# Python 101

Objects are like containers of data.

In Python everything is an Object

Numbers, Strings and Booleans = single value

Tuples, Lists, Sets and Dictionaries – Multiple Values

Every object will have a Name, Type and an ID.   
Type and ID will not change during the life of the object

Rules for naming objects:  
1. Names must only have Alphanumeric and Underscore characters (A-Z, a-z, 0-9, \_)

2. Names must not start with a digit

type(), id() = memory location

Numbers & Booleans:  
Integers

Floating Point Numbers

Complex Numbers

Booleans – true & false

Functions:  
min, max, abs

Math modules (must be imported first)

True and False are Keywords in Python

Has to be Capital T and F

### Strings:

Strings: Sequence of Characters

Raw Strings -r

Pre-Formatted Strings – Triple Quotes

Concatenation Operator +

Repetition Operator \*

Raw Strings use -r

print(“c:\some\name”) \n will be interpreted as a new string

but

print(r”c:\some\name”) will keep it all on the same line and print as written

Triple Quotes: preformatted string print as it written

“”” Python

is

a nice language

to work”””

Concatenation:

str1= "Hi my name is: "

str2 = "Angel"

message = str1 + str2

Hi my name is: Angel

Repetition Operator

“A” \*10

AAAAAAAAAA

### Container Objects

Tuples ()

Lists []

Sets {} unordered without any duplicates

Dictionaries {} unordered key value pairs

File Objects

Strings

### Mutability of Objects

Mutable – Lists, Dictionaries

Immutable – Tuples, Sets, Numbers, Strings

mutable = change items, add items

### Arithmetic Operators

Addition +

Subtraction -

Multiplication \*

Division /

Power of \*\*

Floor Division // will divide to root no decimal

Modulus % (what is left after division) 8%3 = 2

### Bitwise Operators:

Left Shift <<

Right Shift >>

Bitwise AND &

Bitwise OR |

Bitwise Exclusive OR ^

Bitwise NOT ~

### Comparison Operators:

Less Than <

Greater Than >

Less Than or Equal <=

Greater Than or Equal >=

Equal ==

Not Equal !=

Equal Value & ID is

Exists in

### Assignment Operators

+=. -=, \*=, /=

\*\*=, //=, %=

<<=, >>=

&=, |=, ^=

Logical Operators:

* a and b is True only if both a and b are True.
* a or b is True if either a or b is True.
* not a is True only if a is False.

print( 2 < 3 and 3 > 1) True

print( 2 < 3 or 3 > 1) True

print(2 < 3 or not 3 > 1) True

print( 2 < 3 and not 3 > 1) False

### Conditionals

if

if else

if elif else

Blocks of Code – By Indentation not with {}

### Loops:

while condition:  
 block of statements

for x in container:

block of statements

### Break and Continue:

continue: goes back to the beginning of the loop

break: goes out of the loop

else: when loop condition is false,

else part is executed (else is rarely use with loops)

s = "Hello World"

x = 0

while (x <len(s)):

if (s[x] == ' '):

continue

x += 1

print(s[x], end='')

x += 1

while (x <len(s)):

if (s[x] == ' '):

break

x += 1

print(s[x], end='')

x += 1

### Range Function

range(stop)

range(start, stop)

range(start. stop, step)

create a range of numbers.

### String Object Basics:

Strings – sequence of characters

act like arrays

Immutable Objects

Sliceable

### String Methods capitalize()

lower()

upper()

swapcase()

Ljust(width[, fillchar])

rjust(width[, fillchar])

center(width[, fillchar])

lstrip(chars)

rstrip(chars)

strip(chars)

zfill(width)

### More String Methods: find(sub[, start[, end]])

index(sub[, start[, end]])

replace(old[, new[, count]])

isalnum()

isalpha()

isdecimal()

isdigit()

isidentifier()

islower()

isupper()

isnumeric()

isprintable()

isspace()

### Split and Join Methods:

string.split(sep, max-splits)

string.join(container Object)

### String Format Method:

Arguments by position: message: “This is a {0} with {1}”.format(10, 20)

Unpacking Arguments: “{2}, {1}, {0}”.format(\*’abc’)

Arguments by Name: ‘x = {x}, y = {y}’.format(x=10.5, y=20.5)  
  
Arguments by List Items: t =(10, 20)

print(‘{}…{}’.format(t))

“this is a %s” % “string”

### List Objects Basics:

Comma separated items enclosed in square brackets []

Zero base indexed list

Mutable Objects

sliceable

nest-able (list in a list)

sequencing and unpacking

x = [1, 2, 3, 4 ]

a,b,c,d = x

a = 1, b = 2, 3 = c, d = 4  
  
Range Function:

x = range(2, 30, 3)  
x1 = list(x)

result:  
[2, 5, 8, 11, 14, 17, 20, 23, 26, 29]

### List Methods:

append(x) : appends an item(x) to the end of a list

insert(i, x) : inserts an item at index of x

remove(x) : removes x from the list first match x

pop([i]) : removes 1 item at index of I if I not specified it is remove from end of list

clear() : all items removed from x

index(x) : will return the index location of x

count(x) : number of occurences of x

sort(x) : sort will sort

reverse(x) : will reverse the items

del : remove items at the given location like pop, but delete will also delete a range of items using slice

Stacks and Queues:  
  
Data structures with lists:

Stack: Last In – First Out (LIFO)

Queue: First In – First Out (FIFO)  
  
How to create a STACK:  
empty stack

S = []  
S.append(10)  
[10]  
S.append(20)

[10, 20]  
10 is bottom of the Stack, and 20 is top of the stack  
If you add 3 more:  
S.append(30)

S.append(40)

S.append(50)  
[10, 20, 30, 40, 50]  
bottom of stack is 10, and 50 is top of stack  
  
Q = []

Q.append(10)

Q.append(20)

Q.append(30)

Q.append(40)

Q.append(50)

Q

Result:  
[10, 20, 30, 40, 50]  
to run the list as a Queue you need to specify the queue order in pop  
Q.pop(0)  
Result  
[20, 30, 40, 50] FIFO

### List Comprehensions:

Python supports a concept called “list comprehensions”

It can be used to construct lists in a very natural, easy way.

sq = []

for x in range(10):

sq.append(x\*\*2) (powerOf \*\*)

sq

creates a list of x squared:  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Same process using list comprehension:  
sq2 = [x \*\* 2 for x in range(10)]

sqEven = [x for x in sq2 if x % 2 == 0]

sqEven

Results: [0, 4, 16, 36, 64]

sqOdd = [x for x in sq2 if x % 2 != 0]

[1, 9, 25, 49, 81]

list of power of 2 :  
p = []

for x in range(15):

p.append( 2 \*\* x)

results:  
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384]

same with list comprehension:

p2 = [2 \*\* x for x in range(15)]

results:  
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384]

a = [1, 2, 3]

b = [4, 5, 1]

c = [(1,4), (1,5), (2, 4), (2, 5), (2, 1), (3,4), (3, 5), (3, 1)] what we want is a c to be a list of tuples from b and c

for x in a:

for y in b:

c.append((x,y))  
results:  
[(1, 4), (1, 5), (1, 1), (2, 4), (2, 5), (2, 1), (3, 4), (3, 5), (3, 1)]

we don’t want (1,1)  
  
fixed:  
a = [1, 2, 3]

b = [4, 5, 1]

c = []

for x in a:

for y in b:

if x != y:

c.append((x,y))

results:

[(1, 4), (1, 5), (2, 4), (2, 5), (2, 1), (3, 4), (3, 5), (3, 1)]

c2 = [(x,y) for x in a for y in b if x != y]

results:  
[(1, 4), (1, 5), (2, 4), (2, 5), (2, 1), (3, 4), (3, 5), (3, 1)]

List comprehensions is harder to read, the normal for clause is easier to see and understand.

