Introduction to Python programming

Behzad Tabibian (btabibian@tuebingen.mpg.de (mailto:btabibian@tuebingen.mpg.de))
http://btabibian.github.io (http://btabibian@github.io)

J.R. Johansson (robert@riken.jp (mailto:robert@riken.jp)) http://dml.riken.jp/~rob/ (http://dml.riken.jp/~rob/)

The latest version of this IPython notebook (http://ipython.org/notebook.html) lecture is available at http://github.com/btabibian/scientific-python-lectures (http://github.com/btabibian/scientific-python-lectures).

The other notebooks in this lecture series are indexed at http://btabibian.github.io/notebooks/learnpython/ (http://btabibian.github.io/notebooks/learnpython/).

Jupyter notebooks

This file - an IPython notebook - does not follow the standard pattern with Python code in a text file. Instead, an IPython notebook is stored as a file in the JSON (http://en.wikipedia.org/wiki/JSON) format. The advantage is that we can mix formatted text, Python code and code output. It requires the IPython notebook server to run it though, and therefore isn't a stand-alone Python program as described above. Other than that, there is no difference between the Python code that goes into a program file or an IPython notebook.

Modules

Most of the functionality in Python is provided by *modules*. The Python Standard Library is a large collection of modules that provides *cross-platform* implementations of common facilities such as access to the operating system, file I/O, string management, network communication, and much more.

References

- The Python Language Reference: http://docs.python.org/2/reference/index.html (http://docs.python.org/2/reference/index.html)
- The Python Standard Library: http://docs.python.org/2/library/ (http://docs.python.org/2/library/)

To use a module in a Python program it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

In [1]:
```
import math
```

This includes the whole module and makes it available for use later in the program. For example, we can do:

In [3]:
```python
import math

x = math.sin(2 * math.pi)

print(x)
```

```
-2.4492935982947064e-16
```

Alternatively, we can chose to import all symbols (functions and variables) in a module to the current namespace (so that we don't need to use the prefix " `math.` " every time we use something from the `math` module:

In [8]:
```python
from math import *

x = cos(2 * pi)

print(x)
```

```
1.0
```

This pattern can be very convenient, but in large programs that include many modules it is often a good idea to keep the symbols from each module in their own namespaces, by using the `import math` pattern. This would elminate potentially confusing problems with name space collisions.

As a third alternative, we can chose to import only a few selected symbols from a module by explicitly listing which ones we want to import instead of using the wildcard character `*` :

In [9]:
```python
from math import cos, pi

x = cos(2 * pi)

print(x)
```

```
1.0
```

Finally, one standard practice in the scientific community using Python is to shorten namespaces. This usually works my specifiying two characters or three characters *nicknames*.

In [10]:
```python
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

> In many code examples you may find on Internet, the author is using these de facto standard nicknames.

Looking at what a module contains, and its documentation

Once a module is imported, we can list the symbols it provides using the `dir` function:

In [11]:
```python
import math

print(dir(math))
```

```
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'ata
n', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc',
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radia
ns', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

And using the function `help` we can get a description of each function (almost .. not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

In [12]:
```python
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.
```

In [13]:
```python
log(10)
```

Out[13]:
```
2.302585092994046
```

In [14]:
```python
log(10, 2)
```

Out[14]:
```
3.3219280948873626
```

We can also use the `help` function directly on modules: Try

```
help(math)
```

Some very useful modules form the Python standard library are `os` , `sys` , `math` , `shutil` , `re` , `subprocess` , `multiprocessing` , `threading` .

A complete lists of standard modules for Python 2 and Python 3 are available at http://docs.python.org/2/library/ (http://docs.python.org/2/library/) and http://docs.python.org/3/library/ (http://docs.python.org/3/library/), respectively.

Variables and types

Symbol names

Variable names in Python can contain alphanumerical characters `a-z` , `A-Z` , `0-9` and some special characters such as `_` . Normal variable names must start with a letter.

By convension, variable names start with a lower-case letter, and Class names start with a capital letter.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Note: Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. But being a keyword, it cannot be used as a variable name.

Assignment

The assignment operator in Python is `=`. Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one.

