Python Primer

March 1, 2016

1 Python

1.1 Lists and Tuples

1.1.1 Indexing into list

```
In [1]: l = [1, 2, 3] # make a list

        l[1] # index into it
```

```
Out[1]: 2
```

1.1.2 Appending to a list

```
In [2]: l.append(4) # add to it
        l
```

```
Out[2]: [1, 2, 3, 4]
```

1.1.3 Deleting an element

```
In [3]: del l[1]
        l
```

```
Out[3]: [1, 3, 4]
```

1.1.4 Inserting an element

```
In [4]: l.insert(1, 3) # insert into it
        l
```

```
Out[4]: [1, 3, 3, 4]
```

1.1.5 Tuples

```
In [5]: t = (1, 3, 3, 4) # make a tuple

        l == t
```

```
Out[5]: False
```

1.1.6 List to Tuple

```
In [6]: t2 = tuple(l)
        t2 == t
```

```
Out[6]: True
```

1.2 Dictionaries

```
In [7]: Dict = {}
        Dict[1] = 2
        Dict['one'] = 'two'
        Dict['1'] = '2'
        Dict

Out[7]: {1: 2, '1': '2', 'one': 'two'}
```

1.2.1 Keys in Dictionary

```
In [8]: print "Dictionary keys"
        print Dict.keys()

        print "\nValue at 1 :"
        print Dict['1']

        print "\nValue at one"
        print Dict['one']


        one = 1
        print "\nValue at 1"
        print Dict[one]

        print "\nIterate over keys"
        for key in Dict.keys():
            print key

        print "\nDelete key : 1"
        del Dict[1]
        print Dict

Dictionary keys
['1', 1, 'one']

Value at 1 :
2

Value at one
two

Value at 1
2

Iterate over keys
1
1
one

Delete key : 1
{'1': '2', 'one': 'two'}
```

2 Classes and Function

2.1 Functions

In [9]:
```
def printer(x):
    print x

def adder(x,y):
    return x+y

def square(x):
    return x**2

a = 2
b = 3
print "Lets print a:"
printer(a)
print "\nLets print a + b"
printer(adder(a,b))
print "\n So you can pass the return of a function to another function just like everywhere. \n
printer(square(adder(a,b)))
```

```
Lets print a:
2

Lets print a + b
5

 So you can pass the return of a function to another function just like everywhere.
 Lets take it another step further
25
```

2.2 Classes

In [10]:
```
class student(object):

    def __init__(self,name = None ,age = None):
        if name == None:
            self.name = "Amartya"
        else:
            self.name = name

        if age == None:
            self.age = 20
        else:
            self.age = age

    def update_name(self,name):
        self.name = name

    def update_age(self,age):
        self.age = age

    def inc_age(self):
        self.age = self.age + 1
```

```
            def return_info(self):
                temp = [self.name, self.age]
                return temp
```

```
In [11]: Amartya = student()
         print"Amartya:"
         print vars(Amartya)

         Bhuvesh = student("Bhuvesh", 21)

         print "\nBhuvesh:"
         print vars(Bhuvesh)

         print "\nIncrementing Bhuvesh's age"
         Bhuvesh.inc_age()
         print vars(Bhuvesh)

         print "\nMake Amartya  a baby"
         Amartya.update_age(1)
         print vars(Amartya)

         print "\nA list of attributes of Amartya(Just to show what lists are)"
         print Amartya.return_info()
```

```
Amartya:
{'age': 20, 'name': 'Amartya'}

Bhuvesh:
{'age': 21, 'name': 'Bhuvesh'}

Incrementing Bhuvesh's age
{'age': 22, 'name': 'Bhuvesh'}

Make Amartya  a baby
{'age': 1, 'name': 'Amartya'}

A list of attributes of Amartya(Just to show what lists are)
['Amartya', 1]
```

3 Exceptions

```
In [12]: print "Adding 2 and 3"
         printer(adder(2,3))

         print "\nAdding 'Amartya' and 'Bhuvesh'"
         printer(adder("amartya","bhuvesh"))

         print "\nBut say we want to practical and only add numbers , not people."

         def adder(x,y):
             try:
                 if type(x) != 'int' or type(x) != 'float' or type(y) != 'int' or type(y) != 'float':
                     raise ValueError()
```

```
                else:
                    return x+y
            except ValueError:
                print "Error!! Error!! You cant add people\n"

        print "\nAdding 'Amartya' and 'Bhuvesh'"
        printer(adder("amartya","bhuvesh"))
```

```
Adding 2 and 3
5

Adding 'Amartya' and 'Bhuvesh'
amartyabhuvesh

But say we want to practical and only add numbers , not people.

Adding 'Amartya' and 'Bhuvesh'
Error!! Error!! You cant add people

None
```

