Astronomical data analysis using Python

Lecture 4

In [1]:
```python
a = "Hello!"

for i, c in enumerate(a):
    print ("Character no. %d is %s" % (i+1, c)) # i+1 is used because Python is 0-indexed.
```

```
Character no. 1 is H
Character no. 2 is e
Character no. 3 is l
Character no. 4 is l
Character no. 5 is o
Character no. 6 is !
```

```
In [2]: a = "Hello!"

        for i in enumerate(a):
            print (i)
```

```
(0, 'H')
(1, 'e')
(2, 'l')
(3, 'l')
(4, 'o')
(5, '!')
```

What we have covered so far

- Why use Python? Its basic ideas and design philosopy (Zen of Python), learning resources.
- Our slice of Python - astronomical data analysis
- Some data types: **float, str, int** and their type casting functions
- **print()** function and string formatting, tuple unpacking
- builtin functions - **dir(), help(), type()**
- **bool** and expression evaluation
- **sequences** and sequence **slicing**, **mutability**
- almost everything in Python is an **object** - full support for Object Oriented Programming
- Conditionals **if-elif-else**
- Loops: **while** and **for**
- **Indentation**
- **break** and **continue**
- Ordered structures - **sequences, lists**
- Unordered (labelled) structure - **dictionary**
- Elements of good programming style - **PEP 8**

What we will cover next:

- File I/O
- Functions
- Modules

Handling Files

Let us study how to handle files through a simple exercise. The basic approach involves creating file objects in Python and use various methods associated with file objects to handle file I/O. Such an approach is used in many other programming languages.

- open() function is used to create a file object.
- fileObject.read() - reads entire file as one big string.
- fileObject.write() - to write a string into a file.
- fileObject.readlines() - to read each line as an element of a list.
- fileObject.writelines() - to write a set of lines, each one being a string.
- fileObject.close() - to close a file (buffer flush)

Program to ”Double Space” a File

In [1]:
```
"""
Program to create a double spaced file.
Input: File Name
Output: Modified File with .sp extension
"""

import sys # we need this to parse command line arguments.
import os # we need this to check for file's existence
```

In [2]:
```
# Check number of arguments.
if len(sys.argv) == 2:
        infile_name = sys.argv[1]
else:
        print ("Oops! Incorrect Number of Arguments.")
        sys.exit(2)

# Check if file exists.
if not os.path.isfile(infile_name):
        print ("File doesn't exist.")
        sys.exit(3)
```

```
Oops! Incorrect Number of Arguments.

An exception has occurred, use %tb to see the full traceback.

SystemExit: 2

/home/yogesh/.local/lib/python3.6/site-packages/IPython/core/interactiveshell.
py:3304: UserWarning: To exit: use 'exit', 'quit', or Ctrl-D.
  warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```

In [1]:
```
# Open the input file.
infile_name='sample.txt'
infile = open(infile_name, "r")

# Open an output file.
outfile = open(infile_name + ".sp", "w")

# Loop over each line, add new line to each line.
for line in infile.readlines():
        line = line+"\n"
        outfile.write(line)

outfile.close()
infile.close()
```

Please practice reading and writing simple textfiles on your own. Assignment 1 will require you to have these skills. Reading and writing of large data files - images, spectra, data cubes - will be taught a few lectures later. The process - open filehandle, read/write data, close filehandle remains the same.

Functions

Blocks of code that perform a specific task. Very useful in avoiding replication of code where the same or a very similar task is carried out multiple times inside a program.

In Python, a function is defined using the "def" keyword.

We have already seen examples of built-in functions.

- float(), dict(), list(), len() etc.
- math - sqrt(), radians(), sin()
- open(), type() etc.

A Simple Function with no arguments

In [5]:
```
def myfun():
    print ("Hello World!")
    print ("Nice to see you.")

print ("Outside the function.")
```

```
Outside the function.
```

a function is not executed until it is called. Pay attention to how the statements indented one level up are part of the function while the statement indented at the same level is not a part of the function.

In [9]:
```
myfun() # This is how you call our function.
```

```
Hello World!
Nice to see you.
```

Function With One Argument

```
In [5]: def myfun(a):
            print ("Inside MyFun!")
            print (a)
```

```
In [6]: myfun() # WILL GIVE ERROR.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-f67cbbab1c46> in <module>
----> 1 myfun() # WILL GIVE ERROR.

