Learning simple functional programming in Python
This exercise is a prequel for anyone planning to do map/reduce programming in Spark, using Python.

The specific aim of this exercise is to make you familiar with *lambda* expressions, and thence onto the **map**, **reduce**, **filter** and **flatMap** concepts. If you already know these and use lambda expressions, you can ignore this exercise.

Exercise setup
Start the VM.

Open a terminal window, and type

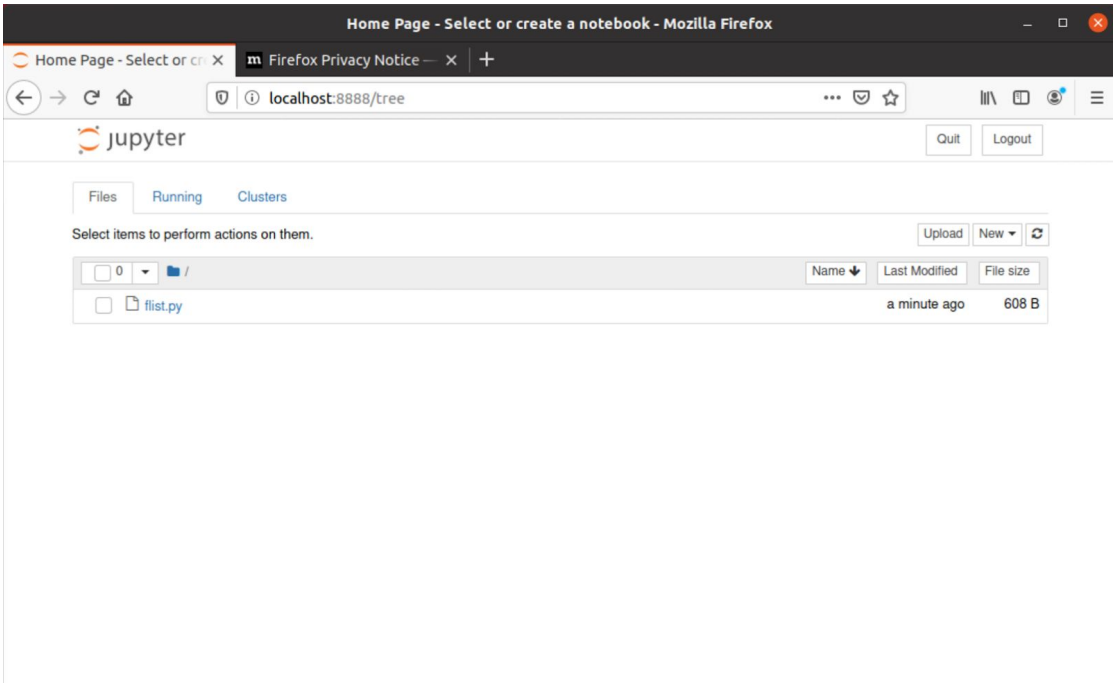
```
mkdir ~/lambda
cd ~/lambda
wget http://freo.me/oxclo-flist -O flist.py
```

This file is some simple Python *syntactic sugar*. Basically it makes the syntax of our exercises look more like the Spark syntax and less like the default Python syntax. *You need this file in the same directory where you start Python from.*

