

# Big Data Engineering in the Cloud

## Pre-course exercise

### Learning simple functional programming in Python

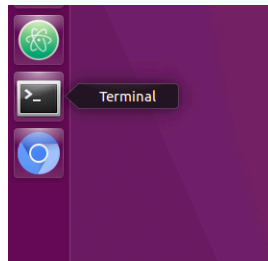
*This exercise is a prequel for anyone planning to do map/reduce programming in Spark, using Python.*

The specific aim of this exercise is to make you familiar with *lambda* expressions, and thence onto the map, reduce, filter and flatMap concepts. It also validates that the VirtualBox VM is working and that you can run *Jupyter*.

### Exercise setup

This exercise assumes that you have successfully downloaded and started the course VirtualBox VM. If you need to log in, the userid/password are **big/big**.

Start a new command shell, by clicking the Terminal icon on the left hand side-bar:



In that window, type:

```
mkdir precourse
cd precourse
wget https://freo.me/oxclo-flist -O flist.py
ls -l
```

You should see something similar to

```
big@big: ~/precourse
File Edit View Search Terminal Help
big@big:~$ mkdir precourse
big@big:~$ cd precourse
big@big:~/precourse$ wget https://freo.me/oxclo-flist -O flist.py
--2017-11-24 10:34:13-- https://freo.me/oxclo-flist
Resolving freo.me (freo.me)... 67.199.248.12, 67.199.248.13
Connecting to freo.me (freo.me)[67.199.248.12]:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://raw.githubusercontent.com/pzfreo/ox-clo/master/code/flist.py [
following]
--2017-11-24 10:34:14-- https://raw.githubusercontent.com/pzfreo/ox-clo/master/
code/flist.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.60.13
3
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)[151.101.60.1
33]:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 608 [text/plain]
Saving to: 'flist.py'

flist.py          100%[=====]          608  --.-KB/s   in 0s

2017-11-24 10:34:15 (157 MB/s) - 'flist.py' saved [608/608]

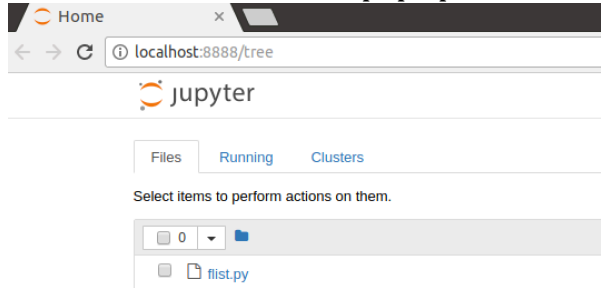
big@big:~/precourse$ ls -l
total 4
-rw-rw-r-- 1 big big 608 Nov 24 10:34 flist.py
```

# Big Data Engineering in the Cloud

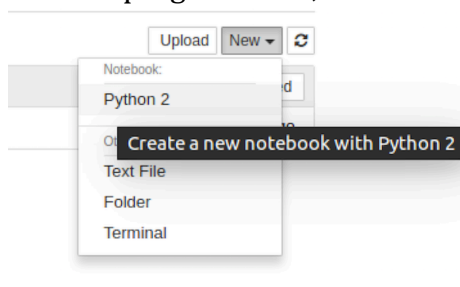
## Pre-course exercise

Now type:  
`jupyter notebook`

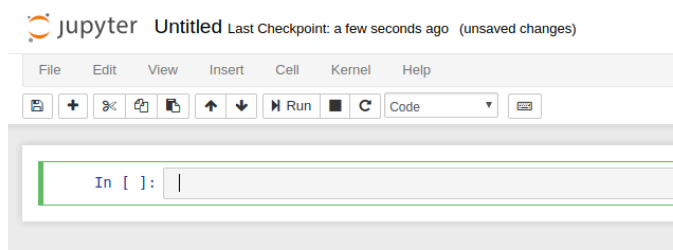
A browser window should pop open:



In the top right corner, click on New->Python 2



This will open a new tab, with a Python Notebook (this is a way of editing and running Python in a browser window that will be used extensively during the course). You should see:



Click on the word **Untitled**, and change the name of the notebook to **precourse**. You should be able to hit Ctrl-S at any time to save. You can find further keyboard mappings here:

<https://www.cheatography.com/weidadeyue/cheat-sheets/jupyter-notebook/>

# Big Data Engineering in the Cloud

## Pre-course exercise

### Exercise Steps

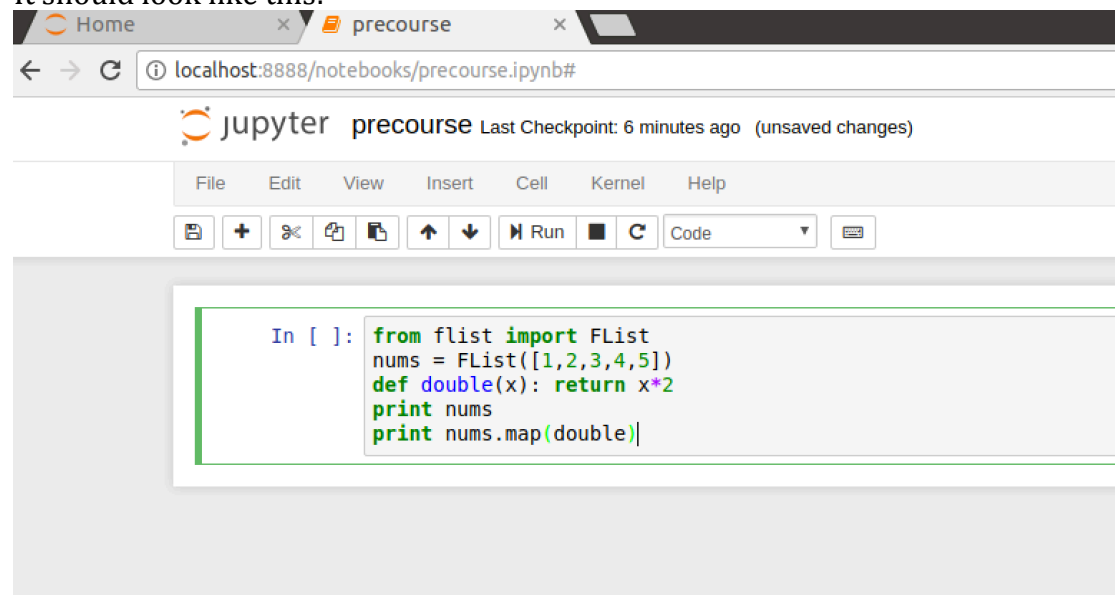
For this exercise, we will be using a simple Python class **FList**.

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

In the cell type:

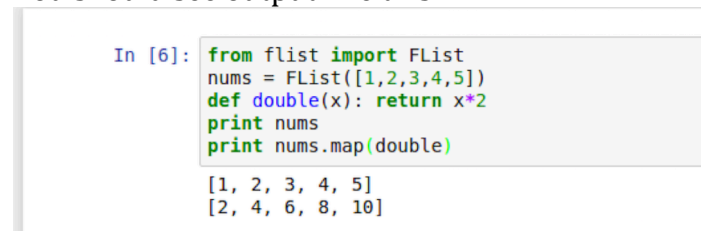
```
from flist import FList
nums = FList([1,2,3,4,5])
def double(x): return x*2
print nums
print nums.map(double)
```

It should look like this:



Now click the Run icon.

You should see output like this:



# Big Data Engineering in the Cloud

## Pre-course exercise

Let's review what is happening there.

FList is just a list, very similar to a normal Python list, but with slightly different behaviour for map, filter, flatMap and reduce.

Double is just a function that returns twice its input.

```
def double(x): return x*2;
```

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

```
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 the next cell, type

```
def even(x): return x%2==0
print nums.filter(even)
```

When you run this cell you should see:

```
[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.

```
lambda x: x*2
```

If you run this, Python will tell you it's a function:

```
lambda x: x*2
<function __main__.<lambda>>
```

# Big Data Engineering in the Cloud

## Pre-course exercise

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

```
(lambda x: x*2)(2)
```

And run it

```
(lambda x: x*2)(2)  
4
```

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

Now we can redo our “double every number in the list”

Run the following:

```
nums.map(lambda x: x*2)
```

```
nums.map(lambda x: x*2)  
[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:

```
nums.map(lambda x: x*2).filter(lambda x: x%2==0)  
[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*. Try this:

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

Now suppose we wanted to add up all the squares from 1 to 5. First we need the list of squares. Try this:

```
squares = nums.map(lambda x: x*x)
print squares
```

```
squares = nums.map(lambda x: x*x)
print squares
```

```
[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 scalability** (as we will find out in the course).

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 lambda:

```
lambda x, y: x+y
```

Let's try that. In a new cell type:

```
print squares.reduce(lambda x,y: x+y)
```

```
print squares.reduce(lambda x,y: x+y)
```

```
55
```

# Big Data Engineering in the Cloud

## Pre-course exercise

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

```
str = FList(['the quick brown fox', 'jumped over the lazy dog'])
print str.map(lambda x: x.split())
```

```
str = FList(['the quick brown fox', 'jumped over the lazy dog'])
print str.map(lambda x: x.split())

[['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.

```
print str.map(lambda x: x.split()).flatten()
```

```
print str.map(lambda x: x.split()).flatten()

['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:

```
print str.flatMap(lambda x: x.split())
```

```
print str.flatMap(lambda x: x.split())

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

That should be enough lambdas and meta-functions to get us started. To recap, we have covered a little bit of *map*, *filter*, *flatten*, *flatMap*, *reduce* and *lambda*.

You can close the browser window. Then click back on the terminal window and hit Ctrl-C. You should see:

```
Shutdown this notebook server (y/[n])?
```

Type **y** and hit Enter. You can now type **exit** to close the Terminal window.

*Congratulations on completing the first exercise.*