**http://nbviewer.jupyter.org/github/jmportilla/Complete-Python-Bootcamp/tree/master/**

**Comments:**

One line: #

Multiple lines: ‘’’ frtgyhujn ‘’’ (3 single quotes)

**Data Types/ Literals:**

**1. int**

i=int, i= None

i=6

**2. str**

string=str, string=None

string= "abcdef" or ‘abcdef’ , both quotes acceptable!

**3. float**

decimal=float

**4. bool literals**

True= True, False=False

**5. List (Basically a user-friendly Hybrid Array, duplication of data allowed)**

**Integers**: lst=[1,2,3,4], lst.append(24) gives lst= [1,2,3,4,24]

**Strings:** strList=['abc' , 'def', 'lij' ], strList.append('23')

**Hybrid:** hybridList=['abc', 34.232100, 23, 2/3]

**6. Tuples (An IMMUTABL list which can be appended to but not deleted from or its elements edited, used to return multiple elements)**

Empty Tuple Initialization: tupp=()

tupp= (1, ‘abc’, 7/3)

Adding elements to a tupple: tupp= tupp + (5, ‘def’)

Accessing a Tuple’s individual element: tupp[3]

Accessing a Tuple’s group of elements: tupp[1:3] # 1 to 3-1=2

tupp[1:2] outputs (‘abc’,) # the comma after ‘abc’ represents the fact that this tuple has only one element

Note: An attempt to modify a tupple gives an error: tupp[2]=9 will give an error

Example of Using Tupples to return multiple elements:

**def quotient\_and\_remainder(x, y):**

**q = x // y**

**r = x % y**

**return (q, r)**

**def main():**

**(quot, rem) = quotient\_and\_remainder(4,5)**

**7. Dictionaries/ Hash Table with Hybrid Data Storage (Mutable, elements are accessible with the keys)**

All String:

my\_dict={'key1':'value1','key2':'value2'}

my\_dict['key1']= 'value1' my\_dict['key2']= 'value2'

Hybrid Data:

my\_dict ={'k1':123,'k2':10.242,'k3':'hello'}

my\_dict['k1']=12 makes it 12

Appending to Dictionary or Adding elements to it: my\_dict[‘New Key’]= ‘New Data’

Create Dictionaries using Dictionary Comprehension:

diction= { x: x\*\*2 for x in range(10) }, x becomes key and x\*\*2 becomes the data

diction= { x+2: x\*2 for x in [1,2,3,4,5,6] }, x+2=key and x\*2=data

len(d) gives the length

**8. Sets** (set1={1,2,3}, set2={2,4,1} ) ( unique values only)

set1= set()

set1.add("cat")

set1.update(["dog","lion"])

set1.remove ("mouse") # works only if “mouse” is a member of set1

set1.discard(“mouse”) # removes “mouse” if it’s a member, else it ignores the command

set1.clear() # removes all elements from the set

set2=set1.copy() # set2=copy of set1. if you do set2=set1, they point to the same data and changing set2, changes set1 as well.

**set2 =** **frozenset(set1) (An immutable set that can be used as a dictionary)**

**Common initializer: ‘None’**

i=data\_Type followed by i=None

**Initialization: (No data\_type declaration is required)**

x = 5 y = True z = 2.71828

**Arithmetic Operations:**

Addition: +

Subtraction: -

Multiplication: \*

float division: / (result is floating)

Integer division: // (result it integral)

Modulus: %

Exponentiation: \*\* (base \*\* exponent)

**Bool Operations:**

**and, or, not**

print (1 or 0) gives 1

print (0 and 1) gives 0

print (not 1) gives False

print (1 < 3) prints True

**Comparisons:**

**X== None, y is None**

**str (string operations):**

**Concatenation:** str3= 'abc' + ‘def’

**Repetition**: str2=’a’\*5 means repeat that string 5 times, making str2: ‘aaaaa’

**Slicing / Selective access:**

**1. Positive indices:** 'abcdefghi'[3:6] says include from position 3 to 6-1=5. Hence, ‘def ’

**2. Negative Index:** Negative index indicates ‘from the end/back’. So ‘abcdefghi’[3:-2] says include from 3 to 2nd position from end. Notice that it’s 2nd from the end, not 2-1=1 from the end.

So, ‘abcdefghi’[3:-2]= ‘defg’

**Reversing a string: s[::-1]** gives reversed string

**Capitalize first word in string:** string.capitalize()

**Lower to upper case**: string.upper()

**Upper to lower**: string.lower()

**Find the number of times a substring appears**: string.count(‘<substring>’)

**Find index at which substring appears:** string.find(‘<substring>’)

Check for: 1. Alpha-numeric: string.isalnum()

2. Alphabetic only: string.isalpha()

3. all lower case: string.islower()

4. all upper: string.isupper()

5. all whitespace: string.isspace()

6. Titled Case: string.istitle()

7. If string ends with a particular substring: string.endswith(‘<substring>’)

**Split string by a substring:** string.split(‘substring’) # ‘hello’.split(‘ll’) returns [‘he’,’o’]

**Partition string into parts with the substring isolated**: string.partition(‘substring’) # ‘hello’.partition(‘e’) returns [‘h’,’e’,’llo’]

**List Operations:**

list1= [1,2,4,5,5, 'abc', 3.7, 3/7]

1. Direct Access/ Indexing: list1[3]= 5

2. Slicing: list1[3:7]= list1[3:7]= [5, 5, 'abc', 3.7]

3. Append: list1.append(24) only adds a single element.

Extend: list1.extend( [ 4,5,6,7] ) will append 4,5,6, and 7 to the list.

4. Deletion: del list1[2] deletes element at index 2

Removing the 1st occurrence of an element: list1.remove(2) makes [1,2,2,3] into [1,2,3]

5. Length of List: len(list1) gives 7

6. Popping like a stack: list1.pop() removes the last element and pops it like a stack.

7. Concatenation of lists: list1=list1+list2

8. Sorting in Ascending order: list1.sort() Works with list of numerical data only.

9. Indexing: list1.index(2) gives the index number of element ‘2’ in list1 if ‘2’ is present, else it throws ValueError!

10. Insertion of an element at an Index: list1.insert(<index number>,<data\_value>) eg, list1.insert(2,’d’)

11. Reversing a list permanently: list1.reverse(), reassigns the list to its reversed version.

**List Comprehensions (Compressed for loop way of filling a list)**

lst=[]

for alphabet in 'abcdefghi':

lst.append(alphabet)

lst1=[alphabet for alphabet in 'abcdefghi']

is equivalent to

This is basically just the same as english descriptions of math functions,domains etc..

Eg: { x: x in even numbers}, { x^2: x in prime numbers} etc. where the colon stands for python’s ‘for’.

Python eg.: **list1=[2\*\*x for x in range(11)]** (remember ‘\*\*’ is the base^exponent operator) list1 gets populated with 2 raised to the power x where x varies from 0 to 10. So, 2 raised to the power 0 to 10

Python eg with Conditional: **list1=[x for x in range(21) if x%2==0]** populates list1 with all even numbers from 0 to 20.

So, you write **list1= [ output** for **variable** inrange() **condition]**

**Nested List Comprehension:**

Result of the inner list comprehension becomes the range of values for the outer list comprehension.

**Eg:** cube=[x\*\*3 for x in[x\*\*2 for x in range(6)]]. “ x\*\*2 for x in range(6) ” yields squares of numbers from 0 to 5. These square values serve as the range of values for the outer comprehension function that finds the cube of each of these values. This yields a result of [0, 1, 64, 729, 4096, 15625]

**String to List and vice-versa conversion:**

**1.** s= "10\*50= 500"

list(s) gives ['1', '0', '\*', '5', '0', '=', ' ', '5', '0', '0']

s=list(s) makes s a list with character as an element of that list. s=['1', '0', '\*', '5', '0', '=', ' ', '5', '0', '0']

**2**. lst=['1', '2', '3']

''.join(lst) gives '123’ # join gives the elements of the lst combined with the character inside ‘’ between the elements

'\_\_'.join(lst) gives '1\_\_2\_\_3'

**Sets Operations:**

**1. Length of Set/ Cardinality:** len(set2)

**2. Intersection:** set1 & set2

If 2 sets are disjoint, **set1 & set 2 = set()** (empty set).

Use **set1.isdisjoint(set2**) to check whether they’re disjoint or not.

**3. Union/ OR:** set1 | set2 or set1.union(set2)

**4. Difference:**

**1.** set1-set2, will return a proper result only if set1>=set2 otherwise it returns set() (null set)

**2.** if set1={1,2,3}, set2={2,3}. set1.symmetric\_difference(set2) and set2.symmetric\_difference(set1) will both return {1}. **So, choose symmetric\_difference**

**5. Equality:** s1 == s2

**6. Inequality:** s1 != s2

**7. Subset**: s1 <= s2

use **set1.issubset(set2)** to check whether set1 is subset of set2

**8. Proper subset**: s1 < s2

**9. Superset:** s1 >= s2

use **set1.issuperset(set2)** to check if set1 is superset of set2

**10. Proper superset**: s1 > s2

**Dictionary Operations:**

**1. Nesting of Dictionaries**: nested\_dict={‘key1’: dict1, ‘key2’:dict2, ‘key3’:dict3} where dict1,2,3 are all dictionaries.

**2**. **Iteration over a Dictionary**: dictio.keys() to iterate over keys, dictio.values() to iterate over data and dictio.items() to iterate over key-data pair that are returned as tuples

d={1: 9, 2: 8, 3: 7, 4: 6, 5: 5}

for x in d.items():

print(x) # returns (1, 9) (2, 8) (3, 7) (4, 6) (5, 5)

for x in d.keys():

print(x) # returns 1 2 3 4 5

for x in d.values():

print(x) # returns 9 8 7 6 5

**Tuples Operations:**

tupp.index(1) gives index of element ‘1’

tupp.count(1) gives the number of times ‘1’ is in the tupple

**Membership Operations:**

**in / not in (look for things in a set or list)**

5 in [1,2,3] #yield False

3 in [1,2,3] #yields True

5 not in [1,2,3] #yields True

**is / is not (compare for same memory location or not)**

a=b=2

a is b #yields True

c=2

a is c #yields False because a and c don’t have the same memory location

**id()**

returns the memory address

**Printing:**

print(‘abc’), print(i) or just write the name of the variable

Output Formatting: print(' Num1: %s\n Num2: %s\n Num3: %s\n' %(a,b,c)) where a,b, and c are variables of type str

**Input:**

**1.** Variable=input(“Give Number:”) #gives Variable a string value

(int)(Variable)+-/\* number #casting is required to do computations

**2.** num = int(input("Type a number... "))

**File I/O:**

**Reading**

**Opening Files (Option 1 or 2):**

**Tags:** r (read), w (write), w+ (write, if file doesn’t exist then create it), a (open in write-append mode), a+ (append mode, if file doesn’t exist then create it)

**1. Both Read-Write mode**

file1= open('C:\\Users\\Divyanshu Sharma\\Desktop\\file.txt')

**file1.read()** outputs the entire file’s contents with ‘\n’ newline characters too. Executing the read() function again yields nothing because the first read() call brings the cursor to the end of the file.

To bring the cursor back to the start use **file1.seek(0).**

**2. Just Reading Mode:**

file1= open('C:\\Users\\Divyanshu Sharma\\Desktop\\file.txt', 'r')

Notice the ‘\\’, double instead of the single slashes. That’s because \U tag in \Users in python means that Unicode is being used, but then it’s followed by ‘s’ which isn’t Unicode. This leads to a “codec can't decode bytes in position 2-3”.

Notice the ‘r’ tag at the end. This means that the file is being opened in read mode. A ‘w’ tag would open the file in write mode.

**Reading Lines:**

**1. Read ONE line at a time: file1.readline()** Reads exactly one line and ‘\n’. eg. 'line1\n'. After reading that line, the ‘reader cursor’ moves to the next line.

**2. Read ALL lines at one time: file1.readlines()** (notice readline() vs readlines() plural!) returns a list whose elements are lines in the file. Character ‘\n’ included.

**3. Read fixed number of characters:** file1.read(7) Reads the first 7 characters. ‘\n’ is read just like any other character. After reading the specified number of characters, the ‘reader cursor’ moves to the next line.

**Closing:** file1.close()

**Example**:

File1 = open('foo.txt', 'r')

for line in File1: #or for line in open('foo.txt', 'r')

print(line)

file\_.close()

**3. Just Writing Mode:**

file1= open(r'C:\Users\Divyanshu Sharma\Desktop\file.txt', 'w')

Notice ‘r’ right before the location of the file. That makes python interpret that string as a raw string and doesn’t do the whole Unicode thing.

‘w’ tag indicates a purely write mode

**Opening:**

**1.** file1= open('C:\\Users\\Divyanshu Sharma\\Desktop\\file.txt', 'w')

‘w’ tag overwrites the existing content

2. file1= open(r'C:\Users\Divyanshu Sharma\Desktop\file.txt', 'a')

‘a’ tag is basically append mode (Add to existing data).

**Writing**: file1.write('new line\n')

**Closing**: file1.close()

**Conditionals**

**If-elif-else:**

if x==1:

print()

elif 10<=x<20:

print()

else:

print()

**Switch**: Python don’t have no Switch

**Loops**

**While:**

i=0

while i<10:

print()

i+=1

print(‘while ended’)

**For:**

**1.** for i in range(5):

print i

**2.** range(start,stop,step)

for i in range(5,10): #5 is the start and 10 is the stop, i goes from 5 to 9

print(i)

**3**. for i in range(5, 11, 2): # for(i=5; i<11; i+=2)

print(i)

**User-defined Functions**

def square(x)

result = x\*x;

return result

print(square(5)) gives 25.

**How to assign a function to a variable**

1. Make a function func()

2. Make a variable=fucn # No ()

3. Even if you delete func, variable will still have the code in it. So, you can call variable() and it’ll do the same thing as what fucn() did.

**Function to check for prime numbers:**

def is\_prime(n):

if n == 2:

return True

if n % 2 == 0 or n <= 1:

return False

for divisor in range(3, int(n\*\*0.5) + 1 , 2):

if n % divisor == 0:

return False

return True

**Lambda Functions (Short, one line functions)**

Syntax: <label (like function name)>= lambda <parameters>: <return value>

Eg: square= lambda num: num\*\*2, takes in num and returns its square

square(10) returns 100.

Eg: CheckEven=lambda num: num%2==0 # returns true if num%2==0 else False

Eg: max\_finder= lambda a,b: a if (a>b) else b # notice that it’s like regular C code, no “ : ”

**Object-Oriented Programming**

**Attributes**: basically the properties of the object student.name, employee.id

**Methods:** Functions in a class. First argument/parameter of each method is ‘self’

**Initializer method (constructor)**: def \_\_init\_\_(self, <attribute>)

**Method to Print an object:** def \_\_str\_\_(self): return self.attribute. Call using print(<object>)

**Method to Print an Object’s length**: def \_\_len\_\_(self): return self.attribute. Call using

len(<object>)

**Method to Delete an Object (Destructor):** def \_\_del\_\_(self): print(‘Object Deleted’). Call using del <Object>

class Dog:

species=’mammal’

def \_\_init\_\_(self, breed, name)

self.breed=breed

self.name=name

Rocky= Dog(“Pug”)

Rocky.breed #gives “Pug”

Rocky.species #gives “mammal”

class Circle:

def \_\_init\_\_(self,radius):

K=Circle(4)

K.area() gives 50.285714285714285

K.circumference() gives 25.142857142857142

self.radius=radius

def area(self):

return 22/7 \* self.radius\*\*2

def circumference(self):

return 2\*22/7\*self.radius

**Inheritance**

class Address: # Base Class

def \_\_init\_\_(self,apt,street,zipcode):

self.apt=apt

self.street=street

self.zipcode=zipcode

def reached(self)

print(‘Control Reached Here’)

def \_\_str\_\_(self):

return str(self.apt)+" "+str(self.street)+" "+str(self.zipcode)

class MyLocation(Address):

def \_\_init\_\_(self,name,apt,street,zipcode):

Address.\_\_init\_\_(self,apt,street,zipcode)

self.name=name

def \_\_str\_\_(self):

return self.name+” “+ Address.\_\_str\_\_(self)

x= MyLocation(‘Div’, 112, ‘Boston’, 452010)

print(X) prints the name+street address

print( X.reached()) prints ‘control reached here’. So basically objects of class MyLocation can access the functions of the base/parents class Address too.

**Exception Handling:**

try:

…insert code that could throw an exception…

except Exception1:

… do this in case of 1st Exception…

Except Exception2:

…do this in case of 2nd Exception condition…

…

…

else:

… if none of the exception conditions match it or when the ‘test’ case runs perfectly do this…

finally: …regardless of there being exceptions or not… this piece of code under ‘finally’ will run

**Tip: If you don’t know the type of exception…**

try: …code…

except: …code…

**To get the full list of exceptions google list of built-in exceptions in Python**

**except TypeError:** is for things being done with wrong types

**Example: (Code to keep on taking input if the user enters something wrong)**

while True

try:

v= int(input(‘Enter a num.: ‘))

except: # didn’t enter an int

print(‘Enter a num. Only:’)

continue

else: # correct input

break

**Modules:**

import <module\_name>, eg. import math, import string

use the built in functions in that module with: module.fucn(), eg. math.sqrt(16)

**Selectively Importing Functions from Modules:**

**Eg:** from math import sqrt

sqrt(16) # returns 4 (no need to have math.sqrt(16))

**Finding New Modules:**

search Python on github.com or just google it and goto the installation section of the page.

**Installing New Modules:**

1. Find where Python is installed on your PC: ( open a new file, try to save it, the default location that opens up is where Python is).

For me: C:\Users\Divyanshu Sharma\AppData\Local\Programs\Python\Python35-32

2. python -m pip install <module name>

**Making Your own module:**

1. Make a file and put the code in it.

2. Save the file in its default location with moduleName.py

3. type, import moduleName

4. call functions inside moduleName with moduleName.fucn()

**Map** (A way to send a list of things into a function and get a list of the result back)

**Syntax:** define a function fucn(), make a list of inputs for the fucn(), get a single list of all the results when inputs have been processed by fucn() using list(map(fucn,input1,input2…))

Eg: **def even(num):**

**if num%2==0:**

**return num\*2**

**else:**

**return 0**

**numbers=[1,2,3,4,5,6]**

**print(list(map(even,numbers)))**

**Trick**: If the function is small, instead of separately defining it, a lambda expression can be used: **list(map(lambda num: num\*\*2,numbers))**

Eg: a=[1,2,3], b=[4,5,6], c=[7,8,9]

# lambda x,y:x+y will return the sum of 2 parameters x and y

list( map ( lambda x,y:x+y, a,b )) # provide a,b as the 2 input lists and return a list where element[i]=a[i]+b[i]

list( map ( lambda x,y,z : x+y\*z, a,b,c )) )) # provide a,b,c as the 3 input lists and return a list where element[i]=a[i]+b[i]\*c[i]

**REDUCE** (takes in a function and a list and returns a single result of doing that function on each element of the list)

from functools import reduce

numbers= [1,2,3,4,5,6] # lambda x,y : x+y takes in 2 nos. and returns their sum

reduce( lambda x,y : x+y , numbers) returns 18.

It will work like this:

add 1 and 2 giving 3, add 3 and 4 giving 7, add 7 and 5 giving 12, add 12 and 6 giving 18. Hence, finally returning 18.

Eg: numbers=[1,234432,-32,2344,120]

reduce(lambda a,b:a if (a>b) else b, numbers) # will return the largest number in the list numbers

**Filter** (Takes in a function that only returns Boolean True or False and a list of inputs. Returns a list of those inputs that gave a True result in the function)

eg: numbers=[1,2,3,4,5,6,7,8,9]

list(filter(lambda x: x%2!=0, numbers)) # returns [1, 3, 5, 7, 9] which is those

entries in numbers that gave a True with the lambda function that checks for

odd numbers.

**all() / any()**

all(<list>) returns True if EVERY entry in the list is True. eg: all([True,True,False]) returns False as not all elements are True

any(<list>) returns True if ANY entry in the list if True. eg: any([True,False,True]) returns True as atleast of the elements is True.

**zip()** (Creates pairs of elements of 2 lists into several tuple)

xvec = [10, 20, 30]

yvec = [7, 5, 3]

print(list(zip(xvec, yvec))) # prints[(10, 7), (20, 5), (30, 3)]

**complex() (generates**

complex(2,3) # returns 2+3j

complex(‘2+5j’) # converts string to complex number, returns 2+5j

**locals() and globals()**

Respectively return the local and global variables and functions in the form of dictionaries with ‘key’=name of variable and the corresponding data= value of the variable

**globals()** returns {'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None, '\_\_package\_\_': None, 'func': <function func at 0x035C9390>, '\_\_builtins\_\_': <module 'builtins' (built-in)>, 'globalVar': 99, '\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}

**locals()** returns the variables that are local to that scope

**globals().keys() or locals().keys()** returns all the keys of that dictionary i.e. all the names of global or local objects

**Function within Function**

**1. This can be used instead of function overloading**

def mainFunc(num=5):

print('Entered mainFunc')

def first():

print("in first fucn")

def second():

print("in second fucn")

if num==5:

first()

else:

second()

print('back in mainFunc')

**2. Returning Function from within a function (no calling, returning)**

def mainFunc(num=5):

print('Entered mainFunc')

def first():

print("in first fucn")

def second():

print("in second fucn")

if num==5:

return first

else:

return second

print('back in mainFunc')

result=mainFunc()# result gets first(), not the function call, the actual function

result1=mainFunc(9)#result1 gets second(), not the function call, the actual function

**3. Function as an argument of a function**

def argumentFunc():

print(‘Function called as argument’)

def callerFunc(func): #takes func as argument

print(‘Caller Function called!’)

return fucn() # because it’s returning fucn() not fucn, fucn has to be a func tion not any other variable

**Decorator (Calling a function from within a function (called the decorator) within a function (called the wrapper) so that you can execute code before and after calling that function. The decorator returns the wrapper)**

**Syntax:**

define some fucn\_needs\_decorator before or after defining decorator

decorator(fucn\_needs\_decorator):

wrapper():

\* insert code that should run before calling fucn\_needs\_decorator \*

fucn\_needs\_decorator() # call fucn\_needs\_decorator

\* insert code that should run after calling fucn\_needs\_decorator \*

return wrapper()

**ways to call the decorator on a function**

1. fucn\_needs\_decorator= decorator(fucn\_needs\_decorator)

Eg:

def decorator\_func(func):

def wrapper\_func():

print('Insert Code before calling func')

func()

print('Insert Code after calling func')

return wrapper\_func

def fucn\_needs\_decorator():

print('This func needs wrapper')

print('\nRunning fucn without a decorator\n')

fucn\_needs\_decorator()

print('\nManually Running fucn WITH a decorator\n')

fucn\_needs\_decorator=decorator\_func(fucn\_needs\_decorator)

fucn\_needs\_decorator()

print('\nAutomatically Running fucn WITH a decorator\n')

2. @decorator # @decorator followed by definition of the function that needs the decorator

def fucn\_needs\_decorator():

…

Eg:

def decorator\_func(func):

def wrapper\_func():

print('Insert Code before calling func')

func()

print('Insert Code after calling func')

return wrapper\_func

print('\nAutomatically Running fucn WITH a decorator\n')

@decorator\_func

def fucn\_needs\_decorator():

print('This func needs wrapper')

fucn\_needs\_decorator()

**Generators** (real time generation of data instead of creating it all at once)

**SAVES PROCESSING POWER. Instead of starting, executing and ending. The generator stays in memory and retains its last state. using next() you can go to the next instance of ‘yield’.**

**eg:**

def myGenerator():

yield 'These'

yield 'a'

yield 'time'

myGenerator\_Instance =myGenerator() # creating an instance of myGenerator()

print(next(myGenerator\_Instance)) # prints ‘These’

print(next(myGenerator\_Instance)) # prints ‘a’

print(next(myGenerator\_Instance)) # prints ‘time’

print(next(myGenerator\_Instance)) #this statement throws a StopIteration error because all yield statements have been exhausted

**If you place a for-loop in the generator, you won’t have to put multiple yield statements**

**eg:**

def myGenerator(number):

for i in range(number): # runs the yield 11 times and sends squares of 0-10

yield i\*\*2

for num in myGenerator(11): # sends 11 to generator

print(num)

**Example of generator retaining the last value of the variables**

def myGenerator(number):

a=5

d=2

temp=a

for i in range(number):

yield temp

temp+=d

for num in myGenerator(11):

print(num)

**Collections Module**

**1. Counter: (Counts the number of times each data value appears)**

from collections import Counter

l=[1,1,1,2,2,8,8,0]

Counter(l) # gives Counter({1: 3, 8: 2, 2: 2, 0: 1})

string='bird bird bird is the word word word the'

Counter(string) # gives Counter({' ': 8, 'r': 6, 'd': 6, 'i': 4, 'o': 3, 'b': 3, 'w': 3, 'e': 2, 't': 2, 'h': 2, 's': 1})

Counter(string.split()) # gives Counter({'word': 3, 'bird': 3, 'the': 2, 'is': 1})

**Count ‘n’ number of most common words:**

string='bird bird bird is the word word word the'

c= Counter(string.split())

**c.most\_common(2)** # gives 2 most common values in form of a list of tuples: [('word', 3), ('bird', 3)]

**Count ‘n’ number of least common words:**

string='bird bird bird is the word word word the'

c= Counter(string.split())

**c.most\_common()[:-n-1:-1]** # gives n lest common values in form of a list of tuples

**2. Ordered Dictionary ( Normal Dictionary that retains the order in which data was stored into it)**

import collections

f= collections.OrderedDict()

f= {1:2,2:3,3:4,4:5} # f now retains its order of elements

**Equality of Ordered Dictionaries**

Normal Dictionaries check if same data is present in the 2 dictionaries being compared. Normal Dictionaries don’t care about what order that same data is present in. Ordered Dictionaries need both same data and same position of the data.

eg: Normal Dictionaries d1={1:’a’, 2:’b’} and d2={2:’b’, 1:’a’} are both same.

if d1 and d2 were Ordered Dictionaries, they won’t be equal as the order of elements is different.

**3. namedtuple (Python’s version of Structures or really short declaration of new Classes/Objects)**

from collections import namedtuple

Student= namedtuple('Student','name age grade')

pronoy= Student(name=’PP’,age=19,grade=14)

rahul= Student(name='Rahul',age=15,grade=10)

print(rahul) # prints Student(name='Rahul', age=15, grade=10)

rahul.name # gives 'Rahul'

rahul.age # gives 15

rahul.grade # gives 10

rahul[0] # gives first data associated with rahul=’Rahul’

rahul[1] # gives second data associated with rahul=15

rahul[2] # similarly, gives 10

**datetime** ( import datetime)

**Current date and time:** datetime.datetime.now() gives datetime.datetime(2017, 6, 20, 2, 44, 9, 234773)

now= datetime.datetime.now()

now.time()

now.date()

now.month

now.date

Time:

t= datetime.time(<hour>,<minute>,<second>,<microsecond>) #t=datetime.time(5,6,17,2322)

t.hour gives 5, t.minute gives 6, t.second gives 17, t.microsecond gives 2322

**Current Time:** datetime.datetime.now().time() gives datetime.time(2, 44, 53, 725895)

Date:

Today’s date: tod=datetime.date.today(), print(tod) prints the today’s date in yyyy-mm-dd formal

d=datetime.date(<year>,<month>,<day>) # d= datetime.date(2017,6,5)

Edit date: d=d.replace(year=2000), d=d.replace(month=7), d= d.replace(day=9)

Differences in two dates: diff= d1-d2 gives diff as the object of timedelta and returns the number of days that have passed

**pdb- Python Debugger** (import pdb)

type pdb.set\_trace() right before the line where your code is breaking. It opens an interactive debugging environment right there and lets you check the values of variables and do different operations on them.

**timeit** ( gives you the time it takes to run a statement particular number of times)

import timeit

timeit.timeit(' "-".join(str(n) for n in range(100))',number=10000)

**# first argument is the statement between ‘ ‘ (single quotes) as a string**

**# second argument is ‘number=’ i.e. the number of times that line should run**

**Regular Expressions** ( find substrings in larger strings and get associated data about where the substring was found)

import re

term='hello'

bigStr='hi hello! I am Andy!'

m= re.search(term,bigStr) # returns <\_sre.SRE\_Match object; span=(3, 8), match='hello'>

#returns None if object is absent.

m.start(), m.end() etc methods can be used

-----------------------------------------------------------------------

finf= re.findall(term,bigStr) #finf is the list of all ‘hello’ found in bigStr

# finf= ['hello', 'hello']

**Syntax to search patterns**

string= ’sdsd..sssddd…!sdddsddd..#..?dsds..$$$...dsssss…sdddd’

text\_patterns=[ ‘sd\*’, # ‘s’ followed by 0 or more ‘d’

‘sd+’, # ‘s’ followed by 1 or more ‘d’

‘sd?’, # ‘s’ followed by 0 or 1 ‘d’

‘sd{3}’ # ‘s’ followed by 3 ‘d’s

‘sd{2,3}’ # s followed by 2 to 3 ‘d’s

# [] creates an OR situation

‘[sd]’ #either ‘s’ or ‘d’

‘s[sd]+’ # ‘s’ followed by 1 or more either ‘s’ or ‘d’

# [^] creates an exclusion

‘[^!#$.]+’ # give me everything except ‘!’,’#’,’$’,’.’

# [-] creates a range of characters to look for

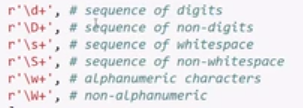
‘[a-z]+’ # 1 or more alphabet between a-z

‘[A-Z]+’ # 1 or more alphabet between A-Z

‘[A-Za-z]+’ # 1 or more alphabet between A-Z or 1 or more alphabet between a-z

‘[A-Z][a-z]+’ # 1 or more instance of an alphabet from A-Z followed by alphabet from a-z

# r’\’ helps look for escape codes such as \n,\d etc.



]

now you can search each element of text\_patterns in string to get results as described in comments in text\_patterns.

**Tips:**

1. a=[1,2,3]

b=a is you literally making b=a, so any future change in a will cause the exact same change in b. a and b are the SAME thing.

b=a[:] is you copying contents of a into an independent entity b.

2. != and <> are the same.

3. i=5; print(i)

is the same as

i=5

print(i)

4. type(a) gives the data type of the variable a

pyqt