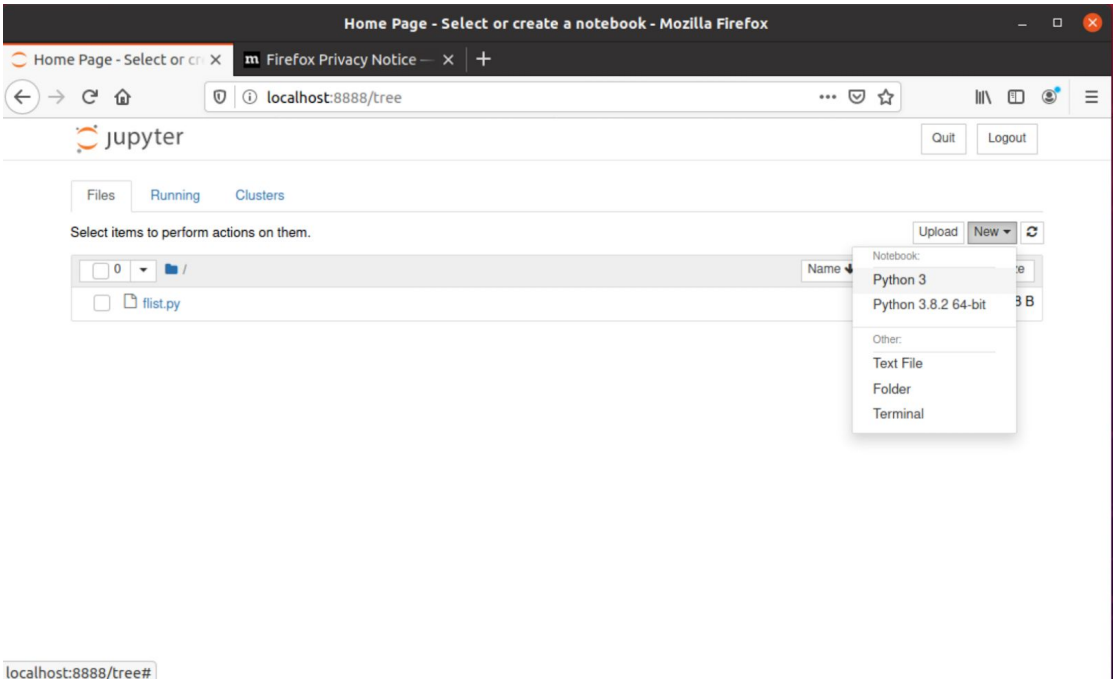
Start Python using Jupyter Notebooks - type:

```
jupyter notebook
```

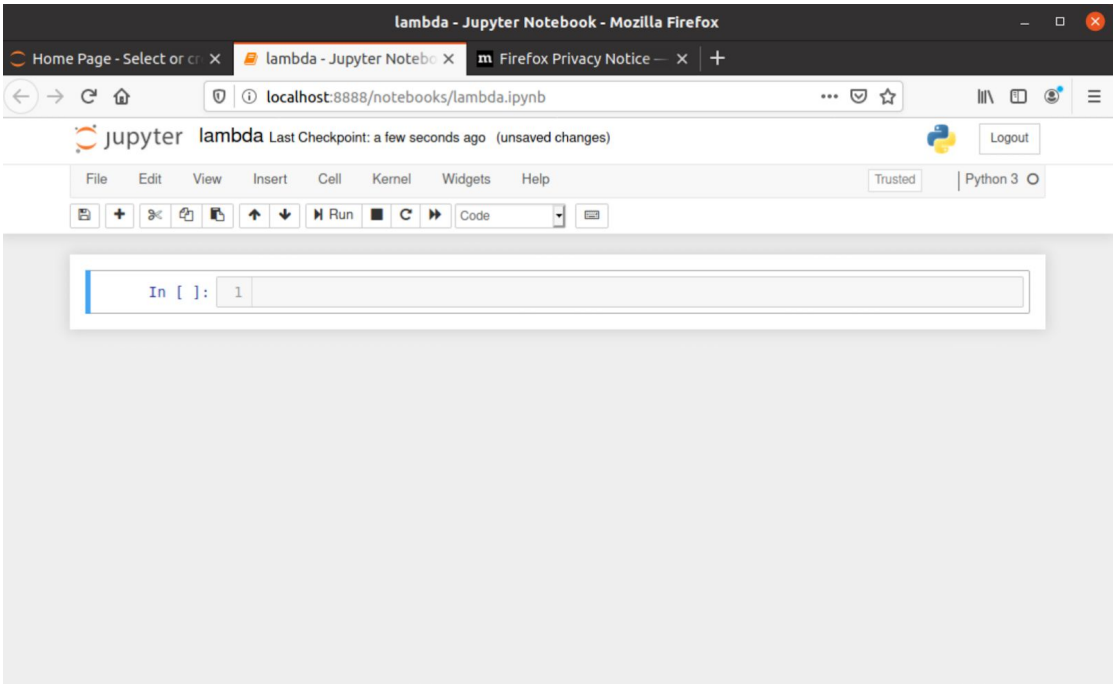
You should see:



Create a new Python3 file:



You should see something like:



Click on the Untitled name and rename to lambda like I have.

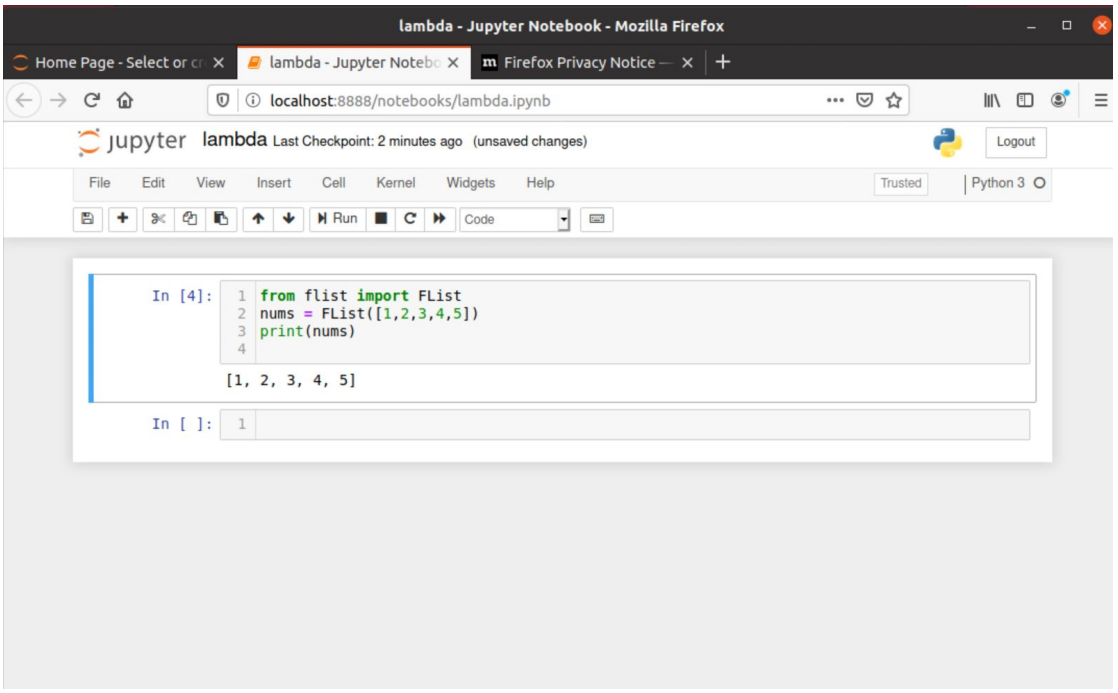
Exercise Steps

All the exercise steps take place in the Python command prompt, and assume you have successfully imported **flist.py**.

FList is just a list.

Type the following into a cell and run it:

```
from flist import FList
nums = FList([1,2,3,4,5])
print(nums)
```



We can define a function double. (in a new cell)

```
In [5]:
def double(x):
    return 2*x
```

The **map** function is a *meta-function*: it takes a function as an argument, and applies it to the list. Try this:

```
In [6]:
def double(x):
    return 2*x

print (nums.map(double))
[2, 4, 6, 8, 10]
```

In pseudo-code we can say that:
[n1,n2,n3].map(double) == [double(n1), double(n2), double(n3)]

The **filter** function is another meta-function. **filter** decides whether to include an element in the list based on the result of calling the function that you pass in. If the function evaluates to **True,** then it keeps the element. Otherwise it removes it. Let's see filter in action. In a new cell try:

```
In [8]:
def even(x):
    return x%2==0

print (nums.filter(even))
```

```
[2, 4]
```

As you can see this approach leads to very expressive code.

However, we can make this code even more expressive if we understand the concept of a lambda. Lambdas are a concept that pre-dates physical computers and goes back to the thinking of a brilliant mathematician called Alonzo Church who formulated the *lambda calculus* in the 1930s.

A lambda is simply an *unnamed* function. Suppose we want a function that returns the square of its input. Type this an execute it.

```
In [9]: lambda x: x*2
Out[9]: <function __main__.<lambda>(x)>
```

As you can see Python believes this to be a function. We can apply that function.

```
In [10]: (lambda x: x*2)(2)
Out[10]: 4
```

Guess what? This lambda is equivalent to our previous function **double.**

Now we can redo our "double ever number in the list"

```
In [11]: nums.map(lambda x:x*2)
Out[11]: [2, 4, 6, 8, 10]
```

Why would we use this instead of defining double as a named function?

The main reasons are that it is more **compact**, and the code is more **self-expressive**. When you start using lambdas you might not appreciate this, because initially it can be confusing, and therefore less readable. But once lambdas become ingrained and hence you can understand them easily, this syntax becomes more readable, because everything is captured right there.

We can also chain these:

```
In [12]: 1 nums.map(lambda x:x*2).filter(lambda x: x%2==0)
Out[12]: [2, 4, 6, 8, 10]
```

(Suprisingly if you double a number, the result is always even!)

We can quickly create new functions. For example, if we wanted all the *even squares*:

```
In [13]: 1 nums.map(lambda x:x*x).filter(lambda x: x%2==0)
Out[13]: [4, 16]
```

Suppose we wanted to add up all the squares.
First we need the list of squares:

```
In [15]: 1 squares = nums.map(lambda x: x*x)
         2 squares
Out[15]: [1, 4, 9, 16, 25]
```

In a procedural language, the normal approach is to use a loop: create a variable **total** and then add each to the total. That isn't very *functional,* because functions don't have variables. More importantly, it has state (the loop counter and the total). State is the enemy of distribution.

Instead, we can have a function that is applied to elements in the list, but instead of returning an element, it returns an accumulator, and then applies this to the next element. In general this technique is called **folding**. Specifically, we call this the **reduce** function. Another way of thinking of reduce is to imagine putting an associative operator *between* the elements of the list. So for example if we wanted to add up the list:

[1, 4, 9, 16, 25]

we simply need to put the + operator between each entry:

[1 + 4 + 9 + 16 + 25]

The plus operation can be defined in a simply lamba:

lambda x, y: x+y

Let's try that:

```
In [12]: 1 squares = nums.map(lambda x: x*x)
         2 squares.reduce(lambda x,y: x+y)
Out[12]: 55
```

We don't just have to use numbers with these meta-functions. Suppose we have two sentences and we want the individual words:

```
In [13]:
1 mystr = FList(['the quick brown fox', 'jumped over the lazy dog'])
2 mystr.map(lambda x: x.split())
3

Out[13]: [['the', 'quick', 'brown', 'fox'], ['jumped', 'over', 'the', 'lazy', 'dog']]
```

This is cool, but you might see an issue here. We have an array of arrays. We might just want a single array. There is a functional pattern called **flattening** that does this.

```
In [14]:
1 mystr = FList(['the quick brown fox', 'jumped over the lazy dog'])
2
3 mystr.map(lambda x: x.split()).flatten()

Out[14]: ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
```

Usually the flattening is needed because of a map. Hence flatMap, which lets us do it all in one go:

```
In [15]:
1 mystr.flatMap(lambda x: x.split())

Out[15]: ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
```

That should be enough lambdas and meta-functions to get us started.

Congratulations on completing the first exercise.