4 Starting Numpy

```
In [13]: import numpy as np #Please don't forget this
```

4.1 Basic types of arrays and matrices

4.1.1 Zero Array and Zero Matrix

```
In [14]: zeroArray = np.zeros(5)
         print "Zero Array"
         print zeroArray
         print "\nZero Matrix:"
         zeroArray = np.zeros([5,10])
         print zeroArray
```

```
Zero Array
[ 0.  0.  0.  0.  0.]

Zero Matrix:
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

4.1.2 Ones array and Ones Matrix

```
In [15]: oneArray = np.ones(5)
         print "Ones Array"
         print oneArray
         print "\nOnes Matrix:"
         oneArray = np.ones([5,10])
         print oneArray
```

```
Ones Array
[ 1.  1.  1.  1.  1.]

Ones Matrix:
[[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]]
```

4.1.3 Identity Matrix

```
In [16]: I = np.identity(5)
         print "Identity Matrix"
         print I
```

```
Identity Matrix
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
```

4.1.4 Basic vector stuff

```
In [17]: A = [1, 2, 3]
         B = np.asarray(A)
         C = [4,5,6]
         D = np.asarray(C)
```

```
In [18]: print "Elementwise Multiplication"
         print B*D
         print "\nElementwise Addition"
         print B+D
         print "\n Dot Product"
         print np.dot(B,D)
```

```
Elementwise Multiplication
[ 4 10 18]

Elementwise Addition
[5 7 9]

 Dot Product
32
```

```
In [19]: print "Lets square each element in the array"
         print [x**2 for x in C]
         print "\n Lets do some more complicated function"

         def updateX(x):
             x = x + 2
             x = np.log(x)
             x = np.power(x,2)
             return x

         print [updateX(x) for x in C]
```

```
Lets square each element in the array
[16, 25, 36]

 Lets do some more complicated function
[3.2104019955684011, 3.7865663081964716, 4.3240771252638117]
```

4.1.5 Useful stuffs that make your life easy when coding stuffs.

```
In [20]: print "Createing an array of numbers from 1 to 9"
         A = np.arange(1,10)
         print A

         print "\n Reshape an array to matrix"
         B = np.reshape(A,[3,3])
         print B

         print "\n Transpose the matrix"
         C = np.transpose(B)
         print C

         print "\n Make elements less than 5 0"
         C[C<5] = 0
         print C
```

```
Createing an array of numbers from 1 to 9
[1 2 3 4 5 6 7 8 9]

 Reshape an array to matrix
[[1 2 3]
 [4 5 6]
 [7 8 9]]

 Transpose the matrix
[[1 4 7]
 [2 5 8]
 [3 6 9]]

 Make elements less than 5 0
[[0 0 7]
 [0 5 8]
 [0 6 9]]
```

```
In [21]: print "Summing up elements"
         print "\n Each column"
         print np.sum(C,axis=0)
         print "\n Each row"
         print np.sum(C,axis=1)
```

```
Summing up elements

 Each column
[ 0 11 24]

 Each row
[ 7 13 15]
```

```
In [22]: print "Mean of elements"
         print "\n Each column"
         print np.mean(C,axis=0)
         print "\n Each row"
         print np.mean(C,axis=1)
```

```
Mean of elements

 Each column
[ 0.          3.66666667  8.        ]

 Each row
[ 2.33333333  4.33333333  5.        ]
```

```
In [23]: print "Product of  elements"
         print "\n Each column"
         print np.prod(C,axis=0)
         print "\n Each row"
         print np.prod(C,axis=1)
```

```
Product of  elements

 Each column
[  0   0 504]

 Each row
[0 0 0]
```

5 Finally Theano!

```
In [24]: import theano
         import theano.tensor as T
```

```
In [25]: # Create the scalars
         x = T.scalar()
         y = T.scalar()
```

```
In [26]: print "Add two numbers"
         temp1 = x + y
         # So this is how you add two "Symbolic variables"

         addTh = theano.function([x,y],temp1)
         theano.pp(addTh.maker.fgraph.outputs[0])
```

```
Add two numbers

Out[26]: '(<TensorType(float64, scalar)> + <TensorType(float64, scalar)>)'
```

```
In [27]: print addTh(1,2)
```

```
3.0
```

```
In [28]: print "Comparing two numbers"

         temp1 = T.le(x, y)
         compTh = theano.function([x,y],temp1)
```

```
        theano.pp(compTh.maker.fgraph.outputs[0])
        print compTh(4,3)
```