Assigning a value to a new variable creates the variable:

In [15]:
```
# variable assignments
x = 1.0
my_variable = 12.2
```

Although not explicitly specified, a variable do have a type associated with it. The type is derived form the value it was assigned.

In [16]:
```
type(x)
```

Out[16]:
```
float
```

If we assign a new value to a variable, its type can change.

In [17]:
```
x = 1
```

In [18]:
```
type(x)
```

Out[18]:
```
int
```

If we try to use a variable that has not yet been defined we get an `NameError`:

In [19]:
```
print(y)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-36b2093251cd> in <module>()
----> 1 print(y)

NameError: name 'y' is not defined
```

Fundamental types

In [20]:
```
# integers
x = 1
type(x)
```

Out[20]:
```
int
```

In [21]:
```
# float
x = 1.0
type(x)
```

Out[21]:
```
float
```

In [22]:
```
# boolean
b1 = True
b2 = False

type(b1)
```

Out[22]:
```
bool
```

In [23]:
```
# complex numbers: note the use of `j` to specify the imaginary part
x = 1.0 - 1.0j
type(x)
```

Out[23]:
```
complex
```

In [24]:
```
print(x)
```

```
(1-1j)
```

In [25]:
```
print(x.real, x.imag)
```

```
(1.0, -1.0)
```

Type utility functions

The module `types` contains a number of type name definitions that can be used to test if variables are of certain types:

In [26]:
```
import types

# print all types defined in the `types` module
print(dir(types))
```

```
['BooleanType', 'BufferType', 'BuiltinFunctionType', 'BuiltinMethodType', 'ClassType', 'C
odeType', 'ComplexType', 'DictProxyType', 'DictType', 'DictionaryType', 'EllipsisType',
'FileType', 'FloatType', 'FrameType', 'FunctionType', 'GeneratorType', 'GetSetDescriptorT
ype', 'InstanceType', 'IntType', 'LambdaType', 'ListType', 'LongType', 'MemberDescriptorT
ype', 'MethodType', 'ModuleType', 'NoneType', 'NotImplementedType', 'ObjectType', 'SliceT
ype', 'StringType', 'StringTypes', 'TracebackType', 'TupleType', 'TypeType', 'UnboundMeth
odType', 'UnicodeType', 'XRangeType', '__builtins__', '__doc__', '__file__', '__name__',
'__package__']
```

In [27]:

```python
x = 1.0

# check if the variable x is a float
type(x) is float
```

Out[27]:

```
True
```

In [28]:

```python
# check if the variable x is an int
type(x) is int
```

Out[28]:

```
False
```

We can also use the `isinstance` method for testing types of variables:

In [29]:

```python
isinstance(x, float)
```

Out[29]:

```
True
```

Type casting

In [30]:

```python
x = 1.5

print(x, type(x))
```

```
(1.5, <type 'float'>)
```

In [31]:

```python
x = int(x)

print(x, type(x))
```

```
(1, <type 'int'>)
```

In [32]:

```python
z = complex(x)

print(z, type(z))
```

```
((1+0j), <type 'complex'>)
```

In [33]:

```python
x = float(z)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-e719cc7b3e96> in <module>()
----> 1 x = float(z)

TypeError: can't convert complex to float
```

Complex variables cannot be cast to floats or integers. We need to use `z.real` or `z.imag` to extract the part of the complex number we want:

In [34]:
```
y = bool(z.real)

print(z.real, " -> ", y, type(y))

y = bool(z.imag)

print(z.imag, " -> ", y, type(y))
```

```
(1.0, ' -> ', True, <type 'bool'>)
(0.0, ' -> ', False, <type 'bool'>)
```

Operators and comparisons

Most operators and comparisons in Python work as one would expect:

- Arithmetic operators `+`, `-`, `*`, `/`, `//` (integer division), '**' power

In [35]:
```
1 + 2, 1 - 2, 1 * 2, 1 / 2
```

Out[35]:
```
(3, -1, 2, 0)
```

In [36]:
```
1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
```

Out[36]:
```
(3.0, -1.0, 2.0, 0.5)
```

In [37]:
```
# Integer division of float numbers
3.0 // 2.0
```

Out[37]:
```
1.0
```

In [38]:
```
# Note! The power operators in python isn't ^, but **
2 ** 2
```

Out[38]:
```
4
```

- The boolean operators are spelled out as words `and`, `not`, `or`.

