Phast-CP

Quick-guide to Python

Starting the environment:

The environment we will use is Ipython with the pylab module. Ipython is an enhanced interactive python terminal, this environment can be started with the command:

```
bash-4.2$ ipython --pylab
```

Variables:

The basic python variable types include ints, floats, strings, lists and tuples. Lists contain a list of variables and tuples contain one or more variables packed together, and for both lists and tuples the variables that are part of them do not have to be of the same type. In python a variables type is implicit, and can be changed dynamically by assigning it a value with a different type. Variables are implicitly defined when they are assigned, so to create a variable you simply have to assign a value to a name. Another useful type is the numpy array that is included with pylab, arrays can be created by passing a list or tuple to the function "array". Numpy arrays and lists can be interchanged using the commands "array" and ".tolist()".

```
In [1]: a = 15

In [2]: b = "Hello World!"

In [3]: c = 2.99e+8
```

```
In [1]: data_list = [2, 4, 2.6e+2, "a string"]

In [2]: tuple_info = ("some string", 3, 0.003)

In [3]: numpy_array = array([2, 3, 5, 7, 11])
```

```
In [4]: data = [1, 0.1, 10]

In [5]: data_array = array([5, 2, -7, 1])

In [6]: array(data)
Out[6]: array([  1. ,   0.1,  10. ])

In [7]: data_array.tolist()
Out[7]: [5, 2, -7, 1]
```

The standard operates work on the numerical types, such as +, -, *, /, % and ** for exponentiation. Lists and strings can be concatenated by the + operator and arrays are concatenated with the function "concatenate" witch takes a tuple of numpy arrays and returns a joined array. Tuples can be packed and unpacked by assigning with a series of variables separated by commas, the number of variables must equal the number of values in the tuple. When working with list or arrays the elements inside the array can be accessed with the "[ ]" operator. The elements are assigned an index in order of there position in the array or list staring at zero and going up to the number of elements minus 1. An element can be accessed by indicating the required index within square brackets directly after the array name. Also you can request a sub set of the array or list by indicating the index of the first and last element required separated by a colon, this

```
In [8]: packed = 1,2,3,4

In [9]: print packed
(1, 2, 3, 4)

In [10]: a, b, c, d = packed

In [11]: print c
3
```

```
In [8]: data = [1, 0.1, 10]

In [9]: data_array = array([5, 2, -7, 1])

In [10]: data[2]
Out[10]: 10

In [11]: data_array[1:3]
Out[11]: array([ 2, -7])
```

operation will return an array or list with all of the elements starting with and including the first one indicated going up to but not including the last one indicated.

Documentation:

Within the Ipython interpreter the one way to get information about a function, method or module is to get its doc-string with the h0elp command. If the documentation is longer that the screen you can hit the down arrow to see more. To exit from help enter the 'q' key. A second way is to enter the name of a function, method or module followed by '?' which will display similar information as the help command, but it also includes some extra information including what type of object it is and where it is found.

```
In [12]: help(append)


In [13]: append?
```

Control structures:

The main control statements in python are the “if” statement and “for” loop. The “if” statement must be followed by a statement that evaluate to True or False and followed by a colon. After an if statement the following statements should be indented. An “if” statement may be followed by “elif” and or an else statement. The “for” loop uses the following syntax, “for” then some temporary variable name followed by “in” then a list like variable such as a list or numpy array. The “for” loop is commonly used with the xrange() statement.

```
In [7]: if 25 > 23:
   ...:     print "25 is larger than 23"
   ...:
25 is larger than 23
```

```
In [8]: for i in [1, 2, 3]:
   ...:     print i
   ...:
1
2
3

In [9]: for value in array((2, 4, 6, 8)):
   ...:     print value * value
   ...:
4
16
36
64

In [10]: for iteral in xrange(3):
   ....:     print iteral
   ....:
0
1
2
```

Program structure:

When writing a python script it is often useful to reuse code that has been write in the form of modules. Modules can be imported with the command “import ” followed by the name of the module. Once a module is imported the functions and variables contained within it can be accessed by prepending the name of the module to the name of the function.

```
In [11]: import math

In [12]: math.cos(2)
Out[12]: -0.4161468365471424
```

Functions are created with the “def” statement followed by the name of the function and the name of the arguments in brackets and ending with a colon, the contents of the function must be indented. After some statements the function may end by returning a variable. Lambda functions are similar to standard functions except that they are not assigned a name and only contain one statement that is returned, but can be used like functions. A lambda function is defined by the statement lambda followed by the arguments and a colon witch is followed by one statement using the arguments.

```
In [16]: def square(value):
   ....:     return value * value
   ....:

In [17]: data = [1, 2, 3, 4]

In [18]: print map(square, data)
[1, 4, 9, 16]

In [19]: print map(lambda val: val * val, data)
[1, 4, 9, 16]
```

Instances are created in a way similar to variables except that they are called like a function and they can have functions called methods and variables called attributes embedded in them. The methods and attributes are accessed by entering the name of the instance followed by a dot and then the method or attribute name.

```
In [20]: data_array = array([1, 2, 3, 4])

In [21]: data_array
Out[21]: array([1, 2, 3, 4])

In [22]: data_array.tolist()
Out[22]: [1, 2, 3, 4]
```

Basic IO:

For command line io the main functions wold be the “print” statement and the “raw_input” function. “print” takes one or more variable or literal separated by commas and prints them to the command line. “raw_input” takes a string to be displayed as a prompt and returns the user input as a string.

```
In [23]: print "a string", 123
a string 123

In [24]: user_input = raw_input("some data: ")
some data: 42

In [25]: print user_input
42
```

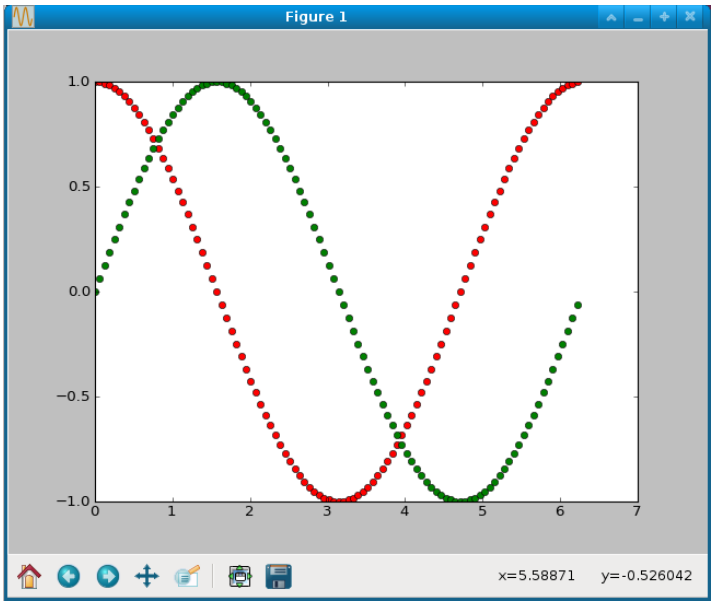
For file io the main functions are “open” and “loadtxt”. The function “open” opens a file for reading or writing, it takes the file name as a string a either 'r' or 'w' for reading or writing. This function returns a instance of a file witch if opened in read mode can be iterated over like a list, or if it is in write mode it can be written to with the method “write”. After file io is finished the object should be closed with the “close” method. The “loadtxt” function from the numpy module can load data from a text file, the file name is passed as an argument, and the data is returned as a two dimensional array of the data in each row and column.

```
In [26]: f = open("test.txt", 'w')

In [27]: f.write("1 2 3\n2 3 4\n3 4 5")

In [28]: f.close()

In [29]: f_in = open("test.txt", 'r')

In [30]: for line in f_in:
   ....:     print line
   ....:
1 2 3

2 3 4

3 4 5

In [31]: loadtxt("test.txt")
Out[31]:
array([[ 1.,  2.,  3.],
       [ 2.,  3.,  4.],
       [ 3.,  4.,  5.]])
```

Plotting:

Pylab has a simple method to plot graphs using the function “plot”. The function “plot” takes two lists or arrays of the same length and uses the first as the x coordinate and the second as the y coordinate. Then it takes a string that determines how the data is displayed, if the string is “ro” the data pints will be displayed as red circle or if it is “gx” it will be displayed as a green x. Also you can pass multiple sets of x list, y list and display string to display multiple data sets on the same graph.

```
In [28]: import math

In [29]: x = []

In [30]: y1 = []

In [31]: y2 = []

In [32]: for i in xrange(100):
   ....:     x = x + [i * 2 * math.pi / 100.0]
   ....:

In [33]: for x_val in x:
   ....:     y1 = y1 + [math.cos(x_val)]
   ....:     y2 = y2 + [math.sin(x_val)]
   ....:

In [34]: plot(x, y1, "ro", x, y2, "go")
Out[34]:
[<matplotlib.lines.Line2D at 0x2617750>,
 <matplotlib.lines.Line2D at 0x2617a50>]
```



Closing the environment:

When you are finished a session you can close the environment by entering the command “exit”.

```
In [32]: exit
bash-4.2$
```

Further reading:

To learn more go to “http://csa.phys.uvic.ca/teaching/online-courses”.