```
Comparing two numbers
0
```

```
In [29]: print "If else operator in Theano"
         xgy = T.ge(x,y)
         res = 2*x*xgy + (1 - xgy)*3*x


         ifelse = theano.function([x,y],res)
         print ""
         print theano.pp(compTh.maker.fgraph.outputs[0])
         print ""
         print ifelse(5,4)
```

```
If else operator in Theano

le(<TensorType(float64, scalar)>, <TensorType(float64, scalar)>)

10.0
```

```
In [30]: #Create the symbolic graph
         z = x + y
         w = z * x
         a = T.sqrt(w)
         b = T.exp(a)
         c = a ** b
         d = T.log(c)

         uselessFunc = theano.function([x,y],d)
         theano.pp(uselessFunc.maker.fgraph.outputs[0])
```

```
Out[30]: 'Elemwise{Composite{log((Composite{sqrt(((i0 + i1) * i0))}(i0, i1) ** exp(Composite{sqrt(((i0 +
```

```
In [31]: print uselessFunc(1,4)
```

```
7.52932798092
```

5.1 Where's the vector stuff

```
In [32]: x = T.vector('x')
         y = T.vector('y')

         A = np.asarray([1,2,3])
         B = np.asarray([4,5,6])
```

```
In [33]: xdoty = T.dot(x,y)
         xaddy = T.sum(x+y)
         dotfn = theano.function([x,y], xdoty)
         print "Lets do dot product in theano"
         print A,B,dotfn(A,B)

         print "\nFunctions with more than one outputs"
```

```
        dotaddfn = theano.function([x,y], [xdoty,xaddy])

        print dotaddfn(A,B)
        print "\n All element wise operations are similar to numpy"
```

```
Lets do dot product in theano
[1 2 3] [4 5 6] 32.0

Functions with more than one outputs
[array(32.0), array(21.0)]

 All element wise operations are similar to numpy
```

5.1.1 The famous logistic function

```
In [34]: x = T.matrix('x')
         s = 1 / (1 + T.exp(-x))
         logistic = theano.function([x], s)

         print theano.pp(logistic.maker.fgraph.outputs[0])
         logistic([[0, 1], [-1, -2]])
```

```
sigmoid(x)
```

```
Out[34]: array([[ 0.5       ,  0.73105858],
                [ 0.26894142,  0.11920292]])
```

5.2 The update comes in

```
In [35]: state = theano.shared(0)
         inc = T.iscalar('inc')

         #Update the state by incrementing it with inc
         accumulator = theano.function([inc], state, updates=[(state, state+inc)])
```

```
In [36]: for i in range(0,10):
             accumulator(i)
             # In order to get the value of the accumulated
             print state.get_value()

         # We can also set the value of a shared variable
         state.set_value(0)
```

```
0
1
3
6
10
15
21
28
36
45
```

5.3 As you might have guessed ML is a lot about updating parameters to achieve lowest cost

5.4 But then we need to choose what to update it with

5.5 Gear up for some magic

5.6 Gradient Magic

In [37]:
```
a = T.scalar('a')
b = T.sqr(a)
c = T.grad(b,a)

gradfn = theano.function([a],c)
print theano.pp(gradfn.maker.fgraph.outputs[0])

print gradfn(4)
```

```
(TensorConstant{2.0} * a)
8.0
```

In [38]:
```
B = theano.shared(np.asarray([1.,2.]))
R = T.sqr(B).sum()
A = T.grad(R, B)

Z = theano.function([], R, updates={B: B - .1*A})
for i in range(10):
    print('cost function = {}'.format(Z()))
    print('parameters    = {}'.format(B.get_value()))
# Try to change range to 100 to see what happens
```

```
cost function = 5.0
parameters    = [ 0.8  1.6]
cost function = 3.2
parameters    = [ 0.64  1.28]
cost function = 2.048
parameters    = [ 0.512  1.024]
cost function = 1.31072
parameters    = [ 0.4096  0.8192]
cost function = 0.8388608
parameters    = [ 0.32768  0.65536]
cost function = 0.536870912
parameters    = [ 0.262144  0.524288]
cost function = 0.34359738368
parameters    = [ 0.2097152  0.4194304]
cost function = 0.219902325555
parameters    = [ 0.16777216  0.33554432]
cost function = 0.140737488355
parameters    = [ 0.13421773  0.26843546]
cost function = 0.0900719925474
parameters    = [ 0.10737418  0.21474836]
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: