Introduction to Python

Falko Krause

November 25, 2008

Abstract

Python (as in Monty Python's Flying Circus) is fun! That is why it is called Python. Python is a highly readable programming language. It is based on multiple programming paradigms. You can use it to program object-oriented, imperative and functional. Python is an interpreted language - Python code can be executed at the moment it is written. Python has great interactive tools and documentation tools.
This tutorial will explain Python mainly by example. It is build upon the official Python tutorial (http://docs.python.org/tutorial/) and can be considered a customized summary of the tutorial. The tutorial contains my opinions about programming with Python and some personal humor (in accordance with the common practice of Python documentation).

1 Getting Started

This part of the tutorial will make use of the software: Python, IPython. The software used is freely available.

Python
http://www.python.org/

IPython
http://ipython.scipy.org
for Windows please have a look at
http://ipython.scipy.org/moin/IpythonOnWindows

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.

```
1  $ python
2  Python 2.5.2 (r252:60911, May 7 2008, 15:19:09)
3  [GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>>
```

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

```
6  >>> myflag = 1
7  >>> if myflag:
8  ...     print "Be careful not to fall off!"
9  ...
10 Be careful not to fall off!
```

If you read the next Section (2) you will understand the example in detail but before that, I will make a short excursion to a software that extends the basic Python interactive shell into a very powerful tool.

IPython If you are serious about learing 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 (and which are very inconvenient to type on a german keyboard layout). The actual use will be demonstrated in this tutorial many times.

2.1 Datatypes

Numbers If you read "The Little Prince", you know that in the world of adults numbers are very important. Here is how Python helps you to get along in the world of adults.
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 was in the last session)

```
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 laguages.

```
>>> 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 `set`s.

```
>>> 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, a values are added/extracted by specifying their key in square barckes.
In `list`s and `tuple`s, 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 `tuple`s, `list`s and `str`s is to specify start and end positions separated by a colon in the square brackets (*list[start:end]*).

```
>>> word = 'W00T this Python lesson is awesome'
>>> word.split()
['W00T', '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 too.

```
>>> 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 you 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 the answer to 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 sagMiau(who):
...     return who+" sagt Miauuuuu"
...
>>> print sagMiau('Jannis')
Jannis sagt 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 sagKompliment(who,person="Falko",antwort="Oh danke"):
...     return who+' sagt: '+person+" du hast die Haare schoen.\n"+person+" sagt: "+...
 ...antwort
...
>>> print sagKompliment("Jannis","Wolf")
Jannis sagt: Wolf du hast die Haare schoen.
Wolf sagt: Oh danke
>>> print sagKompliment("Timo",antwort="Verarschen kann ich mich selber")
Timo sagt: Falko du hast die Haare schoen.
Falko sagt: Verarschen kann ich mich selber
```

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

```
>>> kmplmnt=sagKompliment
>>> print kmplmnt("Falko")
Falko sagt: Falko du hast die Haare schoen.
Falko sagt: Oh danke
```

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. If you write the module `fibo.py` (contents below) with your favorite text editor -

```
"""
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'
```

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

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

you can add

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

to your `fibo.py` file. The fist line evaluates to `True` if the file is executed by the Python interpreter. The second line imports a module called `sys`. This module eables 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 scrip to `/usr/bin`, it will be in your global search path an 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 function 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`.

```
>>> 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 in 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 instace 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'
>>>
```

Naming Conventions There are two styles of writing strings in source-code that I like.

CamelCase
: writing compound words or phrases in which the words are joined without spaces and are capitalized within the compound: ThisIsCamelCase

snake_case
: writing compound words or phrases in which the words are joined with and underscore: this_is_snake_case

In our project, we decided to follow the convention

variables	snake_case
functions	camelCase
classes	CamelCase

This will make your source-code easy to read (for yourself!). Of course you still need to find good names for your variables, classes and functions.

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.highth=200
>>> store.with=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!'
```

7.6 Generators

Generators are written like regular functions but use the `yield` statement (instead of `return`) whenever they want to return data.

```
>>> def reverse(data):
...   for index in range(len(data)-1, -1, -1):
...       yield data[index]
...
>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

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 Writing a Sophisticated Bioinformatics Application

In this last section, we will write a very sophisticated bioinformatics application using the combined knowledge of this tutorial. For that, we will need some extra tools. The tools are freely available and run on Linux, Windows and Os X.

libSBML
: http://sbml.org/Software/libSBML (don't forget to install the Python bindings)

Graphviz
: http://www.graphviz.org/

Epydoc
: http://epydoc.sourceforge.net/

The application will generate a graphical representation of an SBML model. For that, it will extract the information from the SBML file using the libSBML. The libSBML can be imported as a module. The graph images are generated by the graphviz program `dot`.
The following source-code will be saved as `sbml_graph.py` and set as executable.

```
#!/usr/bin/env python
'''
B{SBML Graph Representation}
this module generates a graphical representations of SBML models
'''

import os,sys,optparse,libsbml

dot_path= '/usr/bin/dot' # configure the path to graphviz dot executable here

class SBMLGraph():
    '''this class enables you to create graph representations of SBML models'''

    def __init__(self,sbml_file_name):
        '''
        check if the sbml file exists
        if it exists generate a graph representation
        if not return an error message to the use and exit
        @param sbml_file_name: path to the sbml file
        @type sbml_file_name: str
        '''
        self.graph_dot=''
        self.in_file_path=sbml_file_name
        if not os.path.exists(self.in_file_path):
            print 'The file %s was not found' % self.in_file_path
            sys.exit(1)
        else:
            document = libsbml.readSBMLFromString(open(self.in_file_path,'r').read())
            model= document.getModel()
            self.graph_dot=self.generateGraph(model)
    def generateGraph(self,model):
        '''
        @param model: libsbml model instance
        @type model: libsbml.Model
        @return: graph representation as string in dot format
        @rtype: str
        '''
        #generate a dictionary of all species in the sbml file
        id2libsbml_obj={}
        for species in list(model.getListOfSpecies()):
            id2libsbml_obj[species.getId()]=species

        out="digraph sbmlgraph {\n"
```

```
        #go through all reactions
        for reaction in list(model.getListOfReactions()):

            for i in range(reaction.getNumReactants()):
                reactant_name= id2libsbml_obj[reaction.getReactant(i).getSpecies()].getName...
                  ...() or reaction.getReactant(i).getSpecies()
                out+= "\tS_%s -> R_%s\n" % (reactant_name,reaction.getName() or reaction....
                  ...getId())

            for i in range(reaction.getNumProducts()):
                product_name= id2libsbml_obj[reaction.getProduct(i).getSpecies()].getName()...
                  ... or reaction.getProduct(i).getSpecies()
                out += "\tR_%s -> S_%s\n" % (reaction.getName() or reaction.getId(),...
                  ...product_name)
        return out +"}"

    def writeImage(self,format='svg',filename=''):
        '''
        write the graph image to the hard disk
        @param format: output image format
        @type format: str
        @param filename: filename of image
        @type filename: str
        '''
        if not filename:
            filename = os.path.splitext(os.path.basename(self.in_file_path))[0]+'.'+format

        open('temp.dot','w').write(self.graph_dot)
        os.system("%s temp.dot -T%s -o %s"%(dot_path,format,filename))
        os.remove('temp.dot')


if __name__ == '__main__':

    parser = optparse.OptionParser()
    parser.add_option("-i", "--infile", dest="infile", help="Input: an SBML file")
        parser.add_option("-o", "--outfile", dest="outfile", default="", help="specify a ...
          ...out filename, this is optional")
    parser.add_option("-f", "--imageformat", dest="format", default="", help="output ...
      ...formats are: svg, png, ps, eps, tiff, bmp")
    (options,args) = parser.parse_args()

    if not options.infile:
        print "\nNo input file specified\n"
        parser.print_help()
        sys.exit()
    else:
        graph=SBMLGraph(options.infile)
        graph.writeImage(filename=options.outfile,format=options.format)
```

Here are some execution examples of our script. Since we do not have a SBML model yet, we download a SBML model from the BioModel.net database with `wget` (line 17).

```
1  $ ./sbml_graph.py

3  No input file specified

5  Usage: sbml_graph.py [options]

7  Options:
8    -h, --help          show this help message and exit
9    -i INFILE, --infile=INFILE
10                       Input: an SBML file
11   -o OUTFILE, --outfile=OUTFILE
12                       specify a out filename, this is optional
13   -f FORMAT, --imageformat=FORMAT
14                       output formats are: svg, png, ps, eps, tiff, bmp
15 $ ./sbml_graph.py -i nofile.xml -f png
16 The file nofile.xml was not found
17 $ wget http://www.ebi.ac.uk/biomodels/models-main/publ/BIOMD0000000001.xml
18 ...
19 $ ls
```

```
BIOMD0000000001.xml sbml_graph.py
$ ./sbml_graph.py -i BIOMD0000000001.xml -f png
$ ls
BIOMD0000000001.png BIOMD0000000001.xml sbml_graph.py
```

The created image can be seen in Figure 1. As a final step, we autogenerate a

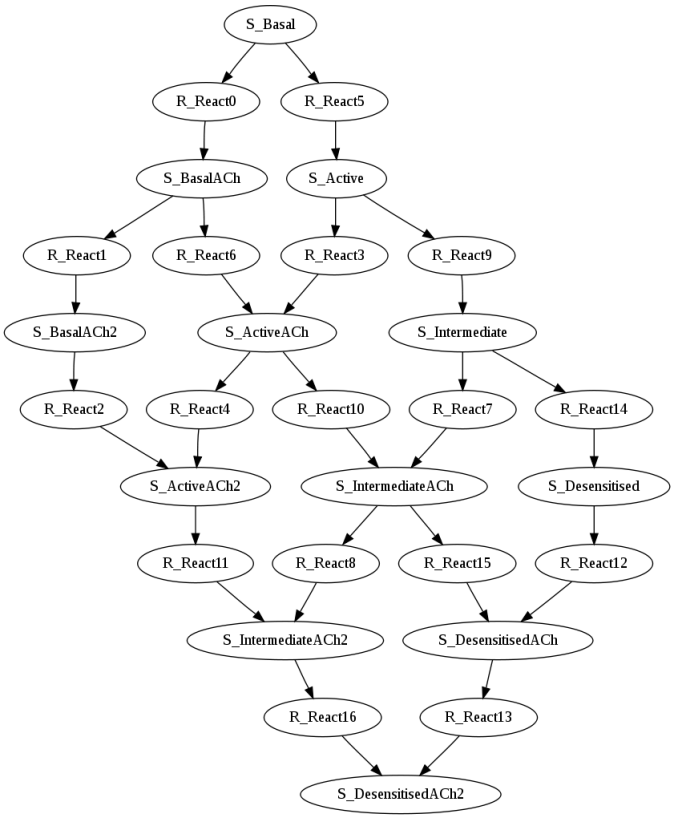

Figure 1: This is the resulting image from our script `sbml_graph.py`

good-looking source-code documentation with Epydoc. As you notice, there are some special strings in the Python documentation like `@param`. These strings enhance the documentation with some text formatting. Let's generate the documentation.

```
$ epydoc sbml_graph.py
```

By default, Epydoc will generate a HTML documentation. The main page can be seen in Figure 2 and the documentation of the class `SBMLGraph` in Figure 3.

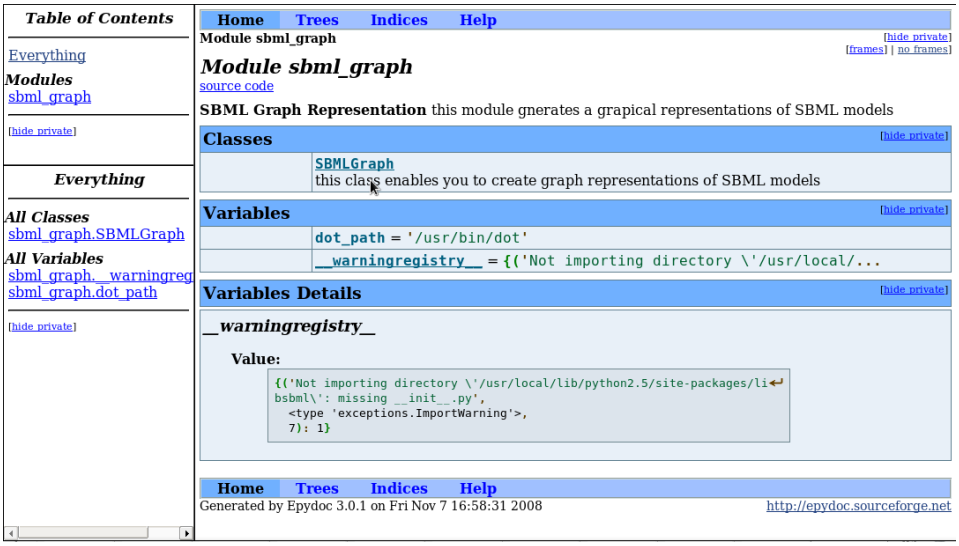

Figure 2: This is the index page of the html documentation generated by Epydoc.

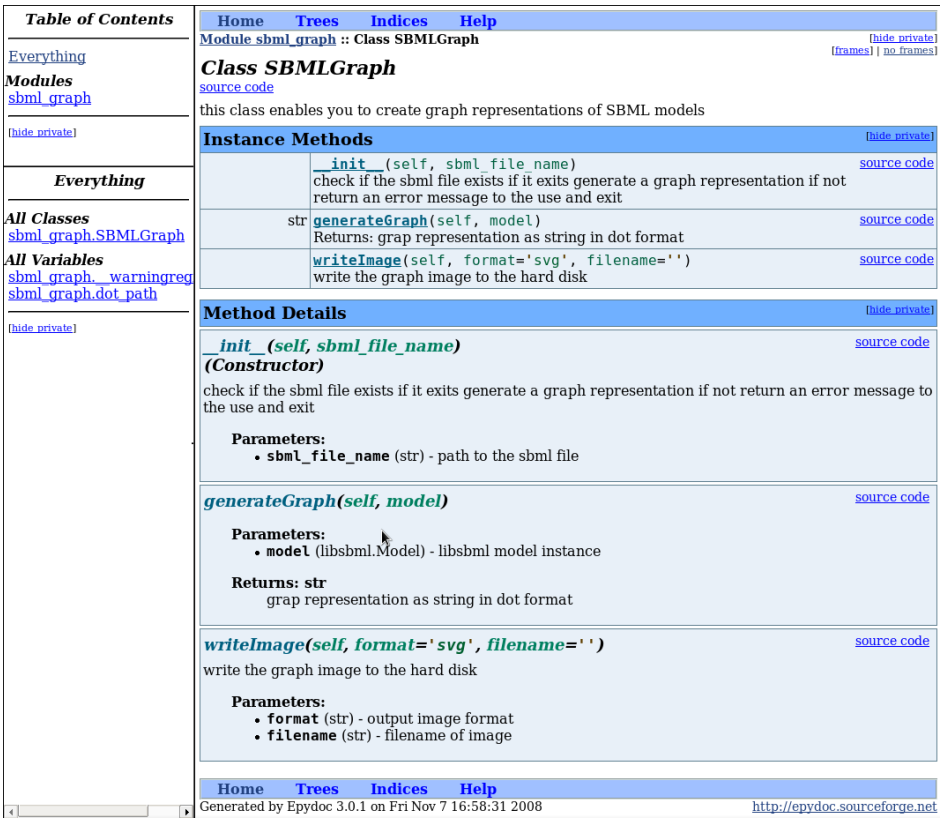

Figure 3: This is the documentation of the class `SBMLGraph` generated by Epydoc.