In [39]:
```
True and False
```

Out[39]:
```
False
```

In [40]:
```
not False
```

Out[40]:
```
True
```

In [41]:
```
True or False
```

Out[41]:
```
True
```

- Comparison operators `>`, `<`, `>=` (greater or equal), `<=` (less or equal), `==` equality, `is` identical.

In [42]:
```
2 > 1, 2 < 1
```

Out[42]:
```
(True, False)
```

In [43]:
```
2 > 2, 2 < 2
```

Out[43]:
```
(False, False)
```

In [44]:
```
2 >= 2, 2 <= 2
```

Out[44]:
```
(True, True)
```

In [45]:
```
# equality
[1,2] == [1,2]
```

Out[45]:
```
True
```

In [46]:
```
# objects identical?
l1 = l2 = [1,2]

l1 is l2
```

Out[46]:
```
True
```

Compound types: Strings, List and dictionaries

Strings

Strings are the variable type that is used for storing text messages.

In [47]:
```
s = "Hello world"
type(s)
```

Out[47]:
```
str
```

In [48]:
```
# length of the string: the number of characters
len(s)
```

Out[48]:
```
11
```

In [49]:
```
# replace a substring in a string with somethign else
s2 = s.replace("world", "test")
print(s2)
```

```
Hello test
```

We can index a character in a string using `[]` :

In [50]:
```
s[0]
```

Out[50]:
```
'H'
```

> **Heads up MATLAB users:** Indexing start at 0!

We can extract a part of a string using the syntax `[start:stop]`, which extracts characters between index `start` and `stop`:

In [51]:
```
s[0:5]
```

Out[51]:
```
'Hello'
```

If we omit either (or both) of `start` or `stop` from `[start:stop]`, the default is the beginning and the end of the string, respectively:

In [52]:
```
s[:5]
```

Out[52]:
```
'Hello'
```

In [53]:
```
s[6:]
```

Out[53]:
```
'world'
```

In [54]:
```
s[:]
```

Out[54]:
```
'Hello world'
```

We can also define the step size using the syntax `[start:end:step]` (the default value for `step` is 1, as we saw above):

In [55]:
```
s[::1]
```

Out[55]:
```
'Hello world'
```

In [56]:
```
s[::2]
```

Out[56]:
```
'Hlowrd'
```

This technique is called *slicing*. Read more about the syntax here: [http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice (http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice)](http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice)

Python has a very rich set of functions for text processing. See for example [http://docs.python.org/2/library/string.html (http://docs.python.org/2/library/string.html)](http://docs.python.org/2/library/string.html) for more information.

String formatting examples

In [57]:
```
print("str1", "str2", "str3")  # The print statement concatenates strings with a space
```

```
('str1', 'str2', 'str3')
```

In [58]:
```
print("str1", 1.0, False, -1j)  # The print statements converts all arguments to strings
```

```
('str1', 1.0, False, -1j)
```

In [59]:
```
print("str1" + "str2" + "str3") # strings added with + are concatenated without space
```

```
str1str2str3
```

In [60]:
```
print("value = %f" % 1.0)       # we can use C-style string formatting
```

```
value = 1.000000
```

In [61]:
```
# this formatting creates a string
s2 = "value1 = %.2f. value2 = %d" % (3.1415, 1.5)

print(s2)
```

```
value1 = 3.14. value2 = 1
```

In [62]:
```
# alternative, more intuitive way of formatting a string
s3 = 'value1 = {0}, value2 = {1}'.format(3.1415, 1.5)

print(s3)
```

```
value1 = 3.1415, value2 = 1.5
```

List

Lists are very similar to strings, except that each element can be of any type.

The syntax for creating lists in Python is `[...]` :

In [63]:
```
l = [1,2,3,4]

print(type(l))
print(l)
```

```
<type 'list'>
[1, 2, 3, 4]
```

We can use the same slicing techniques to manipulate lists as we could use on strings:

In [64]:
```
print(l)

print(l[1:3])

print(l[::2])
```

```
[1, 2, 3, 4]
[2, 3]
[1, 3]
```

Heads up MATLAB users: Indexing starts at 0!

In [65]:
```
l[0]
```

Out[65]:
```
1
```

Elements in a list do not all have to be of the same type:

In [66]:
```
l = [1, 'a', 1.0, 1-1j]

print(l)
```

```
[1, 'a', 1.0, (1-1j)]
```

Python lists can be inhomogeneous and arbitrarily nested:

In [67]:
```
nested_list = [1, [2, [3, [4, [5]]]]]

nested_list
```

Out[67]:
```
[1, [2, [3, [4, [5]]]]]
```

Lists play a very important role in Python, and are for example used in loops and other flow control structures (discussed below). There are number of convenient functions for generating lists of various types, for example the `range` function:

In [68]:
```
start = 10
stop = 30
step = 2

range(start, stop, step)
```

Out[68]:
```
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

In [69]:
```
# in python 3 range generates an interator, which can be converted to a list using 'list(.
# It has no effect in python 2
list(range(start, stop, step))
```

Out[69]:
```
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

In [70]:
```
list(range(-10, 10))
```

Out[70]:
```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [71]:
```
s
```

Out[71]:
```
'Hello world'
```

In [72]:
```
# convert a string to a list by type casting:
s2 = list(s)

s2
```

Out[72]:
```
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

In [73]:
```
# sorting lists
s2.sort()

print(s2)
```

```
[' ', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
```

Adding, inserting, modifying, and removing elements from lists

In [74]:
```
# create a new empty list
l = []

# add an elements using `append`
l.append("A")
l.append("d")
l.append("d")

print(l)
```

```
['A', 'd', 'd']
```

We can modify lists by assigning new values to elements in the list. In technical jargon, lists are *mutable*.

In [75]:
```
l[1] = "p"
l[2] = "p"

print(l)
```

```
['A', 'p', 'p']
```

In [76]:
```
l[1:3] = ["d", "d"]

print(l)
```

```
['A', 'd', 'd']
```

Insert an element at an specific index using `insert`

In [77]:
```
l.insert(0, "i")
l.insert(1, "n")
l.insert(2, "s")
l.insert(3, "e")
l.insert(4, "r")
l.insert(5, "t")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']
```

Remove first element with specific value using 'remove'

In [78]:
```
l.remove("A")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'd', 'd']
```

Remove an element at a specific location using `del` :

In [79]:
```
del l[7]
del l[6]

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't']
```

See `help(list)` for more details, or read the online documentation

Tuples

Tuples are like lists, except that they cannot be modified once created, that is they are *immutable*.

In Python, tuples are created using the syntax `(..., ..., ...)` , or even `..., ...` :

In [80]:
```
point = (10, 20)

print(point, type(point))
```

```
((10, 20), <type 'tuple'>)
```

In [81]:
```
point = 10, 20

print(point, type(point))
```

```
((10, 20), <type 'tuple'>)
```

We can unpack a tuple by assigning it to a comma-separated list of variables:

In [82]:
```
x, y = point

print("x =", x)
print("y =", y)
```

```
('x =', 10)
('y =', 20)
```

If we try to assign a new value to an element in a tuple we get an error:

In [83]:
```
point[0] = 20
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-83-ac1c641a5dca> in <module>()
----> 1 point[0] = 20

TypeError: 'tuple' object does not support item assignment
```

Dictionaries

Dictionaries are also like lists, except that each element is a key-value pair. The syntax for dictionaries is `{key1 : value1, ...}`:

In [84]:
```
params = {"parameter1" : 1.0,
          "parameter2" : 2.0,
          "parameter3" : 3.0,}

print(type(params))
print(params)
```

```
<type 'dict'>
{'parameter1': 1.0, 'parameter3': 3.0, 'parameter2': 2.0}
```

In [85]:
```
print("parameter1 = " + str(params["parameter1"]))
print("parameter2 = " + str(params["parameter2"]))
print("parameter3 = " + str(params["parameter3"]))
```

```
parameter1 = 1.0
parameter2 = 2.0
parameter3 = 3.0
```

In [86]:
```python
params["parameter1"] = "A"
params["parameter2"] = "B"

# add a new entry
params["parameter4"] = "D"

print("parameter1 = " + str(params["parameter1"]))
print("parameter2 = " + str(params["parameter2"]))
print("parameter3 = " + str(params["parameter3"]))
print("parameter4 = " + str(params["parameter4"]))
```

```
parameter1 = A
parameter2 = B
parameter3 = 3.0
parameter4 = D
```

Control Flow

Conditional statements: if, elif, else

The Python syntax for conditional execution of code use the keywords `if` , `elif` (else if), `else` :

In [87]:
```python
statement1 = False
statement2 = False

if statement1:
    print("statement1 is True")

elif statement2:
    print("statement2 is True")

else:
    print("statement1 and statement2 are False")
```

```
statement1 and statement2 are False
```

For the first time, here we encounted a peculiar and unusual aspect of the Python programming language: Program blocks are defined by their indentation level.

Compare to the equivalent C code:

```c
if (statement1)
{
    printf("statement1 is True\n");
}
else if (statement2)
{
    printf("statement2 is True\n");
}
else
{
    printf("statement1 and statement2 are False\n");
}
```

In C blocks are defined by the enclosing curly brakets `{` and `}`. And the level of indentation (white space before the code statements) does not matter (completely optional).

But in Python, the extent of a code block is defined by the indentation level (usually a tab or say four white spaces). This means that we have to be careful to indent our code correctly, or else we will get syntax errors.

Examples:

In [88]:
```
statement1 = statement2 = True

if statement1:
    if statement2:
        print("both statement1 and statement2 are True")
```

```
both statement1 and statement2 are True
```

In [90]:
```
# Bad indentation!
if statement1:
    if statement2:
    print("both statement1 and statement2 are True")  # this line is not properly indented
```

```
  File "<ipython-input-90-78979cdecf37>", line 4
    print("both statement1 and statement2 are True")  # this line is not properly indente
d
        ^
IndentationError: expected an indented block
```

In [91]:
```
statement1 = False

if statement1:
    print("printed if statement1 is True")

    print("still inside the if block")
```

In [92]:
```
if statement1:
    print("printed if statement1 is True")

print("now outside the if block")
```

```
now outside the if block
```

Loops

In Python, loops can be programmed in a number of different ways. The most common is the `for` loop, which is used together with iterable objects, such as lists. The basic syntax is:

`for` loops:

In [93]:
```
for x in [1,2,3]:
    print(x)
```

```
1
2
3
```

The `for` loop iterates over the elements of the supplied list, and executes the containing block once for each element. Any kind of list can be used in the `for` loop. For example:

In [94]:
```
for x in range(4): # by default range start at 0
    print(x)
```

```
0
1
2
3
```

Note: `range(4)` does not include 4 !

In [95]:
```
for x in range(-3,3):
    print(x)
```

```
-3
-2
-1
0
1
2
```

In [96]:
```
for word in ["scientific", "computing", "with", "python"]:
    print(word)
```

```
scientific
computing
with
python
```

To iterate over key-value pairs of a dictionary:

In [97]:
```
for key, value in params.items():
    print(key + " = " + str(value))
```

```
parameter4 = D
parameter1 = A
parameter3 = 3.0
parameter2 = B
```

Sometimes it is useful to have access to the indices of the values when iterating over a list. We can use the `enumerate` function for this:

In [98]:
```
for idx, x in enumerate(range(-3,3)):
    print(idx, x)
```

```
(0, -3)
(1, -2)
(2, -1)
(3, 0)
(4, 1)
(5, 2)
```

List comprehensions: Creating lists using `for` loops:

A convenient and compact way to initialize lists:

In [99]:
```
l1 = [x**2 for x in range(0,5)]

print(l1)
```

```
[0, 1, 4, 9, 16]
```

`while` loops:

In [100]:
```
i = 0

while i < 5:
    print(i)

    i = i + 1

print("done")
```

```
0
1
2
3
4
done
```

Note that the `print("done")` statement is not part of the `while` loop body because of the difference in indentation.

Functions

A function in Python is defined using the keyword `def`, followed by a function name, a signature within parentheses `()`, and a colon `:`. The following code, with one additional level of indentation, is the function body.

In [101]:
```
def func0():
    print("test")
```

In [102]:
```python
func0()
```

```
test
```

Optionally, but highly recommended, we can define a so called "docstring", which is a description of the functions purpose and behaivor. The docstring should follow directly after the function definition, before the code in the function body.

In [103]:
```python
def func1(s):
    """
    Print a string 's' and tell how many characters it has
    """

    print(s + " has " + str(len(s)) + " characters")
```

In [104]:
```python
help(func1)
```

```
Help on function func1 in module __main__:

func1(s)
    Print a string 's' and tell how many characters it has
```

In [105]:
```python
func1("test")
```

```
test has 4 characters
```

Functions that returns a value use the `return` keyword:

In [106]:
```python
def square(x):
    """
    Return the square of x.
    """
    return x ** 2
```

In [107]:
```python
square(4)
```

Out[107]:
```
16
```

We can return multiple values from a function using tuples (see above):

In [108]:
```python
def powers(x):
    """
    Return a few powers of x.
    """
    return x ** 2, x ** 3, x ** 4
```

In [109]:
```python
powers(3)
```

Out[109]:
```
(9, 27, 81)
```

In [110]:
```
x2, x3, x4 = powers(3)

print(x3)
```

```
27
```

Default argument and keyword arguments

In a definition of a function, we can give default values to the arguments the function takes:

In [111]:
```
def myfunc(x, p=2, debug=False):
    if debug:
        print("evaluating myfunc for x = " + str(x) + " using exponent p = " + str(p))
    return x**p
```

If we don't provide a value of the `debug` argument when calling the the function `myfunc` it defaults to the value provided in the function definition:

In [112]:
```
myfunc(5)
```

Out[112]:
```
25
```

In [113]:
```
myfunc(5, debug=True)
```

```
evaluating myfunc for x = 5 using exponent p = 2
```

Out[113]:
```
25
```

If we explicitly list the name of the arguments in the function calls, they do not need to come in the same order as in the function definition. This is called *keyword* arguments, and is often very useful in functions that takes a lot of optional arguments.

In [114]:
```
myfunc(p=3, debug=True, x=7)
```

```
evaluating myfunc for x = 7 using exponent p = 3
```

Out[114]:
```
343
```

Unnamed functions (lambda function)

In Python we can also create unnamed functions, using the `lambda` keyword:

In [115]:
```
f1 = lambda x: x**2

# is equivalent to

def f2(x):
    return x**2
```

In [116]:
```
f1(2), f2(2)
```

Out[116]:
```
(4, 4)
```

This technique is useful for example when we want to pass a simple function as an argument to another function, like this:

In [117]:
```
# map is a built-in python function
map(lambda x: x**2, range(-3,4))
```

Out[117]:
```
[9, 4, 1, 0, 1, 4, 9]
```

In [118]:
```
# in python 3 we can use `list(...)` to convert the iterator to an explicit list
list(map(lambda x: x**2, range(-3,4)))
```

Out[118]:
```
[9, 4, 1, 0, 1, 4, 9]
```

Further reading

- http://www.python.org (http://www.python.org) - The official web page of the Python programming language.
- http://www.python.org/dev/peps/pep-0008 (http://www.python.org/dev/peps/pep-0008) - Style guide for Python programming. Highly recommended.
- http://www.greenteapress.com/thinkpython/ (http://www.greenteapress.com/thinkpython/) - A free book on Python programming.
- Python Essential Reference (http://www.amazon.com/Python-Essential-Reference-4th-Edition/dp/0672329786) - A good reference book on Python programming.

Versions

In [132]:
```
%load_ext version_information

%version_information
```

Out[132]:

Software	Version
Python	2.7.8 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	3.0.0
OS	Linux 3.13.0 45 generic x86_64 with debian jessie sid
	Mon Apr 13 12:00:31 2015 CEST

styling courtesy of http://lorenabarba.com/ (http://lorenabarba.com/)

In [1]:

```
from IPython.core.display import HTML
def css_styling():
    styles = open("./style/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[1]: