Learning Python for CS119

Python Overview

Python is an easy to learn object orientated and functional language with intuitive syntax and a small barrier to development. There is no need to specify the object type or generic object parameter ever making defining objects very simple. Installing new libraries to import into your own module can be done using PIP (Python Installs Packages) in the command line.

Topic Index

1. Functions, objects, and modules
2. Iteration, conditionals, other logic
3. Data structures & object manipulation
4. Lambdas, list comprehensions, and mapping
5. Statistics, plotting, and graphing
6. Multithreading, callbacks, and generators

Functions, Objects, and Modules

A module is a single python file and functions can be defined at the module level as shown below.

```
def foo(x):
    return x ** 2

def bar(x):
    return foo(x) + foo(x)

def foobar(x):
    y = bar(x)
    return 4 * y
```

Multiple objects can be defined within modules and whole modules or objects/functions can be imported from modules. Pay attention to the use of SELF for object fields.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def getX(self):
        return self.x

    def getY(self):
        return self.y
```

Importing modules is very easy and you can specify what objects/functions you want to import.

```
import random
from math import sin, cos, radians, degrees, pi

a = sin(degrees(pi * random.randrange(0, 4)))
```

Iteration, Conditionals, and Other Logic

You can FOR to iterate through an iterable object or range of integers and you can use WHILE to do condition based iteration. You can also use the BREAK and CONTINUE statements. I use PRINT here to print text to the command line.

```
for x in range(0, 100):
    print(x)

i = 0
while i < 100:
    print(i)
    i += 1
```

To do define a list or tuple use the square brackets or parentheses. I use the Python defined length function LEN( ) to grab the objects length.

```
small_list = [1, 2, 3, 4, 5]

small_tuple = ("one", "two")

for item in small_list:
    print(item)

for i in range(0, len(small_tuple)):
    print(small_tuple[i])
```

Here is some example conditional logic with IF, IF ELSE, and ELSE.

```
a = 3
if a <= 0:
    print("0")
elif a < 6:
    print(a)
elif 1 != 12:
    print("not 12")
else:
    print("6+")
```

Data Structures and Object Manipulation

Python provides the following data structures:

Python name	DS equivalent	Memory Allocation
list	Linked List	Dynamic
tuple	Array	Static
dictionary	Hash Table	Dynamic

Here I create some DS's with items already in them.

```
# list
a = [1, 2, 3]

# tuple
b = ("first", "second")

# dictionary
c = {"a" : "book",
     "b" : "table",
     "c" : "fan"}
```

You can add and remove items from lists and dictionaries but not tuples because they are immutable. Here I show how to index and add/remove items. Below I use the list object methods to append an item to add 4 to the list and then print it out. You can index tuples the same way. Dictionaries can be also indexed using square brackets to specify for what key you want to get the value for.

```
a.append(4)
print(a[3])
print(b[1])
c["d"] = "floor"
print(c["d"])
```

Lambdas, List Comprehensions, and Mapping

Lambdas are anonymous functions which can be used to greatly simplify sections of code and avoid repetition. The lambda function below adds the input to it's square.

```
function = lambda x: x + x ** 2
solution = function(5)
```

List comprehensions are also an easy way to clear up messy code with many iterations. List comprehensions can also be nested but that is not always recommended. Here I create a LIST of the even numbers 0 - 100.

```
sample = [x for x in range(0, 100) if x % 2 == 0]
```

It is not necessary to use the IF statement in a list comprehension and list comprehensions can be used to create TUPLES and DICTIONARIES. Here I create a tuple that has every element from the sample list + 2 and a sample dictionary with every element from the sample tuple as the key and every element cubed as the respective values.

```
sample_tuple = (y + 2 for y in sample)
sample_dict = {z : z ** 3 for z in sample_tuple}
```

You can iterate through a dictionary key : value pair using multiple indexing. It's useful when performing actions on a dictionary.

```
for key, value in sample_dict.items():
    print("Key: " + str(key) + " Value: " + str(value))
```

Mapping can be used to transform an iterable to a new iterable of the same length by specifying a function and iterable . The map will return a new iterable where each element corresponds to the function(each element) in the input iterable.

```
my_points_x1 = map(Point.getX, my_points)
my_points_x2 = map(lambda p: p.x, my_points)
def getPointX(p): return p.x
my_points_x3 = map(getPointX, my_points)
```

Statistics, Plotting and Graphing

Import pyplot from matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import math
```

Here I plot a line by providing lists of x and y coordinates of a segmented line.

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.ylabel('some numbers')
plt.show()
```

Here I show a normal curve specified by mean, variance, and parameters for the resolution of the curve and width of the plot.

```
mean = 0
variance = 1
plot_width = 3
resolution = 100
std_dev = math.sqrt(variance)
x = np.linspace(mean - plot_width * std_dev, mean + plot_width * std_dev,
resolution)
plt.plot(x, stats.norm.pdf(x, mean, std_dev))
plt.show()
```

We can use list comprehensions or the RANGE function to create a list of x and y coordinates to plot.

```
x = np.array(range(0, 100))
y = x ** 2
plt.plot(x, y)
plt.show()
```

Multithreading, Callbacks, and Generators

Adding concurrency to code is easy to do using the thread package.

```
import thread
import time
```

The parameters for starting a new thread are a function to execute and the parameters for the function.

```
# thread.start_new_thread(function, args[, kwargs])
a = thread.start_new_thread(print, "thread1")
a.start() # OR a.run()
print(a.isAlive())
a.setName("abc")
print(a.getName())
```

Python's functionality and easy to use multithreading allows for an easy implementation of callbacks.

```
# callback function
def checkSum(a, b):
    if (a + b) % 10 == 3:
        return True
    return False

# calls callback function
def doStuff(func):
    i = 0; j = 0
    while True:
        print(func(i, j))
        i += 1
        j += 1
```

An example generator that uses the YIELD function instead of return and allows for parameters to be used in the function. Generators are useful for creating a general method that you want to use differently in multiple places. Generators can be used similar to iterables (lists, tuples, and dictionaries).

```
def gen(n):
    num = 0
    while num < n:
        yield num
        num += 1

for i in gen(10):
    print(i)
```