1. Modules
2. IO
3. Lambda Functions
4. Some tips and tricks
5. Regex

What are they?

- Modules are collections of classes or functions or definitions or simply python code. (Python files are)
- They are re-usable. One could use the codes from a different module
- There are some standard Python modules (eg. `sys, math`)
- To use the code/module, one has to use the keyword `import` and also should know what exactly is available in the module.

```
1
2  def fib(n):
3      a, b = 0, 1
4      while b < n:
5          print b,
6          a, b = b, a+b
7
8  def fib2(n):
9      result = []
10     a, b = 0, 1
11     while b < n:
12         result.append(b)
13         a, b = b, a+b
14     return result
```

```
>>> 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'
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Different Ways

- To use the `fibo.py`, one could do straight `import fibo` which imports the module
- One could import functions separately too `from fibo import fib`
- When one tries to import, Python looks for the file in the same directory first, or in the Python path. (usually `/usr/local/lib/python/`)

Running Modules/Files

- One could run the modules straight as scripts (applicable to python files as well) `$$python fibo.py <arguments>`
- When the file is run as above, the `__name__` attribute would be set to `"__main__"`. So a check for the value of the attribute should enable us to choose what to do. (Example follows)
- If the file is made executable, one could also use it as a command `$$./fibo.py <arguments>`

```
1 if __name__ == "__main__":
2     import sys
3     fib(int(sys.argv[1]))
4
5 .....
6
7 $ python fibo.py 50
8 1 1 2 3 5 8 13 21 34
```

Dir()

There is a built-in function which lists all the available functions/methods/attributes of a module.
(Example below)

```
>>> import fibo, sys
>>> dir(fibo)
('__name__', 'fib', 'fib2')

>>> dir()
('__builtins__', '__doc__', '__file__',
 '__name__', 'a', 'fib', 'fibo', 'sys')

>>> import __builtin__
>>> dir(__builtin__)
 (.....................................)
```

IO - Console

- Output : We already have used `print`
- Command line arguments: `sys.argv[i]`
- While the program is running:
  There are two methods. They both return a string which was provided by the user. (By hitting the RET)
  - `raw_input()` : returns the input string
  - `input()` : tries to execute the input string. DANGEROUS: Never use this to get input from users. One could compromise the system.

Read/Write Files

- There is a `file` object in python. That is used for the file operations
- One can have a handle to a file by simply using `open('filename')`

  Something similar to the `FILE *fp = fopen("filename", "r")` of *C*

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

Where are you?

- All the operations (to see soon) happens, starting from the present "position" in the file.
- Every operation changes the position of the 'cursor' in the file. (When opening a file, the seek-position is set to be 0)
- To know where we stand now, use `f.tell()`
- To move to a specific location, use `f.seek(index)`
- One has to pay attention to close the files when it is nomore needed. Otherwise, next time it could have a wrong position. `f.close()`

Reading files

- `f.read()` - Gives the text content of the file pointed to by `f`
- `f.readline()` - Gives each line by line from the file. First call gives the first line, the next call gives the next line.
- `f.readlines()` - Gives a list of the lines in the file.
- One can also directy iterate over the lines in the file.

```
>>> f = open("test.txt")
>>> f.read()
'\nThis is a test file\nThis is the second line\n
>>> f.tell()
69
>>> f.seek(0)
>>> f.readline()
'\n'
>>> f.readline()
'This is a test file\n'
>>> f.readlines()
('This is the second line\n', 'This is the final
>>> f.seek(0)
>>> for l in f:
...         print l
...
```

```
19
20 This is a test file
21
22 This is the second line
23
24 This is the final line
```

Writing

- When one wants to write to a file, then the `open` call has to be given specific parameters.
  - open(f, 'w'): Writeable (seek = 0)
  - open(f, 'a'): Would be appended
  - open(f, 'r+'): Readable and Writeable
  - open(f, 'r+a'): Readable and Appendable
  - open(f, 'r'): Readable
- Without a parameter, it is automatically only readable
- Using `f.mode` one can see the mode of opening.

Writing

- `f.write(string)` : Writes to `f`
- `f.writelines(col)` : Writes each member of the col (some collective object), to the file
- `f.flush()` : Writes it really to the file from the memory. Happens with `f.close()` automatically.

Pickle

- `pickle.dump(x, f)` - dumps *x* to the file
- `x = pickle.load(f)` - reads from the file to *x*

Shelves.

What are they

Mini functions.
There are times when we need to write small functions, perhaps not necessary for a reuse of anything. Then we use lambda forms.

- Only expressions can be used. No statements
- No local variables
- Only one expression

```
>>> def f(x):
...     return x*x
...
>>> print f(7)
49
>>> g = lambda x : x*x
>>>
>>> print g(7)
49
```

```
>>> def makeincre(n) : return lambda x: x + n
...
>>>
>>> incr2 = makeincre(2)
>>> incr9 = makeincre(9)
>>>
>>> print incr2(10)
12
>>> print incr9(10)
19
>>>


>>> add = lambda a, b: a+b
>>> add(10, 13)
23
```

To Note

- Variables : A comma separated list of variables. Not in parens.
- Expression : A normal python expression. The scope includes both the variables and the local scope.

Truth values

We already saw that empty means FALSE in python.
The same applies to zero too.

```
1  my_object = 'somestring'
2
3  if len(my_object) > 0:
4      print 'my_object is not empty'
5
6  if len(object):
7      print 'my_object is not empty'
8
9  if object != '':
10     print 'my_object is not empty'
11
12 if object:
13     print 'my_object is not empty'
```

String Theory

- The strings in python contains many methods. One of them is `find` which returns the position of a substring
- But if we need only to check if the substring is present in a big string, we don't need to use that. (More readable code)
- `split` and `join`: These are two string methods which are very useful.

```
>>> string = 'Hi there'
>>> if string.find('Hi') != -1:
...     print 'Success!'
...
Success!
>>> if 'Hi' in string:
...     print 'Success!'
...
Success!
>>>
>>> mystr = 'this is a one two three string'
>>> words = mystr.split()
>>> words
('this', 'is', 'a', 'one', 'two', 'three', 'strin
>>> '*'.join(words)
'this*is*a*one*two*three*string'
>>>
```

Filter, Map and Reduce

`func_tool(function, sequence)`

- `filter`: Filter accepts two parameters, one is a function and the second one a sequence. It returns a list of the elements of the sequence for which the function is TRUE.
- `map`: The returned list would be the results of applying the function to each member of the sequence.
- `reduce`: Initially, the function is applied to the first two elements of the sequence, and the result used as the parameter along with the next elements of the sequence.

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
(5, 7, 11, 13, 17, 19, 23)
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
(1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
(0, 2, 4, 6, 8, 10, 12, 14)
```

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
>>> def sum(seq):
...         def add(x,y): return x+y
...         return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

ZIP

In case we need to combine two lists, How do we do it?
How do we create a dictionary from two lists?

```
>>> l = (x for x in range(1, 10))
>>> k = (y for y in range(90, 99))
>>> l
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> k
(90, 91, 92, 93, 94, 95, 96, 97, 98)
>>>
>>>
>>> lk = ((l(x), k(x)) for x in range(len(l)))
>>> lk
((1, 90), (2, 91), (3, 92), (4, 93), (5, 94),
            (6, 95), (7, 96), (8, 97), (9, 98))
```

```
>>>
>>> lkl = zip(l, k)
>>> lkl
((1, 90), (2, 91), (3, 92), (4, 93), (5, 94),
            (6, 95), (7, 96), (8, 97), (9, 98))
>>>
>>> lkd = dict(lkl)
>>> lkd
{1: 90, 2: 91, 3: 92, 4: 93, 5: 94,
         6: 95, 7: 96, 8: 97, 9: 98}
>>>
```

Regular Expressions Basics

- Alphabets
- Operators : $*, +, ?, |$
- Examples : $(0|1)*, a(bc|d)*, a+$

In Python

In python, there exists a module for regular expressions. Here we can see some example symbols

- . - Stands for any character
- \w - matches all alphanumeric characters and '_'
- \W - matches anything which is not \w
- \d - matches digits

Using them

The standard way to use regular expressions in python is as follows.

- Compile the expression to a patters object.
- Then the object is matched against the test.
- If successfully matches, a Match object is returned, with the relavant information.

```
>>> import re
>>> pattern = re.compile('a(a*)b')
>>> text = 'xyzaaaab3sf'
>>> matcher = pattern.search(text)
>>> print matcher.group()
aaaab
>>>
>>>
```

More in next lecture