Lighting-fast introduction to Python

Szymon Stoma (from Falko Krause's manuscript)

October 24, 2010

1 Getting Started

1.1 Interactive Mode

One of the features that makes Python easy to learn is its ability to function as a command line interpreter. In the command line interpreter (or Python interactive shell), you can type in a command and Python will instantly respond with the result. You can invoke the interactive shell by calling Python without any arguments.

```
$ python
Python 2.5.2 (r252:60911, May 7 2008, 15:19:09)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for ...
 ...more information.
>>>
```

After starting up, Python prompts you for the next command with the primary prompt `>>>`.
For continuation lines, it prompts with the secondary prompt `...`.

```
>>> myflag = 1
>>> if myflag:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

If you read the next Section (2) you will understand the example in detail.

IPython If you are serious about learning Python, you will have to get IPython. IPython is an extension of the interactive shell, it "is an interactive shell for the Python programming language that offers [...] additional shell syntax, code highlighting, and tab completion." (Wikipedia). This means that your commands will be displayed in nice colors (unfortunately not in this script), that you can press the tab key or the "up" key to autocomplete e.g. variable names (this will be referred to as `<TAB>` and `<UP>`) and that you will be able to access the Python documentation instantly.

2 The Basics

Before we start to actually use Python, two important concepts should be explained.

Duck Typing
This style of dynamic typing (assigning a datatype to a variable) is widespread among current popular interpreted languages. Its motto is "If it walks like a duck and quacks like a duck, I would call it a duck.", in Python this means that you can advise an integer number to a variable and the variable will be of type integer (without ever declaring this fact explicitly).

Indentation
Indentation determines the context of commands. This makes Python highly readable and rids it of most of the "swearword" symbols ($♯)}( ) that other languages depend on. The actual use will be demonstrated in this tutorial many times.

2.1 Datatypes

Numbers If you start the Python interactive shell (or IPython), you can use it as a calculator. Just type some integer numbers (`int`) with some common mathematical symbols.

```
>>> 2+2
4
>>> (50-5*6)/4
5
```

If you want to "save" your numbers you can assign them to a variable using the `=` sign. Now you can reuse them for complicated calculations like the one below.

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Of course integer numbers are not enough. In science, we need floating point numbers (`float`).

```
>>> 3 * 3.75 / 1.5
7.5
```

You can convert an `int` into a `float` - just like that.

```
>>> float(width)
20.0
```

This kind of type casting works for most datatypes in Python (!). Python also knows about complex numbers and has functions like rounding (`round()`) etc. built in and ready to use.

Strings String can be expressed in several ways, here is one:

```
>>> 'spam eggs'
'spam eggs'
```

Enclosing a string in single quotes (`'`) will not interpret the contents, this means a newline `'\n'` will return just the characters in the string `\n`. Double quoted strings are interpreted and will convert the newline character(s) into a new line. You could write a string that spans multiple lines like that:

```
>>> hello = "This is a rather long string containing\n\
... several lines of text just as you would do in C.\n\
...     Note that whitespace at the beginning of the line...
 ... is\
...  significant."
>>>
>>> print hello
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is...
 ... significant.
```

At the end of each line a \ declares that the same command continues on the next line. But you could also enclose your string in triple quotes (`'''` or `"""`).

```
>>> hello = '''This is a rather long string containing
... several lines of text just as you would do in C.
...     Note that whitespace at the beginning of the line...
 ... is
... significant.'''
>>> print hello
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is
significant.
```

Concatenating strings is easy.

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
```

In IPython you can see all the string functions by tabbing them

```
In [1]: word = 'Help' + 'A'

In [2]: word
Out[2]: 'HelpA'

In [3]: word.<TAB>
str.__add__          str.__hash__         str....
 ...__subclasses__   str.lower
str.__base__         str.__init__         str....
 ...__weakrefoffset__ str.lstrip
str.__bases__        str.__itemsize__     str.capitalize ...
 ...       str.mro
str.__basicsize__    str.__le__           str.center ...
 ...           str.partition
str.__call__         str.__len__          str.count ...
 ...            str.replace
str.__class__        str.__lt__           str.decode ...
 ...           str.rfind
str.__cmp__          str.__mod__          str.encode ...
 ...           str.rindex
str.__contains__     str.__module__       str.endswith ...
 ...         str.rjust
str.__delattr__      str.__mro__          str.expandtabs ...
 ...       str.rpartition
str.__dict__         str.__mul__          str.find ...
 ...            str.rsplit
str.__dictoffset__   str.__name__         str.index ...
 ...          str.rstrip
str.__doc__          str.__ne__           str.isalnum ...
 ...        str.split
str.__eq__           str.__new__          str.isalpha ...
 ...        str.splitlines
str.__flags__        str.__reduce__       str.isdigit ...
 ...        str.startswith
str.__ge__           str.__reduce_ex__    str.islower ...
 ...        str.strip
str.__getattribute__ str.__repr__         str.isspace ...
 ...        str.swapcase
str.__getitem__      str.__rmod__         str.istitle ...
 ...        str.title
str.__getnewargs__   str.__rmul__         str.isupper ...
 ...        str.translate
str.__getslice__     str.__setattr__      str.join ...
 ...            str.upper
str.__gt__           str.__str__          str.ljust ...
 ...          str.zfill

In [4]: word.upper()
Out[4]: 'HELPA'
```

By the way, if you want to get your last command back, you can press "up" and it will autocomplete your command (even if it you closed and reopened IPython in between)

```
In [4]:wor<UP>
```

Lists Python has a variety of list types
The most basic type is the `tuple`.

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

A normal list is called `list`. This is the closest to what is known as "array" in other programming languages.

```
>>> l = ['spam', 'eggs', 100, 1234]
>>> l
['spam', 'eggs', 100, 1234]
```

A `list` is not very practical if you need to find one of its members. That's why Python has the datatype `set`. The set internally uses a hash function to index its values. In contrast to a `list` the `set` will not store duplicate entries. On a `set` you can use the `in` command to check if a member exists. You can also use functions like `union`, `difference` etc. to create new `sets`.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', '...
 ...orange', 'banana']
>>> s = set(basket)          # create a set without ...
 ...duplicates
>>> s
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in s            # fast membership testing
True
>>> 'crabgrass' in s
False
```

Very similar to a `set` is the Dictionary (`dict`). It contains key / value pairs ({*key:value,key:value*}).

Basically the keys form a `set` that has for each entry a value attached.

```
>>> d = {'jannis': 4098, 'wolf': 4139}
>>> d['guido'] = 4127
>>> d
{'wolf': 4139, 'guido': 4127, 'jannis': 4098}
>>> d['jannis']
4098
```

In the example above values are added/extracted by specifying their key in square brackets.
In `lists` and `tuples`, element positions are the "keys".

```
>>>t[0]
12345
```

And not to forget, `str` is a list type too!
A very convenient way to get subsets from `tuples`, `lists` and `strs` is to specify start and end positions separated by a colon in the square brackets (*list[start:end]*).

```
>>> word = 'WOOT this Python lesson is awesome'
>>> word.split()
['WOOT', 'this', 'Python', 'lesson', 'is', 'awesome']
>>> word[10:17]+word.split()[4]+word[-7:]
'Python is awesome'
```

Leaving start or stop values empty is a shortcut to the very start of the list or respectively the very end of the list. Negative values are subtracted from the length of the list (`-1` is thus the last element of the list).

Other Important Datatypes To express boolean values Python provides the datatype `bool`. Its values are `True` and `False`. Sequences can act as booleans, that is, an empty sequence (e.g. `[]`) acts as `False` and a filled sequence (e.g. `['a','b']`) acts as `True`. The `int 0` is also eqivalent to `False` - all other integers are equivalent to `True`. The same applies to `float`.
The datatype `None` is frequently used to represent the absence of a value. It has only one value: `None`.

2.2 Control Flow

Due to the lack of creativity the introduction to control flow will start with the classic example of the Fibonacci series.

`while` Statements There are many possible implementations of the Fibonacci series, this one uses the `while` statement.

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

The first line shows an example of a *multiple assignment*. The `while` loop on the second line executes *the indented code below* as long as the boolean (`bool`) statement follwing the `while` evaluates to `True`. The comma at the end of the third line will prevent `print` to add a new line every time it is called.

`if` Statements Also `if` statements take a `bool` as input. To create a chain of `if` statements that executes different commands depending on multiple conditions, the `elif` statement (short for "else if") can be used. In the following example, chained "if" statements process the users input. (The user enters the `42` in this example.)

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

`for` Statements Here is one of the strengths of Python. Looping through lists is very simple and intuitive. If you programmed in other languages before that do not have similar "for" loops, you might need a while to adapt to the fact that you can iterate over the items of any sequence (`str`,`list`,`tuple`,`set`) without having to deal with indices.

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...    print x, len(x)
...
cat 3
window 6
defenestrate 12
```

The `range()` Function Generating lists of numbers (e.g. `list` indices) is also easy.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Now you can show the index number of the list entry, if you really need to.

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...    print i, a[i]
...
0 Mary
1 had
2 a
```

```
3 little
4 lamb
```

The example above is rather complicated. In actual source-code, you would write

```
>>> for i,b in enumerate(a):
...     print i, b
...
0 Mary
1 had
2 a
3 little
4 lamb
```

For looping trough dictionaries look into the functions `keys()`, `values()` and `iteritems()`. They are part of the `dict` class. Remember that in IPython, you can easily find them by typing `mydict.<TAB>`.

`break` and `continue` Statements, and `else` Clauses on Loops You can `break` out of the smallest enclosing loop - or just skip to the next iteration of the loop with `continue`. A very convenient feature is that you can execute code in an `else` statement that follows a loop. It is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`).

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:# loop fell through without finding a factor
...         if n==3:
...             continue
...         print n, 'is a prime number'
...
2 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

`pass` Statements The laziest statement is `pass`, it will do nothing. This is very helpful if you write the structure of your code first and fill the actual commands later.

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 10
>>> if x < 0:
...      pass
... elif x == 42:
...      pass #TODO must fill answer to the ultimate ...
  ...question of life, the universe, and everything here ...
  ...later
... else:
...      print 'More'
...
More
```

2.3 Functions

Functions are defined with `def` followed by the function name followed by round brackets that contain the arguments passed to the function. A function can return values by using `return`.

```
>>> def tell_miau(who):
...     return who+" told Miauuuuu"
...
>>> print tell_miau('Jannis')
Jannis told Miauuuuu
```

Default Argument Values and Keyword Arguments You can assign default values to the arguments passed to a function. In addition to that, you can use an argument name as a keyword to pass this specific argument. This is very useful for functions that have many arguments with default values of which you only need to use a few.

```
>>> def tell_compliment(who,person="Falko",reply="Thanks...
  ..."):
...     return who+' told: '+person+" you have beautiful ...
  ...eyes!\n"+person+" replied: "+reply
...
>>> print tell_compliment("Jannis","Eve")
Jannis told: Eva you have beautiful eyes!
Eve replied: Thanks
>>> print tell_compliment("Timo",reply="Eeee?")
Timo told: Falko you have beautiful eyes!
Falko replied: Eeee?
```

What amazed me in Python is that functions are not very different than other datatypes.

```
>>> kmplmnt=tell_compliment
>>> print kmplmnt("Falko")
...
```

Documentation Strings Python has a built in method of documenting your source-code.

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

IPython uses this documentation in a very convenient way

```
In [1]: def my_function():
   ...:     ''' the same here '''
   ...:     pass
   ...:

In [2]: my_function?
Type:           function
Base Class:     <type 'function'>
String Form:    <function my_function at 0x83f4f7c>
Namespace:      Interactive
File:           /home/select/MPG/SBML/semanticSBML/trunk/<...
  ...ipython console>
Definition:     my_function()
Docstring:
    the same here
```

```
In [3]: str?
Type:         type
Base Class:   <type 'type'>
String Form:  <type 'str'>
Namespace:    Python builtin
Docstring:
   str(object) -> string

   Return a nice string representation of the object.
   If the argument is a string, the return value is the...
    ... same object.
```

3 More on Lists

list.append(x)
: Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

list.extend(L)
: Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

list.insert(i, x)
: Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

list.remove(x)
: Remove the first item from the list whose value is `x`. It is an error if there is no such item.

list.pop(i)
: Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list.

list.index(x)
: Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.

list.count(x)
: Return the number of times `x` appears in the list.

list.sort()
: Sort the items of the list, in place.

list.reverse()
: Reverse the elements of the list, in place.

Using Lists as Stacks and Queues is easy with the functions above

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")       # Terry arrives
>>> queue.append("Graham")      # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

List Comprehensions The "old-school" method of manipulating lists in loops is hardly ever used in Python because of its list comprehensions feature. It enables you to manipulate a list on the fly. Once you get used to this feature you will never want to miss it again.

```
>>> freshfruit = [' banana', ' loganberry ', 'passion ...
 ...fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
```

4 Modules

A file containing Python source-code is called a module.
Before we continue you should know that at this point you will need a text editor. You could use the default text editor of your operating system but then you will miss alot of nice features like text highlighting, tab-completion, smart-indentation and syntax checking. For the beginning you can try IDLE (installs by default with Python on Widows) and later on switch to a more advanced IDE (Integrated development environment).
If you write the module `fibo.py` (contents below)

```
"""
Fibonacci numbers module
"""

def fib(n):   # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

you can import it into the interactive shell (or another module) by calling `import` *modulename* (without the .py extension) if the file is in the same folder or Pythons search path. By adding a `.` to the module name you can access the functions of the module.

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you change your module you will have to reload it to see the changes in the interactive shell

```
>>> reload(fibo)
<module 'fibo' from 'fibo.pyc'>
```

Executing Modules as Scripts If you want to execute your module with

```
$ python fibo.py <arguments>
```

you can add

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

to your `fibo.py` file. The first line evaluates to `True` if the file is executed by the Python interpreter. The second line imports a module called `sys`. This module enables you to read argument that the user passed to the script with `sys.argv[]` (in this example the first argument that was passed).

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

In Linux, you can make your file directly executable by adding as first line

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

and setting the file as executable

```
$ chmod +x fibo.py
$ mv fibo.py fibo
$ ./fibo 50
1 1 2 3 5 8 13 21 34
```

In Linux terms, you would now refer to the file as a *Python script*. If you move this script to `/usr/bin`, it will be in your global search path and can be executed from any location of your filesystem. If you are using Linux, you have most likely already used a couple of Python scripts without ever noticing it.

4.1 Standard Modules

Python comes with a library of standard modules. Some of them will be introduced in Section 8. One of the most important modules is `sys`. One of its functions was just introduced. Besides argument parsing, it has functions for e.g. exiting a script `sys.exit()`. Remember you can find out about that in IPython by typing `sys.<TAB>`.

4.2 Packages

A folder containing modules is called a package. This sentence is good to remember but only really true if the folder contains a file called `__init__.py`. A package can of course also consist of subpackages. Here is an example:

```
sound/                    Top-level package
     __init__.py          Initialize the sound ...
      ...package
     formats/             Subpackage for file format ...
      ...conversions
            __init__.py
            wavread.py
            wavwrite.py
            ...
     effects/             Subpackage for sound ...
      ...effects
            __init__.py
            echo.py
            surround.py
            reverse.py
            ...
     filters/             Subpackage for filters
            __init__.py
            equalizer.py
            vocoder.py
            karaoke.py
            ...
     strange/             Subpackage for fibonacci
            __init__.py
            fibo.py
```

Just like modules packages can be imported. You can import a specific subpackage by using *toppackage.subpackage*.

```
>>>import sound.strange.fibo
>>> sound.strange.fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

5 Input and Output

5.1 "Old" string formatting

There are newer and fancier ways to obtain nicely formatted strings in Python, but I chose this one since in my opinion it is the shortest and easiest method of string formatting. A formatted string is a string containing %<*someletter*>. The string is followed by a `%` and has as many variables/values (in a `tuple`) as `%` signs in the string. The <*someletter*> determines how the variables/values are interpreted. A %<*number*><*someletter*> can determine the precision of a number or the number of filling space characters for a string.

```
>>> b = 'hello'
>>> a = '!'
>>> c = "world"
>>> print '%s %s %s'%(b,c,a)
hello world !
>>> print '%20s'%b
               hello
>>> print '%-20s%s'%(b,a)
hello               !
>>> x = 1.23456789
>>> print '%e | %f | %g' % (x, x, x)
1.234568e+00 | 1.234568 | 1.23457
>>> print '%4d'%10
```

```
  10
>>> print '%.4d'%10
0010
```