Tuples:  
Comma separated items enclosed in parenthesis()

Zero based indexed lists

Immutable Objects

Sliceable

Nest-able

practice:  
 t = (1,2,3)

print(type(t))

print(t[0])

t = t + t + t

print(t)

print(t.index(1))

t1 = (range(1,10,2))

t2 = tuple(t1)

print(t2)

nest = (1, 2, 3)

nest2 = (4,5,6)

nest3 = (nest,nest2)

print(nest3)

t1 = (1,2,3)

list = [10,20,30]

T = (1, 2, 3, 10.5, 'john', (1,2,3), [10,20,30])

print(len(T))

T[6][1] = 200

print(T)

t1 = (1,2,3)

t2 = (4,5,6)

print(id(t1))

t1 = t1 + t2

print(t1)

print(id(t1))

### Sets:

Unordered collection with no duplicates enclosed in curly brackets{}

Not Indexed

Support Mathematical Set Operations:

Union, Intersection, Difference and Symmetric Difference

set() function

practice code:  
 s = {1,2,3,4}

print(type(s), len(s))

print(s[3])  
 print(s[3])

TypeError: 'set' object does not support indexing

s = set('aaabbbcccddeeeeeffggg')

print(s)  
 {'d', 'a', 'f', 'c', 'b', 'e', 'g'}  
 set is no duplicates

Create set from list:

t = [1,2,2,3,4,5,5]

s2 = set(t)

print(s2)

{1, 2, 3, 4, 5}

in operators:  
 t = [1,2,2,3,4,5,5]

s2 = set(t)

print(s2)

print(1 in s2) true

print( 6 in s2) false

Mathematical operations

A = {1,2,3,4,5}

B = {8,4,5,6,7}

print(A, B)

{1, 2, 3, 4, 5} {4, 5, 6, 7, 8}

print(A - B) All the elements that are in A and not in B

{1, 2, 3}

print(A|B) all the unique elements from A and B

{1, 2, 3, 4, 5, 6, 7, 8}

print(A&B) all elements in both A and B

{4, 5}

print(A^B) Exclusior operation it gives list of elements in each set which are not in the other set

{1, 2, 3, 6, 7, 8}

Sets support sequence unpacking:  
 x = {10,20,30}

a, b, c = x

print(a, b, c) 10 20 30

How to create an empty set:  
 s = {} NOT SUPPORTED Can’t create an empty set this way.

because dictionary uses curly brackets.

s = set() is how you create an empty set.

### Dictionary Basics:

Unordered collection of key:value pairs

enclosed in curly brackets{}

Keys must be unique with in a dictionary

Mutable objects

Operations: Add, Read, Delete key:value pair

practice code:

scores = {"John":98, "Alex":85, "Tom":88, "Cathy":90}

print(scores)

{'John': 98, 'Alex': 85, 'Tom': 88, 'Cathy': 90}

scores["Kris"]=95

print(scores)

{'John': 98, 'Alex': 85, 'Tom': 88, 'Cathy': 90, 'Kris': 95}

scores = {"John":98, "Alex":85, "Tom":88, "Cathy":90}

print(scores)

{'John': 98, 'Alex': 85, 'Tom': 88, 'Cathy': 90}

scores["Kris"]=95

print(scores)

{'John': 98, 'Alex': 85, 'Tom': 88, 'Cathy': 90, 'Kris': 95}

scores["Cathy"]=99

print(scores)

{'John': 98, 'Alex': 85, 'Tom': 88, 'Cathy': 99, 'Kris': 95}

result = ""

input = input("What score do you want to check?")

for name, score in scores.items():

if score == int(input):

result=name

print(result)

What score do you want to check?88

Tom

print(list(scores))

['John', 'Alex', 'Tom', 'Cathy', 'Kris']

print(scores.keys())

dict\_keys(['John', 'Alex', 'Tom', 'Cathy', 'Kris'])

print(sorted(list(scores)))

['Alex', 'Cathy', 'John', 'Kris', 'Tom']

grades = dict(A=90, B=80, C=70, D=60, F=50)

{'A': 90, 'B': 80, 'C': 70, 'D': 60, 'F': 50}

marks = dict([('Cathy', 90), ('Tom',99)])

{'Cathy': 90, 'Tom': 99}

### List Comprehensions with Dictionaries:

sq = {x:x\*\*2 for x in (1,2,3,4,5)}

print(sq)

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

sq2 = {x:x\*2 for x in range(11)}

print(sq2)

{0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18, 10: 20}

sq2.keys()

dict\_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

sq2.values()

dict\_values([0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20])

sq2.items()

dict\_items([(0, 0), (1, 2), (2, 4), (3, 6), (4, 8), (5, 10), (6, 12), (7, 14), (8, 16), (9, 18), (10, 20)])

### Dictionary Object Methods:

len(d)

d[key] : give the value associated with the key

d[key]=value : updates the value of a pair

del d[key] : delete key value pair

key in d : find if a key is in the dictionary

key not in d : find if a key is not in the dictionary

iter(d) : by passing a dictionary it becomes a iterable item. loop through all the items

d.clear() : clear out all items in dictionary

d.pop(key) : delete a key:value pair

d.popitem() – Random del a key value pair

d.items() : return items in dictionaries

d.keys() : return just the keys

d.values() : return just the values

scores["Cathy"]=99

{'John': 98, 'Alex': 85, 'Tom': 88, 'Cathy': 99, 'Kris': 95}

print(scores["Cathy"])

result: 99

scores[“Alex”]=92

{'John': 98, 'Alex': 92, 'Tom': 88, 'Cathy': 99, 'Kris': 95}

del scores[“John”]

{'Alex': 92, 'Tom': 88, 'Cathy': 99, 'Kris': 95}

print("Cathy" in scores)

True

print(“Gary” in scores)

False

{'Cathy': 90, 'Kris': 95, 'Tom': 88, 'Alex': 90}

s = iter(marks)

print(type(s))

<class 'dict\_keyiterator'>

sq = {x:x\*2 for x in range(1, 11)}

print(sq)

{1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18, 10: 20}

sq.pop(10)

print(sq)

{1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18}

### Dictionary View Objects:

d.items() All pairs

d.keys() All keys

d.values() All values

they each view the dictionary

sv=scores.values()

print(sv)

dict\_values([92, 88, 99, 95])

scores["Steve"]=100

print(sv)

dict\_values([92, 88, 99, 95, 100]) (updates as the dictionary updates why it is called view object.

sum = 0

for s in sv:

sum += s

print(sum)

474

print(92+88+99+95+100)

474

for k,v in scores.items():

print(k, v)

Alex 92

Tom 88

Cathy 99

Kris 95

Steve 100

for v in scores.values():

print(v, end=’’)

92 88 99 95 100

for k,v in scores.items():

print(k, v, end=’’)

Alex 92 Tom 88 Cathy 99 Kris 95 Steve 100

for k in scores.keys():

print(k, end=’’)