TypeError: myfun() missing 1 required positional argument: 'a'
```

As per function definition, one argument / input is needed. An attempt to call the function with none gives an error. EVEN supplying two arguments is wrong.

```
In [7]: myfun("An Input")
```

```
Inside MyFun!
An Input
```

REMEMBER

Python is a dynamically typed language. The true strength of this lies in the fact that you can also call the above function with a float or integer or list input!

```
In [8]: myfun(5)
```

```
Inside MyFun!
5
```

```
In [9]: myfun( [1,2,3] )
```

```
Inside MyFun!
[1, 2, 3]
```

Functions that ”return” something.

```
In [10]: def add(a,b):
             return a+b
```

```
In [11]: a = add(2,3)
         print (a)
```

```
5
```

A function that does not have a return statement returns by default an object called "None".

```
In [12]: b = myfun("Hello")
```

```
Inside MyFun!
Hello
```

```
In [13]: print (b)
```

```
None
```

Functions can return more than one value at a time!

```
In [18]: def sumprod(a,b):
             return a+b, a*b
```

```
In [19]: s, p = sumprod(2,3)
```

Well, technically - Python is returning only one object but that one object is a tuple - in the above case

Optional Arguments

"I want a function to assume some values for some arguments when I don't provide them!" Let's see how this is achieved.

```
In [14]: def myfun(message = "Good Day!"):
             print (message)
```

```
In [15]: myfun("Hello World")
```

```
Hello World
```

```
In [16]: myfun()
```

```
Good Day!
```

Functions with Arbitrary Number of Arguments

```
In [23]: def sumitall(*values):
             total = 0
             for i in values:
                 total += i
             return total
```

```
In [24]: sumitall(2,3,4,5)
```

```
Out[24]: 14
```

```
In [25]: sumitall(2,3,4)
```

```
Out[25]: 9
```

Mixture of Arguments

```
In [26]: sumitall()

Out[26]: 0
```

```
In [27]: def sumitall2(val1, *values):
             total = val1
             for i in values:
                 total += i
             return total
```

```
In [28]: sumitall2(2)

Out[28]: 2
```

```
In [29]: sumitall2(2,3,4)

Out[29]: 9
```

```
In [30]: sumitall2() # WILL GIVE AN ERROR.

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-810847688c33> in <module>
----> 1 sumitall2() # WILL GIVE AN ERROR.

TypeError: sumitall2() missing 1 required positional argument: 'val1'
```

This way, you can design functions the way you want by imposing both a minimum number of arguments and have flexibility of an arbitary number of them!

Functions are Objects

Like lists, dictionaries, ints, floats, strings etc. you can pass functions to other functions since they are just objects.

```
In [31]: def myfun(message):
             print (message)
```

```
In [32]: def do(f, arg):
             f(arg)

         do(myfun, "Something")
```

```
Something
```

```
In [17]: x = myfun # simple variable assignment
         x("Hello!")
```

```
Hello!
```

Function Documentation

Recall using help(math.hypot) to get help on understanding how to use hypot() function. Can we design a function myfun() and ensure that help(myfun) also gives a nice "help" output?

In [34]:
```
def myfun(a,b):
    """
    Input: Two Objects
    Output: Sum of the two input objects.
    """
    return a+b
```

In [35]:
```
help(myfun)
```

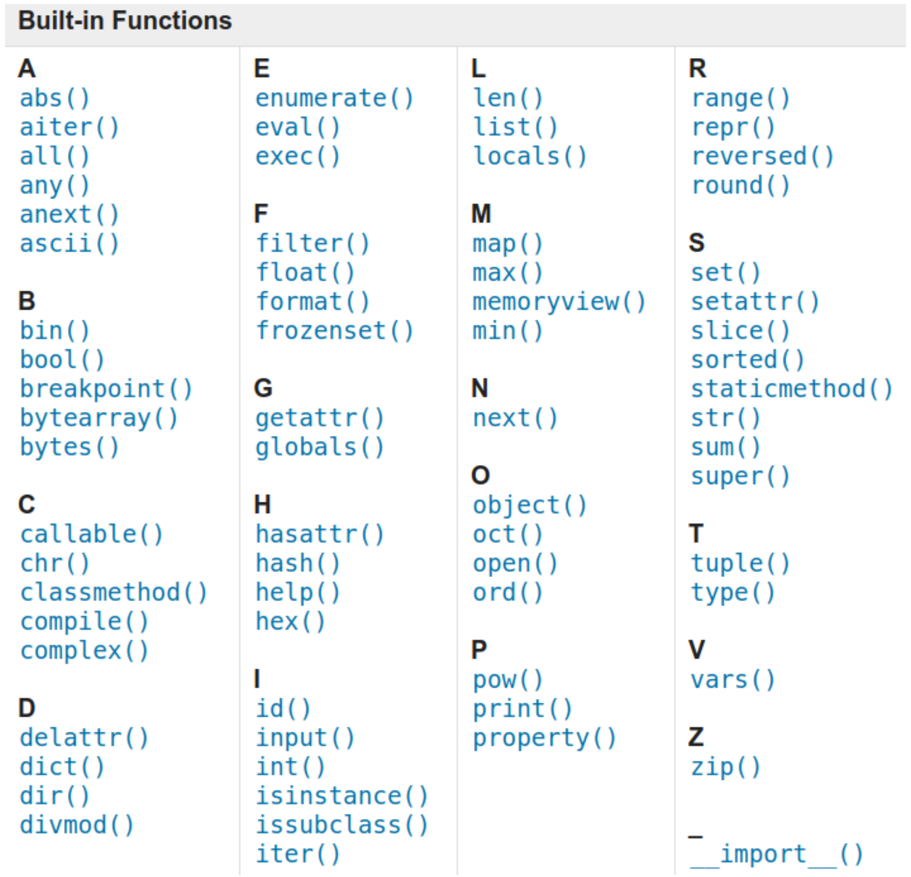
```
Help on function myfun in module __main__:

myfun(a, b)
    Input: Two Objects
    Output: Sum of the two input objects.
```

When designing functions of your own, it is always good to document what the function does so that you and others can use it in the future with ease.

Builtin functions in the builtin module

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

See: https://docs.python.org/3/library/functions.html#built-in-funcs (https://docs.python.org/3/library/functions.html#built-in-funcs)



Built-in Functions

A	**E**	**L**	**R**
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()	**F**	**M**	
ascii()	filter()	map()	**S**
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	**G**	**N**	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	**H**	object()	
callable()	hasattr()	oct()	**T**
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	**Z**
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		**_**
	iter()		__import__()

Modules

Modules can be considered as "namespaces" which have a collection of objects which which you can use when needed. For example, math modules has 42 objects including two numbers "e" and "pi" and 40 functions.

Every program you execute directly is treated as a module with a special name __main__.

So, all the variables you define, the functions you create are said to live in the namespace of __main__.

When you say the following, you are making the namespace of **math** available to you.

```
import math
```

To then access something inside **math**, you say

```
math.object
```

So what happens when you ”import”

- Python interpreter searches for math.py in the current directory or the installation directory (in that order) and compiles math.py, if not already compiled.
- Next, it creates a handle of the same name i.e. "math" which can be used to access the objects living inside math.

```
In [36]: import math
         type(math)
```

```
Out[36]: module
```

Other way to ”import”

In the above example, you are accessing objects inside **math** through the module object that Python created. It is also possible to make these objects become a part of the current namespace.

```
from math import *
```

```
In [37]: from math import *
         radians(45) # no math.radians required.
```

```
Out[37]: 0.7853981633974483
```

WARNING: The above method is extremely dangerous! If your program and the module have common objects, the above statement with cause a lot of mix-up! This is the single most stupid thing you can do as a Python programmer, so please don't do it.

A Middle Ground

If there is an object you specifically use frequently and would like to make it a part of your main namespace, then,

```
from ModuleName import Object
```

Strictly speaking even this is not at all advisable. Please avoid, if possible.

```
In [38]: from math import sin
         print (sin(1.54))
```

```
0.9995258306054791
```

NOTE: If you import the same module again in the same program, Python does not reload. Use reload(ModuleName) for reloading.

Aliases for a Module

If you have decided to access a module's objects from its own namespace, you can choose to alias the module with a name.

```
import numpy as np
np.array(...)
```

Try to use standard abbreviations - e.g. np for numpy

Another example,

```
import matplotlib.pyplot as plt
plt.plot(x,y)
```