Lecture-1-Introduction-to-Python-Programming

September 24, 2014

1 Introduction to Python programming

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

The latest version of this IPython notebook lecture is available at `http://github.com/jrjohansson/scientific-python-lectures`.

The other notebooks in this lecture series are indexed at `http://jrjohansson.github.com`.

1.1 Python program files

- Python code is usually stored in text files with the file ending "`.py`":

  ```
  myprogram.py
  ```

- Every line in a Python program file is assumed to be a Python statement, or part thereof.
  - The only exception is comment lines, which start with the character `#` (optionally preceded by an arbitrary number of white-space characters, i.e., tabs or spaces). Comment lines are usually ignored by the Python interpreter.
- To run our Python program from the command line we use:

  ```
  $ python myprogram.py
  ```

- On UNIX systems it is common to define the path to the interpreter on the first line of the program (note that this is a comment line as far as the Python interpreter is concerned):

  ```
  #!/usr/bin/env python
  ```

If we do, and if we additionally set the file script to be executable, we can run the program like this:

```
$ myprogram.py
```

1.1.1 Example:

In [1]:
```
ls scripts/hello-world*.py
```

```
Invalid switch - "hello-world*.py".
```

In [2]:
```
cat scripts/hello-world.py
```

```
  File "<ipython-input-2-5bdf12dccc10>", line 1
    cat scripts/hello-world.py
              ^
SyntaxError: invalid syntax
```

In [3]: !python scripts/hello-world.py

```
python: can’t open file ’scripts/hello-world.py’: [Errno 2] No such file or directory
```

1.1.2 Character encoding

The standard character encoding is ASCII, but we can use any other encoding, for example UTF-8. To specify that UTF-8 is used we include the special line

```
# -*- coding: UTF-8 -*-
```

at the top of the file.

In [4]: cat scripts/hello-world-in-swedish.py

```
      File "<ipython-input-4-e59e79ffa71d>", line 1
    cat scripts/hello-world-in-swedish.py
                  ^
SyntaxError: invalid syntax
```

In [5]: !python scripts/hello-world-in-swedish.py

```
python: can’t open file ’scripts/hello-world-in-swedish.py’: [Errno 2] No such file or directory
```

Other than these two *optional* lines in the beginning of a Python code file, no additional code is required for initializing a program.

1.2 IPython 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 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.

1.3 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.

1.3.1 References

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

1.4 SciPy for Numerical Analysis

1.4.1 Evaluation of Special functions

The “scipy.special” contains the definitions and code for useful functions

1.4.2 Higher Mathematics: the math library

Evaluate the expression $\pi 10^7 \sqrt(90.1)$

In order to use the value of π, we have to first load the math library; the statement from math import * loads all (the * wildcard character tells you that) of the functions from

```
In [6]: import math
        #help(math)
```

```
In [7]: from math import *
        print pi*10**7*sqrt(90.1)
```

```
298203178.477
```

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

1.4.3 four methods for loading a library

When importing libraries into Python, we have four alternatives (math library used as an example):

```
In [8]: from math import sin
        from math import *
        import math
        import math as m
```

Method (1) loads only the sine function, and method (2) loads all of the functions in the math library; the advantage of this method is that it allows us to call a function by its name in the particular library, for example, to calculate the sine of x, we simply type

```
sin(x)
```

Method (3) also loads the entire math library, but now to calculate the sine of x, we must type

```
math.sin(x)
```

The fourth method, is simply allows one to have a shorthand method for addressing the math library; now we need only type

```
m.sin(x)
```

to calculate the sin(x) using the math library. Methods (3) and (4) are the preferred way to load libraries, because they remove all ambiguity as to what library a particular funtion belongs to.

```
In [9]: import math

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

        print(x)
```

```
1.0
```

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 [10]: 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 [11]: from math import cos, pi

         x = cos(2 * pi)

         print(x)
```

```
1.0
```

1.4.4 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 [12]: import math

         print(dir(math))
```

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
```

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 [13]: 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 [14]: log(10)

Out[14]: 2.302585092994046

In [15]: log(10, 2)

Out[15]: 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/ and http://docs.python.org/3/library/, respectively.

1.5 Variables and types

1.5.1 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 convention, 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.

1.5.2 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 [16]: # 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 [17]: type(x)

Out[17]: float
```