5.2 Reading and Writing Files

When you open a file with the command `open`, you have to define what you want to do with the file, e.g. `'r'` read, `'w'` write, `'rw'` read and write, `'a'` append (like write, but append to the end of the file). The function will return a file handle to you. On the file handle you can do operations like reading a file or writing contents into the file.

```
>>> f=open('/etc/issue', 'r')
>>> f.read()
'Ubuntu 8.10 \\n \\l\n\n'
>>> f.close()
```

If you are done with the file operations, it is always wise to `close` the filehandle. On line two, we use the function `read` to read the whole file into a string; the `readline` function will read the file line by line; but I especially like `readlines`, it will read the whole file into a `list` where each list element is a line of text.

```
>>> for line in open('/etc/passwd', 'r').readlines():
...     print 'Length: %-5s Content: %s'%(len(line),line...
  ...[:-1])
...
Length: 32   Content: root:x:0:0:root:/root:/bin/bash
Length: 38   Content: daemon:x:1:1:daemon:/usr/sbin:/bin...
  .../sh
Length: 27   Content: bin:x:2:2:bin:/bin:/bin/sh
Length: 27   Content: sys:x:3:3:sys:/dev:/bin/sh
```

5.3 The `pickle` Module

"Serialization is the process of saving an object onto a storage medium [...] such as a file" (Wikipedia). This module puts serialization at your fingertips.

```
>>> import pickle
>>> x=[('man',1),(2,'this is getting'),{True:'so very',...
  ...False:'complicated'}]
>>> f1=open('test.picklefile','w')
>>> pickle.dump(x, f1)
>>> f1.close()
>>> f2=open('test.picklefile','r')
>>> x = pickle.load(f2)
>>> x
[('man', 1), (2, 'this is getting'), {False: '...
  ...complicated', True: 'so very'}]
```

6 Errors and Exceptions

Up until now we typed everything correctly into the interactive shell, but this time we won't!

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

In the example, the error is detected at the keyword print, since a colon (':') is missing before it. File name and line number are printed, so you know where to look in case the input came from a script.

Exceptions Have a look at some exceptions you could encounter in your Python programming adventures

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

The last line of each exception tells you what is wrong. If one of these exception is raised inside a Python script the script will terminate. This is not always desirable, read on to see how to prevent this.

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try ...
  ...again..."
...     else:
...         print "good boy!"
...
```

If code in between the `try` / `except` statements will raise the exception specified behind `except` (`ValueError`), this exception will be caught and the code enclosed by the `except` statement will be executed. If no exceptions are raised, the `except` will be ignored. An optional `else` can be added to define commands that are executed in case no exception is raised.

Raising Exceptions You can also `raise` exceptions whenever you feel like it.

```
>>> raise Exception('spam', 'eggs')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: ('spam', 'eggs')
```

Each exception is a class that inherits from the `Exception` base class. For now we can just `raise` a basic `Exception`. Before we can create our own exceptions we should understand how classes work in Python. At the end of the Section 7 I will show you how to create a custom exception.

7 Classes

Classes are the essential concept of object-oriented programming. The realization of this concept in Python is (as you might already expect) easy to use.

Class Definition Syntax Let's define a class.

```
>>> class MyLameClass:
...     pass
```

Class Objects This was too easy, right? Let's create a more sophisticated class.

```
>>> class Animal:
...     ''' This is an animal '''
...     nana="nana"
...     def __init__(self,number_of_legs):
...         self.legs=number_of_legs
...     def saySomething(self):
...         print "I am an Animal, I have %s legs" % self....
 ...legs
...
```

Just like functions and modules, classes can have documentation strings.
Class objects support two kinds of operations: attribute references and instantiation.
Attribute references use the standard syntax used for attribute references in Python: *obj.name*

```
>>> Animal.nana
"nana"
```

Class instantiation uses the function notation. Just pretend that the class object is a function that returns a new instance of the class.

```
>>> my_pet = Animal(4)
```

The command above created a new instance of the class that was assigned to the local variable `my_pet`. When a class is instantiated, the special function `__init__()` is called. If you know other programming languages, this function is known as constructor. The Python "constructor" is optional.
Class instances have instance variables and instance functions (methods), you will recognize them by the variable `self`. In our example, we have created the instance variable `legs` and the method `saySomething()`. In general, you will only need instance variables and methods if you work with a class (along local variables and functions for the use within the instance functions). You can use a method by writing *obj.method()*

```
>>> my_pet.saySomething()
I am an Animal, I have 4 legs
```

7.1 Random Remarks

self The variable `self` is named "self" because of convention, there is no special meaning behind it. It is however important to follow this convention if you want to use e.g. an external source code documentation generator or if you want to give your source code to other programmers

Just Do It Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
>>> # Function defined outside the class
... def f1(self, x, y):
...     return min(x, x+y)
...
>>> class C:
...     f = f1
...     def g(self):
...         return 'hello world'
...     h = g
...
```

You can also add instance variables and methods to a class instance later on.

```
>>> c=C()
>>> c.new='oh interesting'
>>> c.new
'oh interesting'
>>>
```

7.2 Inheritance

A key feature of object-orientation and classes is inheritance, here is an example

```
>>> class Cat(Animal):
...      '''This is the animal cat'''
...      def __init__(self):
...              '''cats always have 4 legs, this is ...
  ...initialized in this function'''
...              Animal.__init__(self,4)
...      def petTheCat(self):
...              print "purrrrrr"
...
>>> snuggles=Cat()
>>> snuggles.saySomething()
I am an Animal, I have 4 legs
>>> snuggles.petTheCat()
purrrrrr
>>>
```

It is also possible to have a multiple inheritance in Python (`class DerivedClassName(Base1, Base2, Base3)`).

7.3 Private Variables

To make a variable private you add two underscores before the variable name e.g. `self.__furr`. Private variables (and methods) can only be accessed from within the class (or module) they are defined in.

7.4 Odds and Ends

You can (ab)use classes for data storage.

```
>>> class MyData:
...     pass
...
>>> store=MyData()
>>> store.height=200
>>> store.width=400
```

7.5 Exceptions Are Classes Too

Like I promised before, I will now show you how to create a custom exception.

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

8 A Very Brief Tour of the Standard Library

8.1 Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()    # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')
>>> os.path.exists('/etc/issue') # check if a file or ...
  ...folder exists
True
```

8.2 Optparser

The optparse module enables you to write a simple interface to your Python script. The following file will be saved as `test.py` and set to be executable (see Section 4).

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

import optparse,sys

if __name__ == '__main__':

       parser = optparse.OptionParser()
       parser.add_option("-i", "--infile", dest="infile"...
        ..., help="Input file for this script")
       parser.add_option("-o", "--outfile", dest="...
        ...outfile", default="", help="Output file for ...
        ...this script")
       (options,args) = parser.parse_args()

       if not options.infile and not options.outfile:
              print "\nNo input file or output file ...
               ...specified\n"
              parser.print_help()
              sys.exit()
       else:
              contents=open(options.infile,'r').read()
              open(options.outfile,'w').write(contents+'...
               ...\nthis is new content')
```

You can now execute your script and it will give you a nicely structured output that explains its usage.

```
$ ./test.py

No input file or output file specified

Usage: test.py [options]

Options:
  -h, --help            show this help message and exit
  -i INFILE, --infile=INFILE
                        Input file for this script
  -o OUTFILE, --outfile=OUTFILE
                        Output file for this script
$ ./test.py -h
Usage: test.py [options]

Options:
  -h, --help            show this help message and exit
  -i INFILE, --infile=INFILE
                        Input file for this script
  -o OUTFILE, --outfile=OUTFILE
                        Output file for this script
$ echo "hello world">myfile.txt
$ cat myfile.txt
hello world
$ ./test.py -i myfile.txt -o myoutfile.txt
$ cat myoutfile.txt
hello world

this is new content
$
```

9 Exercises

The file ex01.py is attached to this document. Please fill the function bodies, so the given problems are solved (for each problem verify if your solution pass tests included in the file):

```
# A. match_ends
# Given a list of strings, return the count of the number...
 ... of
# strings where the string length is 2 or more and the ...
 ...first
# and last chars of the string are the same.
# Note: python does not have a ++ operator, but += works.
def match_ends(words):
  # +++your code here+++
  return

# B. front_x
# Given a list of strings, return a list with the strings
# in sorted order, except group all the strings that ...
 ...begin with 'x' first.
# e.g. ['mix', 'xyz', 'apple', 'xanadu', 'aardvark'] ...
 ...yields
# ['xanadu', 'xyz', 'aardvark', 'apple', 'mix']
# Hint: this can be done by making 2 lists and sorting ...
 ...each of them
# before combining them.
def front_x(words):
  # +++your code here+++
  return
```

```
# C. sort_last
# Given a list of non-empty tuples, return a list sorted ...
 ...in increasing
# order by the last element in each tuple.
# e.g. [(1, 7), (1, 3), (3, 4, 5), (2, 2)] yields
# [(2, 2), (1, 3), (3, 4, 5), (1, 7)]
# Hint: use a custom key= function to extract the last ...
 ...element form each tuple.
def sort_last(tuples):
  # +++your code here+++
  return

# D. Given a list of numbers, return a list where
# all adjacent == elements have been reduced to a single ...
 ...element,
# so [1, 2, 2, 3] returns [1, 2, 3]. You may create a new...
 ... list or
# modify the passed in list.
def remove_adjacent(nums):
  # +++your code here+++
  return

# A. donuts
# Given an int count of a number of donuts, return a ...
 ...string
# of the form 'Number of donuts: <count>', where <count> ...
 ...is the number
# passed in. However, if the count is 10 or more, then ...
 ...use the word 'many'
# instead of the actual count.
# So donuts(5) returns 'Number of donuts: 5'
# and donuts(23) returns 'Number of donuts: many'
def donuts(count):
  # +++your code here+++
  return

# B. both_ends
# Given a string s, return a string made of the first 2
# and the last 2 chars of the original string,
# so 'spring' yields 'spng'. However, if the string ...
 ...length
# is less than 2, return instead the empty string.
def both_ends(s):
  # +++your code here+++
  return

# C. fix_start
# Given a string s, return a string
# where all occurences of its first char have
# been changed to '*', except do not change
# the first char itself.
# e.g. 'babble' yields 'ba**le'
# Assume that the string is length 1 or more.
# Hint: s.replace(stra, strb) returns a version of string...
 ... s
# where all instances of stra have been replaced by strb.
def fix_start(s):
  # +++your code here+++
  return

# D. MixUp
# Given strings a and b, return a single string with a ...
 ...and b separated
# by a space '<a> <b>', except swap the first 2 chars of ...
 ...each string.
# e.g.
#   'mix', pod' -> 'pox mid'
#   'dog', 'dinner' -> 'dig donner'
# Assume a and b are length 2 or more.
def mix_up(a, b):
  # +++your code here+++
  return

# E. verbing
# Given a string, if its length is at least 3,
# add 'ing' to its end.
# Unless it already ends in 'ing', in which case
# add 'ly' instead.
# If the string length is less than 3, leave it unchanged...
 ....
# Return the resulting string.
def verbing(s):
  # +++your code here+++
  return

# F. not_bad
# Given a string, find the first appearance of the
# substring 'not' and 'bad'. If the 'bad' follows
# the 'not', replace the whole 'not'...'bad' substring
# with 'good'.
# Return the resulting string.
# So 'This dinner is not that bad!' yields:
# This dinner is good!
def not_bad(s):
  # +++your code here+++
  return

# G. front_back
# Consider dividing a string into two halves.
# If the length is even, the front and back halves are ...
 ...the same length.
# If the length is odd, we'll say that the extra char ...
 ...goes in the front half.
# e.g. 'abcde', the front half is 'abc', the back half '...
 ...de'.
# Given 2 strings, a and b, return a string of the form
#  a-front + b-front + a-back + b-back
def front_back(a, b):
  # +++your code here+++
  return
```