Alex Tom Cathy Kris Steve

### Python Functions:

Basic Syntax

Return Statement

Passing parameters

Default Values

Doc String

function is a piece of code with a name that we can call and use numerous times or once.

def getStudentByScore(num):

result = ""

input = input("What score do you want to check?")

for name, score in scores.items():

if score == int(input):

result=name

print(result)

def greet():

name = input("What is your name?")

message = ("Hello %s!" % (name))

print(message)

What is your name?Angel

Hello Angel!

def returnString():

message = "this is a string"

return message

print(str1)

this is a string

parameter:

passing a parameter:

def getSum(a, b):

result = a + b

print(result)

Hobbies.getSum(2,6)

8

def greetN(num):

i = 0

while(i<num):

print("Hello World")

i += 1

Hello World

Hello World

Hello World

DOC string:  
First comment in function used for description

### Parameters Passing in Functions:

Three ways to pass parameters:  
Positional Parameters

Variable Number of Parameters

Named Parameters

#positional parameters

def sumNums(n1, n2, **n3= 30**):

result = n1 + n2 + n3

print("%s + %s + %s = %s." % (n1, n2, n3, result))

return(result)

**n3= 30** = default variable

#variable number of parameters

def sumVariableNums(\*args):

sum = 0

for n in args:

sum += n

print("Total = " + str(sum))

def sumVariNums(\*args):

sumList = []

for n in args:

sum = 0

for i in n:

sum = sum + i

sumList.append(sum)

return sumList

Named Parameters:

Passing Mixed Parameters

#named parameters:  
 def fruitBasket(\*\*kwargs):

count = 0

for fruit, cnt in kwargs.items():

printString = "{:20}{:10}".format(fruit, cnt)

print(printString)

count += cnt

print("{:20}{:10}".format("Total Fruit: ", count))

Hobbies.fruitBasket(apples=100, banana=144, pears=77, grapes=200, mangoes=35)

passing mixed:

def myFunc (n1, n2, n3, \*args, \*\*kwargs ):

sum1 = sum2 = sum3 = 0

sum1= n1 + n2 + n3

for n in args:

sum = sum2 +n

for K, v in kwargs.items():

sum3 = sum3 + v

sums = [sum1, sum2, sum3]

return sums

sums = Hobbies.myFunc(10,20,30,11,22,33,44, one=100, two=200, three=300)

print(sums)

def getStudentByScore(num):

result = ""

input = input("What score do you want to check?")

for name, score in scores.items():

if score == int(input):

result=name

print(result)

def greet():

name = input("What is your name?")

message = ("Hello %s!" % (name))

print(message)

def returnString():

message = "this is a string"

return message

def getSum(a, b):

result = a + b

print(result)

def greetN(num):

i = 0

while(i<num):

print("Hello World")

i += 1

#positional parameters

def sumNums(n1, n2, n3):

result = n1 + n2 + n3

print("%s + %s + %s = %s." % (n1, n2, n3, result))

return(result)

#variable number of parameters

def sumVariableNums(\*args):

sum = 0

for n in args:

sum += n

print("Total = " + str(sum))

def sumVariNums(\*args):

sumList = []

for n in args:

sum = 0

for i in n:

sum = sum + i

sumList.append(sum)

return sumList

def fruitBasket(\*\*kwargs):

count = 0

for fruit, cnt in kwargs.items():

printString = "{:20}{:10}".format(fruit, cnt)

print(printString)

count += cnt

print("{:20}{:10}".format("Total Fruit: ", count))

def myFunc (n1, n2, n3, \*args, \*\*kwargs ):

sum1 = sum2 = sum3 = 0

sum1= n1 + n2 + n3

for n in args:

sum2 = sum2 +n

for K, v in kwargs.items():

sum3 = sum3 + v

sums = [sum1, sum2, sum3]

return sums

### Iterators

Iterable Objects: all containers (strings, Tuples, Lists, Dictionaries):

"Iterators are required to have an \_\_iter\_\_() method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted."  
  
How a for loop works?

for x in list

I = iter(x)

type becomes a list\_iterator

in 2.7

next(I)

in 3.0

I.\_next\_()

#what happens in the for loop:

I = iter(x) #this makes the string or list iterable

print(type(I))

#calls this function

next(I)

I= iter(x)

print(next(I))

print(next(I))

print(next(I))

print(next(I))

print(next(I))

print(next(I))

Result:

1

2

3

4

5

Traceback (most recent call last):

File "test.py", line 38, in <module>

print(next(I))

StopIteration

### Generator Functions:

A generator function is function that return an iter-able object.

(range object)

How to write a Generator Function

x = range(10)

for i in x: print(i, end=' ')

result:  
0 1 2 3 4 5 6 7 8 9 >Exit code: 0

#y = range(start, stop, step)

y = range(10, 100, 5)

for x in y: print(x, end=' ')

result:

10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 >Exit code: 0

print(type(y))

result: <class 'range'>

Writing a Generator Function:  
 def myRange(start, stop, step):

i = start

while(i < stop):

yield i

i = i + step  
The **python** **yield** **statement** turns a function into a generator, i.e. a function which can return an intermediate value to the caller (the next value), but maintain the function's local state so that the function can resume again at the place it left off

#This is the main program

if \_\_name\_\_ == '\_\_main\_\_':

R = Hobbies.myRange(10, 30, 5)

for i in R:

print(i, end=' ')

result: 10 15 20 25 >Exit code: 0

Write a function count\_digits to find number of digits in the given number:

def countDigits(number):

count = 0

for i in str(number):

count += 1

print("Number of digits in %s = %s" % (number, count))

Write a function istrcmp to compare two strings, ignoring the case.

def istrcmp(str1, str2):

if str1.upper() == str2.upper():

print("%s == %s: True" % (str1, str2))

else:

print("%s == %s: False" % (str1, str2))

### Lambda Functions:

Lambda operator or lambda function is a way to create small anonymous functions, ie functions without name.

These functions are throw away functions, ie, they are jut needed where they have been created. Lambda functions are mainly used in combination with the functions (filter(), map(), and reduce().

These functions are very simple, just one line of code, no loops and no code blocking. But we can use simple if-else conditions

The general syntax of a Lambda function is quite simple:

lambda argument\_list:expression

just an expression

Function for power of 2:  
def power2(x):

result = x \*\* 2

print( result)

Lambda Function:  
lambda x : x \*\* 2

<function <lambda> at 0x0000022AE0751378>

normal use:

sq = lambda x : x \*\* 2  
print(sq(5)) =25

print(type(sq))

result: <class 'function'>

sum = lambda x, y, z : x + y + z

print(sum(1,2,3))

result: 6

lambda input parameters : output expression

isEven = lambda x : x % 2 == 0

print(isEven(5)) False

print(isEven(6)) True

func1 = lambda x : x +2 if x%2 == 0 else x+1

print(func1(3)) result 4

print(func1(4)) result 6

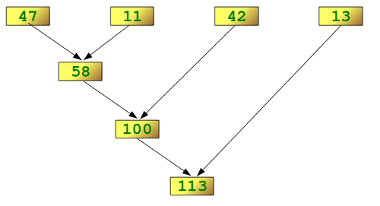
### map(), reduce() and filter() functions:

1. map(fn, x)
2. reduce(fn, x)
3. filter(fn, x)

fn: Any function created with a def keyword or a Lambda function. Normally we use Lambda functions with these 3 functions.

x: Any container object like a String, List or Tuple

1. map(fn, x) executes ‘fn’ function on each item of the container object and produces a new container object.
2. The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.   
     
   If seq = [ s1, s2, s3, ... , sn ], calling reduce(func, seq) works like this:

import functools for Python 3.x not in 2.7  
  
reduce(lambda x,y:x+y, [47, 11, 42, 113]) = 113  


1. The function filter(function, list) offers an elegant way to filter out all the elements of a list, for which the function *function* returns True.   
   The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list *l*. Only if f returns True will the element of the list be included in the result list.

**Map() exampleS:**  
 c = [1,2,3,4,5,6,7,8,9]

fn = lambda x: x\*2

print(list(map(fn, c)))

C = "Python Programming Is Fun".split()

fn1 = lambda s:len(s)

print(list(map(fn1, C)))

x = [1,2,3]

y = [10,20,30]

fn2 = lambda a, b:a\*b

print(list(map(fn2, x,y)))

**Reduce() examples:**

Get the max number from a list of numbers:

C = [10,11,22,33,5,77,22]

fn = lambda x,y: x if x>y else y

print(functools.reduce(fn,C))

result: 77

Add numbers in List:  
 fn1 = lambda x, y: x +y

print(functools.reduce(fn1, C))

result: 180

**Filter() examples:**

c = [i for i in range(20)]

fn = lambda x: x>10

print(list(filter(fn, c)))

result: [11, 12, 13, 14, 15, 16, 17, 18, 19]

fn1 = lambda x: x%2

print(list(filter(fn1, c)))

result: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

Challenge 1:  
Write a function using variable number of parameters concept to count the number of occurrences of each character in a given bunch of strings.

def countChars(\*args):

letters = {}

for i in args:

for c in i:

try:

letters[c.upper()] += 1

except:

letters[c.upper()] = 1

print(letters)

return(letters)

countChars("Gary", "Kris", "Cathy", "John", "Alex")

result: {'G': 1, 'A': 3, 'R': 2, 'Y': 2, 'K': 1, 'I': 1, 'S': 1, 'C': 1, 'T': 1, 'H': 2, 'J': 1, 'O': 1, 'N': 1, 'L': 1, 'E': 1, 'X': 1}

Challenge 2:  
Write myRange function that takes one, two, or three parameters and produces numerical sequences just like Python’s range function

def myRange(start, stop, step=1):

i = start

while(i<stop):

yield i

i += step

R = myRange(5,30)

print(list(R))

result: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]

def myRangeBig(\*args):

start = stop = step = 0

if (len(args) ==3):

start = args[0]

stop = args[1]

step = args[2]

elif(len(args) == 2):

start = args[0]

stop = args[1]

step = 1

else:

start = 0

stop = args[0]

step = 1

i = start

while(i<stop):

yield i

i += step

def myFunc(x,y,z, \*args, \*\*kwargs):

sum1 = sum2 = sum3 = 0

sum1 = x + y +z

for arg in args: sum2 += arg

for v in kwargs.values():sum3 +=v

sums = [sum1, sum2, sum3]

return sums

total = myFunc(10,20,30,1,2,3,4,5,6,7, one=100, two=200, three=300)

print(total)

result: [60, 28, 600]

## OOP – Classes and Objects (Object Oriented Programming)

### OOP Basic Concepts:

Class – template for creating objects

Object – An instance of a Class

Class Consists of Data & Action Methods

Data – Class Variables & Object Variables

Class Variables – Shared by all objects of the class

Object Variables – Instance Variables are within the object space and accessible only to that object.

Encapsulation – Packing of Data variables and action methods is called encapsulation

Inheritance – classes can be extended (both attributes and functionality) by creating derived classes using all the properties of base (super) class.

Multiple Inheritance – Creating classes using more than one base class. Python supports multiple inheritance.

Methods Overriding – A base class method can be changed or extended in a derived class.

### Creating Classes and Objects:

start with keyword class, followed by the name of the class:

body of the class start

1 Constructor or initializer

def \_init\_(self, name, age):

self.name = name

self.age = age

Examples:

class Person:

count = 0

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

Person.count += 1

def changeAge(self, age):

self.age = age

def getCount(self):

result = Person.count

return result

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

return self.name + '--' + str(self.age)

#This is the main program

if \_\_name\_\_ == '\_\_main\_\_':

p1 = Person("John", 20)

print(p1)

print(p1.getCount())

p2 = Person("Gary", 18)

print(p2)

print(p2.getCount())

p1.changeAge(34)

print(p1)

print(p1.getCount())

### Inheritance:

Child Inherit properties from parent

Derived class inherits properties from its base class

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def changeAge(self, age):

self.age = age

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

result = self.name + '--' + str(self.age)

return result

class Student(Person):

def \_\_init\_\_(self, name, age, grade, gpa):

Person.\_\_init\_\_(self, name, age)

self.grade = grade

self.gpa = gpa

def changeGrade(self, grade):

self.grade = grade

def changeGpa(self, gpa):

self.gpa = gpa

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

result = Person.\_\_str\_\_(self) + '--' + str(self.grade) + '--' + str(self.gpa)

return result

p1 = Person("John", 20)

p2 = Person("Gary", 18)

p1.changeAge(34)

print(p1)

s1 = Student("James", 18, 12, 3.2)

print(s1)

s2 = Student("Peter", 22, 12, 4.0)

print(s2)

s2.changeGpa(3.8)

print(s2)

s2.changeAge(21)

print(s2)

John--34

James--18--12--3.2

Peter--22--12--4.0

Peter--22--12--3.8

Peter--21--12--3.8

### Multiple Inheritance:

#!/usr/bin/python

class Person:

def \_\_init\_\_(self, firstName, lastName, age):

self.firstName = firstName

self.lastName = lastName

self.name = firstName + " " + lastName

self.age = age

def changeAge(self, age):

self.age = age

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

result = self.name + '--' + str(self.age)

return result

#------------------------------------------------------------

class Address:

def \_\_init\_\_(self, street, city, state):

self.street = street

self.city = city

self.state = state

def changeCity(self, city):

self.city = city

def changeStreet(self, street):

self.street = street

def changeState(self, state):

self.state = state

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

result = str(self.street) + '--' + self.city+ '--' + self.state

return result

#------------------------------------------------------------

class Student(Person, Address):

def \_\_init\_\_(self, firstName, lastName, age, grade, gpa, street, city, state):

Person.\_\_init\_\_(self, firstName, lastName, age)

Address.\_\_init\_\_(self, street, city, state)

self.grade = grade

self.gpa = gpa

def changeGrade(self, grade):

self.grade = grade

def changeGpa(self, gpa):

self.gpa = gpa

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

result = Person.\_\_str\_\_(self) + '--' + str(self.grade) + '--' + str(self.gpa) + '--' + Address.\_\_str\_\_(self)

return result

#This is the main program

if \_\_name\_\_ == '\_\_main\_\_':

p1 = Person("John", "Smith", 20)

p1.changeAge(34)

print(p1)

s1 = Student("James", "Jones", 18, 12, 3.2, "East La Salle", "Brigham City", "Utah")

s2 = Student("Peter", "Pumpkineater", 22, 12, 4.0, "49th Street", "New York City", "New York")

print(s2)

s2.changeGpa(3.8)

print(s2)

s2.changeAge(21)

print(s2)

s1.changeCity("Colorado Springs")

s1.changeState("Colorado")

print(s1)

## Working with Files:

File Types: text & binary

Opening Files

Open Modes

Reading & Writing text files

Reading & Writing binary files.