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

```
In [18]: x = 1

In [19]: type(x)

Out[19]: int
```

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

```
In [20]: print(y)


        ---------------------------------------------------------------------------
    NameError                                 Traceback (most recent call last)

        <ipython-input-20-36b2093251cd> in <module>()
    ----> 1 print(y)


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

1.5.3 Fundamental types

```
In [21]: # integers
         x = 1
         type(x)
```

```
Out[21]: int
```

```
In [22]: # float
         x = 1.0
         type(x)
```

```
Out[22]: float
```

```
In [23]: # boolean
         b1 = True
         b2 = False

         type(b1)
```

```
Out[23]: bool
```

```
In [24]: # complex numbers: note the use of `j` to specify the imaginary part
         x = 1.0 - 1.0j
         type(x)
```

```
Out[24]: complex
```

```
In [25]: print(x)
```

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

```
In [26]: print(x.real, x.imag)
```

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

1.5.4 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 [27]: import types

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

```
['BooleanType', 'BufferType', 'BuiltinFunctionType', 'BuiltinMethodType', 'ClassType', 'CodeType', 'Comp
```

```
In [28]: x = 1.0

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

```
Out[28]: True
```

```
In [29]: # check if the variable x is an int
         type(x) is int
```

```
Out[29]: False
```

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

```
In [30]: isinstance(x, float)
```

```
Out[30]: True
```

1.5.5 Type casting

```
In [31]: x = 1.5

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

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

```
In [32]: x = int(x)

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

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

```
In [33]: z = complex(x)

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

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

```
In [34]: x = float(z)
```

```
        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-34-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 [35]: 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'>)
```

1.6 Operators and comparisons

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

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

```
In [36]: 1 + 2, 1 - 2, 1 * 2, 1 / 2
```

```
Out[36]: (3, -1, 2, 0)
```

```
In [37]: 1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
```

```
Out[37]: (3.0, -1.0, 2.0, 0.5)
```

```
In [38]: # Integer division of float numbers
         3.0 // 2.0
```

```
Out[38]: 1.0
```

```
In [39]: # Note! The power operators in python isn't ^, but **
         2 ** 2
```

```
Out[39]: 4
```

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

```
In [40]: True and False
```

```
Out[40]: False
```

```
In [41]: not False
```

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

```
In [42]: True or False
```

```
Out[42]: True
```

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

```
In [43]: 2 > 1, 2 < 1
```

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

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

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

```
In [45]: 2 >= 2, 2 <= 2
```

```
Out[45]: (True, True)
```

```
In [46]: # equality
         [1,2] == [1,2]
```

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

```
In [47]: # objects identical?
         l1 = l2 = [1,2]

         l1 is l2
```

```
Out[47]: True
```

1.7 Compound types: Strings, List and dictionaries

1.7.1 Strings

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

```
In [48]: s = "Hello world"
         type(s)
```

```
Out[48]: str
```

```
In [49]: # length of the string: the number of characters
         len(s)
```

```
Out[49]: 11
```

```
In [50]: # 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 [51]: s[0]
```

```
Out[51]: '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 [52]: s[0:5]
```

```
Out[52]: '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 [53]: s[:5]
```

```
Out[53]: 'Hello'
```

```
In [54]: s[6:]
```

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

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

```
Out[55]: '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 [56]: s[::1]
```

```
Out[56]: 'Hello world'
```

```
In [57]: s[::2]
```

```
Out[57]: '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

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

String formatting examples

```
In [58]: print("str1", "str2", "str3")  # The print statement concatenates strings with a space
```

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

```
In [59]: print("str1", 1.0, False, -1j)  # The print statements converts all arguments to strings
```

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

```
In [60]: print("str1" + "str2" + "str3") # strings added with + are concatenated without space
```

```
str1str2str3
```

```
In [61]: print("value = %f" % 1.0)       # we can use C-style string formatting
```

```
value = 1.000000
```

```
In [62]: # this formatting creates a string
         s2 = "value1 = %.2f. value2 = %d" % (3.1415, 1.5)

         print(s2)
```

