```

- Conventions have evolved for the shorthands of some packages (e.g., np for numpy). Following them improves code readability.
- For the same reason, it is good practice to put your `import` statements at the beginning of your document (which I didn't do here).

Functions

Defining Functions

- User defined functions are declared using the `def` keyword:

```
In [54]: def mypower(x, y): #zero or more arguments, here two
        """Compute x^y."""
        return x**y
        mypower(2, 3) #positional arguments
```

Out[54]: 8

- The *docstring* is shown by the help function:

```
In [55]: help(mypower)
```

Help on function mypower in module `__main__`:

```
mypower(x, y)
    Compute x^y.
```

Several outputs

- Functions can have more than one output argument:

```
In [56]: def plusminus(a, b):  
         return a+b, a-b  
c, d = plusminus(1, 2); c, d
```

```
Out[56]: (3, -1)
```

Keyword arguments

- Instead of *positional arguments*, can also pass in *keyword arguments*:

```
In [57]: mypower(y=2, x=3)
```

```
Out[57]: 9
```

- Functions can specify *default arguments*:

```
In [58]: def mypower(x, y=2): #default arguments have to appear at the end  
        """Compute x^y.  
        """  
        return x**y  
mypower(3)
```

```
Out[58]: 9
```

```
In [59]: mypower(3, 3)
```

```
Out[59]: 27
```


Variable Scope

- Variables defined in functions are local (not visible in the calling scope):

```
In [60]: def f():  
         z=1  
         f()
```

```
In [61]: try:  
         print(z) #x is local to function f  
except NameError as e:  
         print(e)
```

name 'z' is not defined

Calling Convention

- Python uses a *calling convention* known as *call by object reference*.
- This means that modifications to its arguments made by a function are visible to the caller (i.e., outside the function):

```
In [62]: x=[1] #Recall that lists are mutable  
def f(y):  
    y[0]=2 #Note no return statement; equivalent to `return None`  
f(x);print(x) #Note that x has been modified in calling scope
```

[2]

Nested Functions

- Functions can be defined inside other functions. They will only be visible to the enclosing function.
- Nested functions can see variables defined in the enclosing function.

```
In [63]: def mypower(x, y):  
        def helper(): #no need to pass in x and y  
            return x**y #because the nested function can see them  
        a=helper()  
        return a  
mypower(2, 3)
```

Out[63]: 8

Advanced Material on Functions

Splatting and Slurping

- Splatting: passing the elements of a sequence into a function as positional arguments, one by one.

```
In [64]: def mypower(x, y):  
         return x**y  
args=[2, 3] #a list or a tuple  
mypower(*args) #splat (unpack) positional arguments into function
```

Out[64]: 8

- We can splat keyword arguments too, but we need to use a dict (key-value store):

```
In [65]: kwargs={'y': 3, 'x': 2} # a dict  
mypower(**kwargs) #splat keyword arguments
```

Out[65]: 8

- Slurping allows us to create *vararg* functions: functions that can be called with any number of positional and/or keyword arguments.

```
In [66]: def myfunc(*myargs, **mykwargs):  
          for (i, a) in enumerate(myargs): print("The %sth positional argument was %s." %  
          (i, a))  
          for a in mykwargs: print("Got keyword argument %s=%s." % (a, mykwargs[a]))  
          myfunc(0, 1, x=2, y=3)
```

```
The 0th positional argument was 0.  
The 1th positional argument was 1.  
Got keyword argument y=3.  
Got keyword argument x=2.
```

- The asterisk means "collect all (remaining) positional arguments into a tuple".
- The double asterisk means "collect all (remaining) keyword arguments into a dict".

Closures

- Functions are *first class objects* in Python
- This implies that functions can return other functions.
- Such functions are called *closures*, because they close about (capture) the local variables of the enclosing function.

```
In [67]: def makemultiplier(factor):  
        """Return a function that multiplies its argument by `factor`."""  
        def multiplier(x):  
            return x*factor  
        return multiplier  
timesfive=makemultiplier(5)  
type(timesfive)
```

```
Out[67]: function
```

```
In [68]: timesfive(3)
```

```
Out[68]: 15
```

Anonymous Functions

- Anonymous functions (or *lambdas*) are functions without a name (duh...) and whose function body is a single expression.
- They are often useful for functions that are needed only once (e.g., to return from a function, or to pass to a function)
- E.g., the previous example could be written

```
In [69]: def makemultiplier(factor):  
        """Return a function that multiplies its argument by `factor`."""  
        multiplier = lambda x: x*factor  
        return multiplier  
timesfive=makemultiplier(5)  
timesfive(3)
```

```
Out[69]: 15
```