Text Files – textual data (html, xml script files created with Text Editors)

Binary Files – Textual Data and Custom Binary data. (Image, Audio and Video files)

### File Open & Modes:

fp = open(‘filename’, mode)

Modes:  
Read Mode: r / r brb = read binary

Write Mode: w / wb

Append mode: a / ab

Read-Write Mode: r+

File Methods:

readline() – Reads one line from the file object

read() – Buffered read

write() – Buffered write

tell() – to get Current file pointer position

seek() to move the file pointer

close() – to close the file.

fp = open("c:\\test.txt", 'a')

fp.write("\n This is not a file")

fp.close()

fr = open("c:\\test.txt", 'r')

for line in fr:

print(line)

fr.close()

result:  
This is a file

This is not a file

f = open("c:\\test.txt", 'r')

str1 = f.readline()

print(str1)

result:  
This is a file

f = open("c:\\test.txt", 'r')

n = 0

for line in f:

n+=1

print(n, line, end=" ")

f.close()

print()

print("Read {} lines.".format(n))

result:  
1 This is a file

2 This is not a file

3 This is not a file

Read 3 lines.

Linux command: ! ls -l ‘foldername’

Read from and write to another:

fileR = open("c:\\test.txt",'r')

fileW = open('c:\\write.txt', 'w')

for line in fileR:

fileW.write(line)

print("Done")

this will create a new file if it doesn’t exist

Buffered Read and Write:

BUFFSIZE = 25000

iFile = 'c:\\BASECore64o.txt'

oFile = 'c:\\NewFile.txt'

fileI = open(iFile, 'r')

fileO = open(oFile, 'w')

buffer = fileI.read(BUFFSIZE)

while (len(buffer)):

fileO.write(buffer)

print("{} bytes written to {}".format(len(buffer), oFile))

open Image:

def openImage():

image = open('c:\\BlueFullMoon.jpg', 'rb')

print(image)

image.close()

### pointer positions:

f = open('C:\\test.txt', 'r')

fpos = f.tell()

print("fpos =", fpos)

line = f.readline()

print(line, end="")

fpos = f.tell()

print("fpos =", fpos)

results:

fpos = 0

This is a file

fpos = 16

Seek(), tell():  
 f = open('C:\\test.txt', 'r')

fpos = f.tell()

print("fpos =", fpos)

data = f.read()

print(len(data))

print(data)

fpos = f.tell()

print("fpos =", fpos)

print('-' \*50)

f.seek(100)

fpos = f.tell()

print("fpos =", fpos)

print('-' \*50)

f.seek(10)

print(data)

### Using Standard Modules:

Importing and using

Python standard library Modules

<https://docs.python.org/3/py-modindex.html>

Third Party Modules: Scipy, PyWin

<http://www.scipy.org>

Creating our own Modules

Modules are python scripts which contain code

Random Module:  
def createRandomNumber():

ranNum = rand.randint(1,100)

print(ranNum)

return ranNum

list shuffle:  
def createSeqIntList():

num = 0

num = int(input("Please enter a number for the sequential list?"))

x = list(range(num))

return x

def shuffleList(list):

print(list)

rand.shuffle(list)

print(list)

createRandomNumber()

result 78

x = createSeqIntList()

result: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

shuffleList(x)

result: [10, 6, 1, 5, 7, 8, 4, 3, 9, 11, 0, 2]

DateTime Module:

def getNow():

now = dt.datetime.now()

return now

result: 2018-01-29 13:31:55.078417

now = getNow()

print(now)

print(str(now.year) + "," + str(now.month) + "," + str(now.day))  
result:

2018-01-29 13:41:05.398028

2018,1,29

OS Module:

Find path:  
def getEnvPath():

path = os.getenv('PATH')

print(path)

result: C:\Program Files (x86)\SciTE\\SciTE.exe;C:\ProgramData\Oracle\Java\javapath;C:\Program Files (x86)\Intel\iCLS Client\;C:\Program Files\Intel\iCLS Client\;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files\Intel\Intel(R) Management Engine Components\DAL;C:\Program Files (x86)\Intel\Intel(R) Management Engine Components\DAL;C:\Program Files\Intel\Intel(R) Management Engine Components\IPT;C:\Program Files (x86)\Intel\Intel(R) Management Engine Components\IPT;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files\Intel\WiFi\bin\;C:\Program Files\Common Files\Intel\WirelessCommon\;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files\dotnet\;C:\Program Files\Microsoft DNX\Dnvm\;C:\Program Files (x86)\Skype\Phone\;C:\Program Files\Microsoft SQL Server\130\Tools\Binn\;C:\Users\Owner\Anaconda3;C:\Users\Owner\Anaconda3\Library\mingw-w64\bin;C:\Users\Owner\Anaconda3\Library\usr\bin;C:\Users\Owner\Anaconda3\Library\bin;C:\Users\Owner\Anaconda3\Scripts;C:\Users\Owner\AppData\Local\Microsoft\WindowsApps

get current working directory:  
def getCurrentWorkingDirectory():

cd = os.getcwd()

print(cd)

return cd

result: C:\Users\Owner\Dropbox\PythonScripts

### Creating New Modules:

sample module:

def greet(n):

i = 0

while i<n:

i+=1

print("Hello World")

def sumNums(\*args):

sum = 0

for n in args:

sum += n

print(sum)

def myRange(start, stop, step):

i = start

while i <=stop:

yield i

i += step

save to library of python install to import:

then import to script to use:

my.sumNums(10,20,30,40,50)

result:

10

30

60

100

150

x = my.myRange(1, 20,3)

print(list(x))

result: [1, 4, 7, 10, 13, 16, 19]

Modules can have data as well

### Exception Handling:

Exceptions are Run-time errors

examples: file not found, divide by zero, invalid input etc

Try:-Except

<https://docs.python.org/3/library/exceptions.html>

### Relational Database Basics:

RDBMS: Relational Database Management System( Oracle, Sybase, DB2, MySql)

in Relational Database, data is organized in tables, rows and columns

SQLite3

Most Important Operations – Select, Insert, Update, Delete

SQL – Structured Query Language

**id name course score**

101 John Python 101 98

102 Tom Python 101 88

103 Kris Python 101 95

104 Gary Python 101 92

105 Cathy Python 101 90

Sample Queries:  
SELECT name, score

FROM dbo.student

returns all rows and returns the name and score

INSERT dbo.student(name, score)

VALUES(‘Alex’, 91)

UPDATE dbo.student

SET score = 90

WHERE name = ‘Tom’

DELETE dbo.student

WHERE name = ‘Gary’

CREATE table student(id int, name text, score int)

SELECT \*

FROM dbo.student

ORDER BY name

Practise:

import sqlite3

if \_\_name\_\_ == "\_\_main\_\_":  
 db = sqlite3.connect("Student.db")

db.execute("drop table if exists grades")

db.execute("create table grades(id int, name text, score int)")

db.execute("insert into grades(id,name,score) values(101,'John',99)")

db.execute("insert into grades(id,name,score) values(102,'Gary',90)")

db.execute("insert into grades(id,name,score) values(103,'James',80)")

db.execute("insert into grades(id,name,score) values(104,'Cathy',85)")

db.execute("insert into grades(id,name,score) values(105,'Kris',95)")

db.commit()

result = db.execute("select name,score from grades order by score")

for row in result:

print(row)

print(“-“ \* 50)

result = db.execute(“select \* from grades where name = ‘Gary’”)

db.close()

Results:

('James', 80)

('Cathy', 85)

('Gary', 90)

('Kris', 95)

('John', 99)

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

(102, 'Gary', 90)

### Regular Expressions:

Regular expression is a sequence of characters that define a search pattern

examples:  
r’cat’

r’ca+t’,

r’[a-z][\d]{ 3,5}’

These are typically used in applications that involve a lot of text processing.

These are also an integral part of UNIX command line utilities, like sed, grep and awk.

Perl, Ruby and Awk include support for re in the core language itself.

Other languages like C, C++ and Python, support re through extension libraries.

Python standard library provides a module called “re” for Regular Expressions.

Python syntax of regular expressions is very similar to Perl.

Regular expressions are also referred as ‘regex’ or ‘regexp’

Learning Regular Expressions involves 2 steps:

1. Learning how to use re module methods
2. How to write search patterns using meta character

re Module Methods:  
re.search(pattern, string, flags=0)

Looks for match for the pattern, any where is the string.

if match found: Returns a match object. if not Returns None

re.match(pattern, string, flags=0)

looks for a match for the pattern, at the beginning of the string

if match found: Returns a match object. if not Returns None

re.group(), re.groups(), span(), start(), end()

some important methods of Match Object

re.findall(pattern, string, flags=0)

Returns a list of all non-overlapping matches in the string

re.finditer(pattern, string, flags=0)

Returns an iterator over all non-overlapping matches in the string.

For each match, the iterator returns a match object.

sub(pattern, repl, string, count=0, flags=0)

string substitution/replacement

compile(patterns, flags=0)

compiles a regular expression pattern, returns a pattern object

split(pattern, string, flags=0)

a more powerful method than a string split()method.

We can split on multiple delimiters

Examples:

import re  
 str1="This is a string - not a paragraph"

m = re.search(r'is', str1)

print(m.group()) = is

print(m.start(), m.end()) = (2, 4)

print(m.span()) =(2, 4)

#beginning of the string only

print(re.match(r'is', str1))

m1 = re.match(r'Th', str1)

print(m1.group(), m1.start(), m1.end(), m1.span())

results:

None

('Th', 0, 2, (0, 2))

text = "abararabarabar"

r = re.findall(r'ar', text)

print len(r)

print r,

print

results:

4

['ar', 'ar', 'ar', 'ar']

rit = re.finditer(r'ar', text)

print rit

for i in rit:

print(i.group(), i.start(), i.end())

print(sys.version)

results:

('ar', 2, 4)

('ar', 4, 6)

('ar', 8, 10)

('ar', 12, 14)

2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:25:58) [MSC v.1500 64 bit (AMD64)]

text = "abararabarabar"

print re.sub(r'ar', 'ti', text)

print re.sub(r'ar', 'ti', text, count=2)

results:

abtitiabtiabti

abtitiabarabar

pat = re.compile(r'ba')

print re.findall(pat, text)

results:

['ba', 'ba', 'ba']

text3 = "akaks ksdkdkd; aksakks: ajsjss, shshs; ususu; hshs"

test = list(text3.split())

print test

results:

['akaks', 'ksdkdkd;', 'aksakks:', 'ajsjss,', 'shshs;', 'ususu;', 'hshs']

#multiple spliters not just one as in test.split()

print re.split(r'[ ;:,]\s\*', text3)

results:   
['akaks', 'ksdkdkd', 'aksakks', 'ajsjss', 'shshs', 'ususu', 'hshs']

re.search looks for instance(returns first match only )

re.match looks only for the start (begins with)

r = raw string that don’t interpret \ as a python character

how to replace characters in a string easily with re.

sub = re.sub(r'ba', 'it', text2)

print(sub)

sub2 = re.sub(r'ba', 'xy', text2, count=2)

print(sub2)

compile creates a pattern to use in search, match, findall, finditer, any search

pat = re.compile(r'ba')

sub3 = re.sub(pat, 'sh', text2)

print(sub3)

### Writing RE Patterns:

1. Repetitions
2. Non Greedy Repetitions
3. Character sets
4. Character Ranges
5. Escape Codes
6. Anchoring
7. Flags
8. Groups and Named Groups

#### Repetitions (.\*+?{m}{m,n}):

‘ab\*’ : a followed by zero or more b’s. (\* preceding character) GREEDY

‘ab+’ : a followed by one or more b’s. (+ one or more preceding character) GREEDY

‘ab?’ : a followed by zero or one b. (? zero or one preceding character) GREEDY

‘ab{n}’ : a followed by n b’s , ({2} a followed by 2 b’s (n times of preceding character)

‘ab{m,n}’ : a followed by min m and max n b’s.

‘ab{m,}’ : a followed by min m and unlimited b’s.

‘ab.’ : ab followed by a single non newline character.

Examples:

# followed by zero or more bs

print("using ab\* repitition")

pat = re.compile(r'ab\*')

m= re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

result:

using ab\* repetition

('a', (0, 1))

['a', 'a', 'abbbbb', 'a', 'a', 'abbbb', 'ab', 'abb', 'ab', 'abbbb', 'ab', 'a']

# followed by one or more b

print("using ab+ repitition")

pat1 = re.compile(r'ab+')

m = re.search(pat1,text)

m1 = re.findall(pat1,text)

print(m.group(), m.span())

print(m1)

result:

using ab+ repetition

('abbbbb', (2, 8))

['abbbbb', 'abbbb', 'ab', 'abb', 'ab', 'abbbb', 'ab']

# followed by zero or one b

print("using ab? repitition")

pat3 = re.compile(r'ab?')

m = re.search(pat3,text)

m1 = re.findall(pat3,text)

print(m.group(), m.span())

print(m1)

result:

using ab? repetition

('a', (0, 1))

['a', 'a', 'ab', 'a', 'a', 'ab', 'ab', 'ab', 'ab', 'ab', 'ab', 'a']

# followed by n bs

print("using ab{n} repitition")

pat4 = re.compile(r'ab{2}')

m = re.search(pat4,text)

m1 = re.findall(pat4,text)

print(m.group(), m.span())

print(m1)

result:

using ab{2} repetition

('abb', (2, 5))

['abb', 'abb', 'abb', 'abb']

# followed by min m and max n bs

print("using ab{m,n} repitition")

pat5 = re.compile(r'ab{1,3}')

m = re.search(pat5,text)

m1 = re.findall(pat5,text)

print(m.group(), m.span())

print(m1)

result:

using ab{1,3} repetition

('abbb', (2, 6))

['abbb', 'abbb', 'ab', 'abb', 'ab', 'abbb', 'ab']

# followd by min m, unlimited b

print("using ab{m,} repitition")

pat6 = re.compile(r'ab{1,}')

m = re.search(pat6,text)

m1 = re.findall(pat6,text)

print(m.group(), m.span())

print(m1)

result:

using ab{1,} repetition

('abbbbb', (2, 8))

['abbbbb', 'abbbb', 'ab', 'abb', 'ab', 'abbbb', 'ab']

# followed by a single non newline character

print("using ab. repitition")

pat7 = re.compile(r'ab.')

m = re.search(pat7, text)

m1 = re.findall(pat7, text)

print(m.group(), m.span())

print(m1)

result:

using ab. repetition

('abb', (2, 5))

['abb', 'abb', 'aba', 'aba', 'abc']

Greedy Repetitions:  
\* + ?

Try to match as much as possible

A way to make them not greedy:  
Adding a question mark to the end

Example:  
 print("using ab\*? repetition")

pat = re.compile(r'ab\*?')

m= re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

Results: Non-Greedy

using ab\*? repetition tries to match with the minimum match

('a', (0, 1))

['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']

Greedy

using ab\* repetition

('a', (0, 1))

['a', 'a', 'abbbbb', 'a', 'a', 'abbbb', 'ab', 'abb', 'ab', 'abbbb', 'ab', 'a']

print("using ab+? repetition")

pat1 = re.compile(r'ab+?')

m = re.search(pat1,text)

m1 = re.findall(pat1,text)

print(m.group(), m.span())

print(m1)

results: Non-Greedy

using ab+? repetition

('ab', (2, 4))

['ab', 'ab', 'ab', 'ab', 'ab', 'ab', 'ab']

Greedy:  
using ab+ repetition

('abbbbb', (2, 8))

['abbbbb', 'abbbb', 'ab', 'abb', 'ab', 'abbbb', 'ab']

print("using ab?? repetition")

pat3 = re.compile(r'ab??')

m = re.search(pat3,text)

m1 = re.findall(pat3,text)

print(m.group(), m.span())

print(m1)

result: Non-Greedy

using ab?? repetition

('a', (0, 1))

['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']

Greedy:  
using ab? repetition

('a', (0, 1))

['a', 'a', 'ab', 'a', 'a', 'ab', 'ab', 'ab', 'ab', 'ab', 'ab', 'a']

### Character Sets & Ranges

A character set is a group of characters enclosed in a square brackets[], any one of which can match at that point in the pattern.

As character sets grow larger, typing every character that should match could become tedious. A more compact format is using character ranges.

examples:

a[xy] : would match either ax or ay

a[^xy] : would exclude ax and ay. ^ excludes characters in []

[a-k] : would match with any character between a and k

[^p-z] : exclude matches with any character between p and z

Examples:

text = "xyyxaxyyyzzazx"

pat = re.compile(r'[xy]')

m = re.search(pat, text)

print(m.group(), m.span())

m1 = re.findall(pat, text)

print(m1)

results:

using [xy] pattern

('x', (0, 1))

['x', 'y', 'y', 'x', 'x', 'y', 'y', 'y', 'x']

print("using [^xy] pattern")

pat1 = re.compile(r'[^xy]')

m = re.search(pat1, text)

print(m.group(), m.span())

m1 = re.findall(pat1, text)

print(m1)

results:

using [^xy] pattern

('a', (4, 5))

['a', 'z', 'z', 'a', 'z']

print("using [a-k] pattern")

pat1 = re.compile(r'[a-k]')

m = re.search(pat1, text)

print(m.group(), m.span())

m1 = re.findall(pat1, text)

print(m1)

using [a-k] pattern

('a', (4, 5))

['a', 'a']

print("using [l-z] pattern")

pat1 = re.compile(r'[l-z]')

m = re.search(pat1, text)

print(m.group(), m.span())

m1 = re.findall(pat1, text)

print(m1)

results:

using [l-z] pattern

('x', (0, 1))

['x', 'y', 'y', 'x', 'x', 'y', 'y', 'y', 'z', 'z', 'z', 'x']

print("using x[xy] pattern")

pat1 = re.compile(r'x[xy]')

m = re.search(pat1, text)

print(m.group(), m.span())

m1 = re.findall(pat1, text)

print(m1)

results:

using x[xy] pattern

('xy', (0, 2))

['xy', 'xy']

print("using x[xy]+ pattern")

pat1 = re.compile(r'x[xy]+')

m = re.search(pat1, text)

print(m.group(), m.span())

m1 = re.findall(pat1, text)

print(m1)

results:

using x[xy]+ pattern

('xyyx', (0, 4))

['xyyx', 'xyyy']

text = "xxy xyxyx xaxb xxyy aaxz"

print("using x[^xy]+ pattern")

pat1 = re.compile(r'x[^xy]+')

m = re.search(pat1, text)

print(m.group(), m.span())

m1 = re.findall(pat1, text)

print(m1)

results:

using x[^xy]+ pattern

('x ', (8, 10))

['x ', 'xa', 'xb ', 'xz']

text1 = "This is a sample text. -- with some Punctuation marks!!!"

print("using '[A-Z][a-z]' pattern")

pat1 = re.compile(r'[A-Z][a-z]\*')

m = re.search(pat1, text1)

print(m.group(), m.span())

m1 = re.findall(pat1, text1)

print(m1)

results:  
using '[A-Z][a-z]' pattern

('This', (0, 4))

['This', 'Punctuation']

print("using '[^.\-! ]+' pattern")

pat1 = re.compile(r'[^.\-! ]+')

m = re.search(pat1, text1)

print(m.group(), m.span())

m1 = re.findall(pat1, text1)

print(m1)

results:  
using '[^.\-! ]' pattern

('This', (0, 4))

['This', 'is', 'a', 'sample', 'text', 'with', 'some', 'Punctuation', 'marks']

### Escape Codes, Anchoring and Flags:

We can use special escape codes to find specific types of patterns in data, such as digits, non-digits, whitespace etc.

Escape Codes:

\d: matches single digit

\D: Matches a single non-digit

\w: matches a single alphanumeric character

\W: matches a single non-alphanumeric character

\s: Matches a single whitespace character (tab, space, newline, etc)

\S: Matches a single non-whitespace character

\b: Matches on word boundary

Examples:

text = "The cost of Python course is $125."

print("using '\d'")

pat = re.compile(r'\d')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

results:

('1', (30, 31))

['1', '2', '5']

print("using '\D'")

pat = re.compile(r'\D')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

results:

using '\D'

('T', (0, 1))

['T', 'h', 'e', ' ', 'c', 'o', 's', 't', ' ', 'o', 'f', ' ', 'P', 'y', 't', 'h', 'o', 'n', ' ', 'c', 'o', 'u', 'r', 's', 'e', ' ', 'i', 's', ' ', '$', '.']

print("using '\w'")

pat = re.compile(r'\w')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

Results:

using '\w'

('T', (0, 1))

['T', 'h', 'e', 'c', 'o', 's', 't', 'o', 'f', 'P', 'y', 't', 'h', 'o', 'n', 'c', 'o', 'u', 'r', 's', 'e', 'i', 's', '1', '2', '5']

print("using '\W'")

pat = re.compile(r'\W')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

Results:

using '\W'

(' ', (3, 4))

[' ', ' ', ' ', ' ', ' ', ' ', '$', '.']

print("using '\s'")

pat = re.compile(r'\s')

m = re.search(pat, text)

m1 = re.findall(pat, text)

#print(m.group(), m.span())

print(m1)

results:

using '\s'

(' ', (3, 4))

[' ', ' ', ' ', ' ', ' ', ' ']

print("using '\S'")

pat = re.compile(r'\S')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

results:

using '\S'

('T', (0, 1))

['T', 'h', 'e', 'c', 'o', 's', 't', 'o', 'f', 'P', 'y', 't', 'h', 'o', 'n', 'c', 'o', 'u', 'r', 's', 'e', 'i', 's', '$', '1', '2', '5', '.']

print("using '\b'")

pat = re.compile(r'\b')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

Results:

using '\b'

('', (0, 0))

['', '', '', '', '', '', '', '', '', '', '', '', '', '']

print("using '\B'")

pat = re.compile(r'\B')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

('', (1, 1))

['', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '']

print("using '\d+'")

pat = re.compile(r'\d+')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

result:

using '\d+'

('125', (30, 33))

['125']

using '\D+'

('The cost of Python course is $', (0, 30))

['The cost of Python course is $', '.\t']

Anchoring:

In addition to describing the content of a pattern to match, the relative location can be specified in the input text where the pattern should appear by using anchoring instructions.

^ = Start of a string

$ = end of a string

\A = Start of a string

\z = end of a string

\b = empty string at the beginning and end of a word.

Examples:

text1 = "This is a beautiful day"

print("using r'^is' pattern")

pat = re.compile(r'^is')

m = re.search(pat, text1)

m1 = re.findall(pat, text1)

#print(m.group(), m.span())

print(m1)

results:

using r'^is' pattern

[]

print("using r'^T' pattern")

pat = re.compile(r'^T')

m = re.search(pat, text1)

m1 = re.findall(pat, text1)

print(m.group(), m.span())

print(m1)

results:

using r'^T' pattern

('T', (0, 1))

['T']

print("using r'\.$' pattern")

pat = re.compile(r'\.$')

m = re.search(pat, text1)

m1 = re.findall(pat, text1)

print(m.group(), m.span())

print(m1)

results;

using r'\.$' pattern

('.', (23, 24))

['.']

print("using r'\bis\b' pattern")

pat = re.compile(r'\bis\b')

m = re.search(pat, text1)

m1 = re.findall(pat, text1)

print(m.group(), m.span())

print(m1)

using r'\bis\b' pattern

('is', (5, 7))

['is']

FLAGS:

Sometimes we need to slightly tweak the behavior of the regular expression. the regular expression engine in Python offer a small number of flags that modify the behavior of the entire expression.

re.IGNORECASE or re.I or 2:

case-insensitive

re.DOTALL or re.S or 16:

makes . character to include\n new line character.

re.MULTILINE or re.M or 8:

maks the ^ and $ characters, which normally would only match against the beginning or end of the string, to instead match against the beginning or end of any line within the string.

re.VERBOSE or re.X or 64:

Makes complicated regular expressions to be more readable. This flag does two things:

1. it causes all whitespace(other than in character classes) to be ignored, including line breaks.
2. it treats the # character (again unless its inside a character class) as a comment character.

re.DEBUG or 128:

provides some debugging information while compiling a regular expression

Multiple Flags:

some times we may have to use multiple flags at the same time, this is done using bitwise OR Operator

example: re.I | re.S | re.M

Examples:

text = "Python python PYTHON pYtHoN"

print("using r'Python'")

pat = re.compile(r'Python')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.span())

print(m1)

results:

using r'Python'

('Python', (0, 6))

['Python']

print("using r'Python'")

pat = re.compile(r'Python')

m = re.search(pat, text, re.I)

m1 = re.findall(pat, text, re.I)

print(m.group(), m.span())

print(m1)

results:

using r'Python'

('Python', (0, 6))

['Python', 'python', 'PYTHON', 'pYtHoN']

print("using r'.+'")

#pat = re.compile(r'Python')

m = re.search(r'.+', text, re.DOTALL)

m1 = re.findall(r'.+', text, re.DOTALL)

print(m.group(), m.span())

print(m1)results:

results:

Py

thon.

using r'.+'

('Py\nthon.', (0, 8))

['Py\nthon.']

pat = re.compile(r'Py')

print(re.sub(pat, 'My', text))

print(re.sub(r'Py', 'My', text, flags=2))

results:

Python is fun. Learning python.

Mython is fun. Learning python.

Mython is fun. Learning Mython.

### Groups and Named Groups:

Regular Expressions provide a mechanism to split the express into groups. When using groups, we will be able to select each individual group within the match in addition to getting the entire match. You can specify groups within a regular expression by using parenthesis.

Examples:

text = "123-4567 is my telephone."

pat = re.compile(r'[\d]{3}-[\d]{4}')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group())

print(m1)

results:

123-4567

['123-4567']

pat = re.compile(r'([\d]{3})-([\d]{4})')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.groups(), m.group(1), m.group(2))

print(m1)

results:

('123-4567', ('123', '4567'), '123', '4567')

[('123', '4567')]

pat = re.compile(r'(?P<first3>[\d]{3})-(?P<last4>[\d]{4})')

m = re.search(pat, text)

m1 = re.findall(pat, text)

print(m.group(), m.groups(), m.group('first3'), m.group('last4'))

print(m1)

Results:

('123-4567', ('123', '4567'), '123', '4567')

[('123', '4567')]

Practical RE Session:

text= ['123 456 7890', '(123) 456 7890']

pat = r'\d{3}\s\d{3}\s\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print("we got the first phone number ", m.group())

result:  
'123 456 7890'

pat = r'\(\d{3}\)\s\d{3}\s\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print("we got the second phone number ", m.group())

results:

'(123) 456 7890

pat = r'\(?\d{3}\)?\s\d{3}\s\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print(m.group())

results:

123 456 7890

(123) 456 7890

text= ['123 456 7890', '(123) 456 7890', '123-466-7890']

pat = r'\(?\d{3}\)?\s\d{3}\s\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print(m.group())

results:

123 456 7890

(123) 456 7890

pat = r'\(?\d{3}\)?[\s\-]\d{3}[\s\-]\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print(m.group())

results:

123 456 7890

(123) 456 7890

123-466-7890

text= ['123 456 7890', '(123) 456 7890', '123-466-7890', '123.456.7890']

pat = r'\(?\d{3}\)?[\s\-\.]\d{3}[\s\-\.]\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print(m.group())

print('-'\*40)

results:

123 456 7890

(123) 456 7890

123-466-7890

123.456.7890

text= ['123 456 7890', '(123) 456 7890', '123-466-7890', '123.456.7890', '1234567890']

pat = r'\(?\d{3}\)?[\s\-\.]?\d{3}[\s\-\.]?\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print(m.group())

results:

123 456 7890

(123) 456 7890

123-466-7890

123.456.7890

1234567890

text= ['1 123 456 7890','123 456 7890','+1 (123) 456 7890', '(123) 456 7890', '123-466-7890', '123.456.7890', '1234567890']

pat = r'\+?\d?\s?\(?\d{3}\)?[\s\-\.]?\d{3}[\s\-\.]?\d{4}'

for dt in text:

m = re.search(pat, dt)

if m:

print(m.group())

print('-'\*40)

results:  
1 123 456 7890

123 456 7890

+1 (123) 456 7890

(123) 456 7890

123-466-7890

123.456.7890

1234567890

pat = r'\+?\d?\s?\(?\d{3}\)?[\s\-\.]?\d{3}[\s\-\.]?\d{4}'

patc = re.compile(pat)

for dt in text:

m = re.search(patc, dt)

if m:

print(m.group())

print('-'\*40)

results:

1 123 456 7890

123 456 7890

+1 (123) 456 7890

(123) 456 7890

123-466-7890

123.456.7890

1234567890

### NUMPY

Arrays are not part of core Python language

NumPy is an extension to the Python programming language, adding support for large, multi-dimensional array and matrices.

NumPy includes a large library of high-level mathematical functions to operate on these arrays.

It is part of the SciPy Stack

<http://www.scipy.org/>

Getting NumPy Module:

on a terminal window or shell

pip install numpy

Works with Python 2.7 not so much with Jupiter Notebooks and Anaconda