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

```
In [63]: # 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
```

1.7.2 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 [64]: 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 [65]: 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 [66]: l[0]

Out[66]: 1
```

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

```
In [67]: l = [1, 'a', 1.0, 1-1j]

         print(l)

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

Python lists can be inhomogeneous and arbitrarily nested:

```
In [68]: nested_list = [1, [2, [3, [4, [5]]]]]

         nested_list

Out[68]: [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 [69]: start = 10
         stop = 30
         step = 2

         range(start, stop, step)

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

In [70]: # 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[70]: [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

In [71]: list(range(-10, 10))

Out[71]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [72]: s

Out[72]: 'Hello world'

In [73]: # convert a string to a list by type casting:
         s2 = list(s)

         s2

Out[73]: ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

In [74]: # 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 [75]: # 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 [76]: l[1] = "p"
         l[2] = "p"

         print(l)
```

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

```
In [77]: l[1:3] = ["d", "d"]

         print(l)
```

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

Insert an element at an specific index using `insert`

```
In [78]: 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 [79]: l.remove("A")

         print(l)
```

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

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

```
In [80]: 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

1.7.3 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 [81]: point = (10, 20)

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

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

```
In [82]: 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 [83]: 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 [84]: point[0] = 20
```

```
        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-84-ac1c641a5dca> in <module>()
    ----> 1 point[0] = 20


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

1.7.4 Dictionaries

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

```
In [85]: 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 [86]: 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 [87]: 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
```

1.8 Control Flow

1.8.1 Conditional statements: if, elif, else

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

```
In [88]: 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:

```
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 [89]: 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 indented
            ^
    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
```

1.9 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:

1.9.1 `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)
```

1.9.2 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]
```

1.9.3 `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.

1.10 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]: func0()
```

```
test
```

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

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

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

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

```
Help on function func1 in module __main__:

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

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

```
test has 4 characters
```

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

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

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

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

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

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

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

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

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

          print(x3)
```

```
27
```

1.10.1 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
```

1.10.2 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]
```

1.11 Classes

Classes are the key features of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object.

In Python a class can contain *attributes* (variables) and *methods* (functions).

A class is defined almost like a function, but using the `class` keyword, and the class definition usually contains a number of class method definitions (a function in a class).

- Each class method should have an argument `self` as it first argument. This object is a self-reference.
- Some class method names have special meaning, for example:
  - `__init__`: The name of the method that is invoked when the object is first created.
  - `__str__` : A method that is invoked when a simple string representation of the class is needed, as for example when printed.
  - There are many more, see http://docs.python.org/2/reference/datamodel.html#special-method-names

```
In [119]: class Point:
              """
              Simple class for representing a point in a Cartesian coordinate system.
              """

              def __init__(self, x, y):
                  """
                  Create a new Point at x, y.
                  """
                  self.x = x
                  self.y = y

              def translate(self, dx, dy):
                  """
                  Translate the point by dx and dy in the x and y direction.
                  """
                  self.x += dx
                  self.y += dy

              def __str__(self):
                  return("Point at [%f, %f]" % (self.x, self.y))
```

To create a new instance of a class:

```
In [120]: p1 = Point(0, 0) # this will invoke the __init__ method in the Point class

          print(p1)         # this will invoke the __str__ method

Point at [0.000000, 0.000000]
```

To invoke a class method in the class instance `p`:

```
In [121]: p2 = Point(1, 1)

          p1.translate(0.25, 1.5)

          print(p1)
          print(p2)

Point at [0.250000, 1.500000]
Point at [1.000000, 1.000000]
```

Note that calling class methods can modifiy the state of that particular class instance, but does not effect other class instances or any global variables.

That is one of the nice things about object-oriented design: code such as functions and related variables are grouped in separate and independent entities.

1.12 Modules

One of the most important concepts in good programming is to reuse code and avoid repetitions.

The idea is to write functions and classes with a well-defined purpose and scope, and reuse these instead of repeating similar code in different part of a program (modular programming). The result is usually that readability and maintainability of a program is greatly improved. What this means in practice is that our programs have fewer bugs, are easier to extend and debug/troubleshoot.

Python supports modular programming at different levels. Functions and classes are examples of tools for low-level modular programming. Python modules are a higher-level modular programming construct, where we can collect related variables, functions and classes in a module. A python module is defined in a python file (with file-ending `.py`), and it can be made accessible to other Python modules and programs using the `import` statement.

Consider the following example: the file `mymodule.py` contains simple example implementations of a variable, function and a class:

```
In [122]: %%file mymodule.py
          """
          Example of a python module. Contains a variable called my_variable,
          a function called my_function, and a class called MyClass.
          """

          my_variable = 0

          def my_function():
              """
              Example function
              """
              return my_variable

          class MyClass:
              """
              Example class.
              """

              def __init__(self):
                  self.variable = my_variable

              def set_variable(self, new_value):
                  """
                  Set self.variable to a new value
                  """
                  self.variable = new_value

              def get_variable(self):
                  return self.variable
```

```
Writing mymodule.py
```

We can import the module `mymodule` into our Python program using `import`:

```
In [123]: import mymodule
```

Use `help(module)` to get a summary of what the module provides:

```
In [124]: #help(mymodule)

In [125]: mymodule.my_variable

Out[125]: 0

In [126]: mymodule.my_function()

Out[126]: 0

In [127]: my_class = mymodule.MyClass()
          my_class.set_variable(10)
          my_class.get_variable()
```

```
Out[127]: 10
```

If we make changes to the code in `mymodule.py`, we need to reload it using `reload`:

```
In [128]: reload(mymodule)  # works only in python 2
```

```
Out[128]: <module 'mymodule' from 'mymodule.pyc'>
```

1.13 Exceptions

In Python errors are managed with a special language construct called "Exceptions". When errors occur exceptions can be raised, which interrupts the normal program flow and fallback to somewhere else in the code where the closest try-except statement is defined.

To generate an exception we can use the `raise` statement, which takes an argument that must be an instance of the class `BaseExpection` or a class derived from it.

```
In [129]: raise Exception("description of the error")
```

```
        ---------------------------------------------------------------------------
    Exception                                 Traceback (most recent call last)

        <ipython-input-129-8f47ba831d5a> in <module>()
    ----> 1 raise Exception("description of the error")


        Exception: description of the error
```

A typical use of exceptions is to abort functions when some error condition occurs, for example:

```
def my_function(arguments):

    if not verify(arguments):
        raise Expection("Invalid arguments")

    # rest of the code goes here
```

To gracefully catch errors that are generated by functions and class methods, or by the Python interpreter itself, use the `try` and `except` statements:

```
try:
    # normal code goes here
except:
    # code for error handling goes here
    # this code is not executed unless the code
    # above generated an error
```

For example:

```
In [130]: try:
              print("test")
              # generate an error: the variable test is not defined
              print(test)
          except:
              print("Caught an expection")
```

```
test
Caught an expection
```

To get information about the error, we can access the `Exception` class instance that describes the exception by using for example:

```
except Exception as e:
```

```
In [131]: try:
              print("test")
              # generate an error: the variable test is not defined
              print(test)
          except Exception as e:
              print("Caught an exception:" + str(e))
```

```
test
Caught an exception:name 'test' is not defined
```

1.14 Further reading

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

1.15 Versions

```
In [132]: %load_ext version_information

          %version_information
```

```
    ---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)

    <ipython-input-132-14367795162a> in <module>()
----> 1 get_ipython().magic(u'load_ext version_information')
      2
      3 get_ipython().magic(u'version_information')


    C:\Users\Elias\Anaconda\lib\site-packages\IPython\core\interactiveshell.pyc in magic(self, arg_s
   2203         magic_name, _, magic_arg_s = arg_s.partition(' ')
   2204         magic_name = magic_name.lstrip(prefilter.ESC_MAGIC)
-> 2205         return self.run_line_magic(magic_name, magic_arg_s)
   2206
   2207     #-------------------------------------------------------------------------


    C:\Users\Elias\Anaconda\lib\site-packages\IPython\core\interactiveshell.pyc in run_line_magic(se
   2124                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2125             with self.builtin_trap:
-> 2126                 result = fn(*args,**kwargs)
   2127             return result
```

```
    2128


     C:\Users\Elias\Anaconda\lib\site-packages\IPython\core\magics\extension.pyc in load_ext(self, mo


     C:\Users\Elias\Anaconda\lib\site-packages\IPython\core\magic.pyc in <lambda>(f, *a, **k)
     191     # but it's overkill for just that one bit of state.
     192     def magic_deco(arg):
 --> 193         call = lambda f, *a, **k: f(*a, **k)
     194
     195         if callable(arg):


     C:\Users\Elias\Anaconda\lib\site-packages\IPython\core\magics\extension.pyc in load_ext(self, mo
      61         if not module_str:
      62             raise UsageError('Missing module name.')
 ---> 63         res = self.shell.extension_manager.load_extension(module_str)
      64
      65         if res == 'already loaded':


     C:\Users\Elias\Anaconda\lib\site-packages\IPython\core\extensions.pyc in load_extension(self, mo
      96             if module_str not in sys.modules:
      97                 with prepended_to_syspath(self.ipython_extension_dir):
 ---> 98                     __import__(module_str)
      99             mod = sys.modules[module_str]
     100             if self._call_load_ipython_extension(mod):


     ImportError: No module named version_information
```

```
In []: #help(scipy.special)
       import scipy.special
```

```
In []: import numpy
       a=scipy.special.exp10(-16)
       print a
       numpy.log(1+a)
```

```
In []: P1=numpy.poly1d([1,0,1]) # defines polynomial from its coefficients
```

```
In []: print P1
```

```
In []: print P1.r; print P1.o; P1.deriv() # roots,order,derivative
```

```
In []: P2=numpy.poly1d( (1, 1, 1), True) #define poly specifying roots
```

```
In []: print P2
```

```
In []: P1( numpy.arange(10) ) # eval.uate at 0,1, ... ,9
```

There are also a handful of routines associated to polynomials - roots (to compute zeros), polyder (to compute derivatives), polyint (to compute integrals), polyadd (to add polynomials), polysub (to subtract polynomials), polymul (to multiply polynomials), polydiv (to perform polynomial division), polyval (to evaluate polynomials), and polyfit (to compute the best fit polynomial of certain order for two given arrays of data).

The usual binary operators +, - , *, and / perform the corresponding operations with polynomials. In addition, once a polynomial is created, any list of values that interacts with them is inunediately casted to a polynomial. Therefore, the following four commands are equivalent:

- numpy.polyadd(P1, numpy.poly1d([2,1]))
- numpy.polyadd (P1, [2, 1])
- P1+ numpy.poly1d([2,1])
- P1 + [2, 1]

```
In []: P1/ (2, 1) # quotient and remainder
```

$\frac{x^2+1}{2x+1} = (x/2 - 1/4) + \frac{5/4}{2x+1}$

```
In []: P0=numpy.poly1d([1.25])
       print P0
```

1.15.1 Interpolation and regression

Interpolation is a basic method in numerical computation that is obtained from a discrete set of data points, some higher order structure thatt contains the previous data. The best known example is the interpolation of a sequence of points (x_k, y_k) in a plane to obtain a curve that goes through all the points in the order dictated by the sequence. If the points in the previous sequence are in the tight position and order, it is possible to find a univariate function, y =f(x) for which y_k = f(x_k). It is often reasonable to request this interpolating ftmction to be a polynomial, or a rational function, or a more complex functional object. Interpolation is also possible in higher dimensions. The objective of the *scipy.interpolate* module is precisely to offer a complete set of optimally coded applications to address this problem in different settings.

```
In []: import scipy.interpolate
       import matplotlib.pyplot
       %matplotlib inline
       x=numpy.linspace (-1, 1, 10); xn=numpy.linspace ( -1, 1, 1000)
       y=numpy.sin (x)
       polynomial=scipy.interpolate.lagrange(x, numpy.sin(x))
       matplotlib.pyplot.plot(xn,polynomial (xn) ,x,y, 'or')
```

More advanced one-dimensional interpolation is possible with piecewise polynomials (PiecewisePolynomial). This allows control over the degrees of different pieces, as well as the delivatives at their intersections. Other interpolation options in the *scipy.interpolate* module are PCHIP monotonic cubic interpolation (pchip), or even univariate splines (InterpolatedUnivariateSpline).

```
InterpolatedUnivariateSpline(x, y, w=None, bbox=[None, None], k=3)
```

The arrays x and y contain the dependent and independent data, respectively, The array w contains positive weights for spline fitting. The two-sequence bbox specifies the boundary of the approximation interval. The last option indicates the degree of the smoothing polynomials (k).

For instance, we desire to interpolate five points as shown in the following session. These points are ordered by shictly increasing x values. We need to perform this interpolation with four cubic polynomials (one for every two consecutive points), in such a way that at least the first derivative of each two consecutive pieces agree on their intersection. We will proceed as follows:

```
In []: x=numpy.arange(5); y=numpy.sin(x)
       xn=numpy.linspace(0,4,40)
       interp=scipy.interpolate.InterpolatedUnivariateSpline(x, y)
       matplotlib.pyplot.plot (x, y, '.', xn, interp(xn))
```

SciPy excels at interpolating in two-dimensional grids as well. It performs well with simple piecewise polynomials (LinearNDinterpolator), with piecewise constants (NearestNDinterpolator), or with more advanced splines (BivariateSpline). It is capable of carrying spline interpolation on rectangular meshes in a plane (RectBivariateSpline) or on the surface of a sphere (RectSphereBivariateSpline). For unstructured data, besides basic BivariateSpline, it is capable of computing smooth approximations (SmoothBivariateSpline) or more involved weighted least-squares splines (LSQBivariateSpline).

The following code creates a 10 x 10 grid of uniformly spaced points in the square from (0, 0) to (9, 9), and evaluates !the function, sin (x) * cos (y) on them. We use these points to create a BivariateSpline, and evaluate the resulting ftmcltion on the square for all values.

```
In []: from mpl_toolkits.mplot3d import Axes3D
       from mpl_toolkits.mplot3d import Axes3D
       import matplotlib.pyplot as plt
       import numpy as np
       x=y=numpy.arange (10)
       f=(lambda i,j: numpy.sin(i)*numpy.cos(j)) # function to interpolate
       A=numpy.fromfunction(f, (10,10)) #generate samples
       spline=scipy.interpolate.RectBivariateSpline(x,y,A)
       fig=matplotlib.pyplot.figure()
       #subplot=fig.add_subplot(111, projection='3d')
       subplot = fig.add_subplot(1, 1, 1, projection='3d')
       X = np.linspace(0, 9, 100)
       xlen = len(X)
       Y = np.linspace(0, 9, 100)
       ylen = len(Y)
       xx,yy=numpy.meshgrid(X, Y) # larger grid for plotting
       A=spline (numpy.linspace ( 0, 9., 100), numpy.linspace ( 0, 9, 100))
       subplot.plot_surface(xx ,yy ,A)

In []: fig.add_subplot?

In []: subplot.plot_surface?

In []: """
       Demonstrate the mixing of 2d and 3d subplots
       """
       from mpl_toolkits.mplot3d import Axes3D
       import matplotlib.pyplot as plt
       import numpy as np

       def f(t):
           s1 = np.cos(2*np.pi*t)
           e1 = np.exp(-t)
           return np.multiply(s1,e1)


       ################
       # First subplot
       ################
       t1 = np.arange(0.0, 5.0, 0.1)
```

```
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

# Twice as tall as it is wide.
fig = plt.figure(figsize=plt.figaspect(2.))
fig.suptitle('A tale of 2 subplots')
ax = fig.add_subplot(2, 1, 1)
l = ax.plot(t1, f(t1), 'bo',
            t2, f(t2), 'k--', markerfacecolor='green')
ax.grid(True)
ax.set_ylabel('Damped oscillation')


#################
# Second subplot
#################
ax = fig.add_subplot(2, 1, 2, projection='3d')
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
        linewidth=0, antialiased=False)

ax.set_zlim3d(-1, 1)

plt.show()
```

In []:
```
cd
```

In []:
```
cd C:\Users\Elias\Documents\IPython Notebooks
```

In []:
```
!ipython nbconvert --to latex Lecture-1-Introduction-to-Python-Programming.ipynb
```

In []: