Github: https://github.com/yashchinchole/Python

Basic Arithmetic

In [1]:
```
input('Enter Something into this box: ')
```

Out[1]:
```
Enter Something into this box: YASH
'YASH'
```

In [2]:
```
# Addition
2+1
```

Out[2]:
```
3
```

In [3]:
```
# Subtraction
2-1
```

Out[3]:
```
1
```

In [4]:
```
# Multiplication
2*2
```

Out[4]:
```
4
```

In [5]:
```
# Division
3/2
```

Out[5]:
```
1.5
```

In [6]:
```
# Floor Division
7//4
```

Out[6]:
```
1
```

In [7]:
```
# Modulo
7%4
```

Out[7]:
```
3
```

In [8]:
```
# Powers
2**3
```

Out[8]:
```
8
```

In [9]:
```
# Can also do roots this way
4**0.5
```

Out[9]:
```
2.0
```

Variable Assignment

In [10]:
```
my_dogs1 = 2
my_dogs1
```

Out[10]:
```
2
```

In [11]:
```
type(my_dogs1)
```

Out[11]:
```
int
```

In [12]:
```
my_dogs2 = ['Sammy', 'Frankie']
my_dogs2
```

Out[12]:
```
['Sammy', 'Frankie']
```

In [13]:
```
type(my_dogs2)
```

Out[13]:
```
list
```

Strings

In [14]:
```
s = 'Hello World'
print('\n')
print(s)
```

```

Hello World
```

In [15]:
```
len('Hello World')
```

Out[15]:
```
11
```

In [16]:
```
# Show first element (in this case a letter)
s[0]
```

Out[16]:
```
'H'
```

In [17]:
```
# Grab everything past the first term all the way to the length of s which is len(s)
s[1:]
```

Out[17]:
```
'ello World'
```

In [18]:
```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[18]:
```
'Hel'
```

In [19]:
```
#Everything
s[:]
```

Out[19]:
```
'Hello World'
```

In [20]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[20]:
```
'd'
```

In [21]:
```
# Grab everything but the last letter
s[:-1]
```

Out[21]:
```
'Hello Worl'
```

In [22]:
```
# Grab everything, but go in step sizes of 2
s[::2]
```

Out[22]:
```
'HloWrd'
```

In [23]:
```
# We can use this to print a string backwards
s[::-1]
```

Out[23]:
```
'dlroW olleH'
```

In [24]:
```
# Concatenate strings!
s + ' concatenate me!'
```

Out[24]:
```
'Hello World concatenate me!'
```

In [25]:
```
s
```

Out[25]:
```
'Hello World'
```

In [26]:
```
# We can reassign s completely though!
s = s + ' concatenate me!'
s
```

Out[26]:
```
'Hello World concatenate me!'
```

In [27]:
```
# Upper Case a string
s.upper()
```

Out[27]:
```
'HELLO WORLD CONCATENATE ME!'
```

In [28]:
```
# Split a string by blank space (this is the default)
s.split()
```

Out[28]:
```
['Hello', 'World', 'concatenate', 'me!']
```

In [29]:
```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

Out[29]:
```
['Hello ', 'orld concatenate me!']
```

In [30]:
```
letter = 'z'
letter*10
```

Out[30]:
```
'zzzzzzzzzz'
```

In [31]:
```
x, y = 'some', 'more'
print("I'm going to inject %s text here, and %s text here."%(x,y))
```

```
I'm going to inject some text here, and more text here.
```

In [32]:
```
print('I wrote %s programs today.' %3.75)
print('I wrote %d programs today.' %3.75)
```

```
I wrote 3.75 programs today.
I wrote 3 programs today.
```

In [33]:
```
print('Floating point numbers: %10.2f' %(13.144))
```

```
Floating point numbers:      13.14
```

In [34]:
```
print('The {2} {1} {0}'.format('fox','brown','quick'))
```

```
The quick brown fox
```

In [35]:
```
print('First Object: {a}, Second Object: {b}, Third Object: {c}'.format(a=1,b='Two',c=
```

```
First Object: 1, Second Object: Two, Third Object: 12.3
```

In [36]:
```
print('{0:<8} | {1:^8} | {2:>8}'.format('Left','Center','Right'))
print('{0:<8} | {1:^8} | {2:>8}'.format(11,22,33))
```

```
Left     |  Center  |    Right
11       |    22    |       33
```

In [37]:
```
name = 'Fred'
print(f"He said his name is {name!r}")
```

```
He said his name is 'Fred'
```

List

In [38]:
```
my_list = ['A string',23,100.232,'o']
my_list
```

Out[38]:
```
['A string', 23, 100.232, 'o']
```

In [39]:
```
len(my_list)
```

Out[39]:
```
4
```

In [40]:
```
# Grab everything UP TO index 3
my_list[:3]
```

Out[40]:
```
['A string', 23, 100.232]
```

In [41]:
```
# Reassign
my_list = my_list + ['add new item permanently']
```

```
my_list
```

Out[41]:
```
['A string', 23, 100.232, 'o', 'add new item permanently']
```

In [42]:
```
# Create a new list
list1 = [1,2,3]
```

In [43]:
```
# Append
list1.append('append me!')
list1
```

Out[43]:
```
[1, 2, 3, 'append me!']
```

In [44]:
```
# Pop off the 0 indexed item
list1.pop(0)
list1
```

Out[44]:
```
[2, 3, 'append me!']
```

In [45]:
```
new_list = ['a','e','x','b','c']
new_list
```

Out[45]:
```
['a', 'e', 'x', 'b', 'c']
```

In [46]:
```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
new_list
```

Out[46]:
```
['c', 'b', 'x', 'e', 'a']
```

In [47]:
```
# Use sort to sort the list (in this case alphabetical order, but for numbers it will
new_list.sort()
new_list
```

Out[47]:
```
['a', 'b', 'c', 'e', 'x']
```

In [48]:
```
# Let's make three lists
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]

# Make a list of lists to form a matrix
matrix = [lst_1,lst_2,lst_3]
matrix
```

Out[48]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [49]:
```
# Square numbers in range and turn into list
lst = [x**2 for x in range(0,11) if x%2 == 0]
lst
```

Out[49]:
```
[0, 4, 16, 36, 64, 100]
```

Dictionaries

In [50]:
```
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

# Can then even call methods on that value
my_dict['key3'][0].upper()
```

Out[50]:
```
'ITEM0'
```

In [51]:
```
# Create a new dictionary
d = {}
# Create a new key through assignment
d['animal'] = 'Dog'
# Can do this with any object
d['answer'] = 42
#Show
d
```

Out[51]:
```
{'animal': 'Dog', 'answer': 42}
```

In [52]:
```
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}

# Keep calling the keys
d['key1']['nestkey']['subnestkey']
```

Out[52]:
```
'value'
```

In [53]:
```
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}
d
```

Out[53]:
```
{'key1': 1, 'key2': 2, 'key3': 3}
```

In [54]:
```
# Method to return a list of all keys
d.keys()
```

Out[54]:
```
dict_keys(['key1', 'key2', 'key3'])
```

In [55]:
```
# Method to grab all values
d.values()
```

Out[55]:
```
dict_values([1, 2, 3])
```

In [56]:
```
# Method to return tuples of all items  (we'll learn about tuples soon)
d.items()
```

Out[56]:
```
dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

Tuples

In [57]:
```
# Can also mix object types
t = ('one','one',2)

# Show
t
```

Out[57]:
```
('one', 'one', 2)
```

In [58]:
```
# Use indexing just like we did in lists
t[0]
```

Out[58]:
```
'one'
```

In [59]:
```
# Use .index to enter a value and return the index
t.index('one')
```

Out[59]:
```
0
```

In [60]:
```
# Use .count to count the number of times a value appears
t.count('one')
```

Out[60]:
```
2
```

Set

In [61]:
```
x = set()
# We add to sets with the add() method

x.add(1)
x.add(2)

x
```

Out[61]:
```
{1, 2}
```

In [62]:
```
# Create a list with repeats
list1 = [1,1,2,2,3,4,5,6,1,1]

# Cast as set to get unique values
set(list1)
```

Out[62]:
```
{1, 2, 3, 4, 5, 6}
```

Files

In [63]:
```
%%writefile test.txt
Hello, this is a quick test file.
```

```
Overwriting test.txt
```

Alternatively, to grab files from any location on your computer, simply pass in the entire file path.

For Windows you need to use double \ so python doesn't treat the second \ as an escape character, a file path is in the form:

myfile = open("C:\Users\YourUserName\Home\Folder\myfile.txt")

In [64]:
```
# Open the text.txt we made earlier
my_file = open('test.txt')

# We can now read the file
my_file.read()
```

Out[64]:
```
'Hello, this is a quick test file.\n'
```

In [65]:
```
# But what happens if we try to read it again?
my_file.read()
```

Out[65]:
```
''
```

In [66]:
```
# Seek to the start of file (index 0)
my_file.seek(0)
```

Out[66]:
```
0
```

In [67]:
```
# Now read again
my_file.read()
```

Out[67]:
```
'Hello, this is a quick test file.\n'
```

In [68]:
```
# Readlines returns a list of the lines in the file
my_file.seek(0)
my_file.readlines()
```

Out[68]:
```
['Hello, this is a quick test file.\n']
```

In [69]:
```
my_file.close()
```

In [70]:
```
# Add a second argument to the function, 'w' which stands for write.
# Passing 'w+' lets us read and write to the file

my_file = open('test.txt','w+')
```

In [71]:
```
# Write to the file
my_file.write('This is a new line')
```

Out[71]:
```
18
```

In [72]:
```
my_file = open('test.txt','a+')
my_file.write('\nThis is text being appended to test.txt')
my_file.write('\nAnd another line here.')

my_file.seek(0)
print(my_file.read())
```

```
This is a new line
This is text being appended to test.txt
And another line here.
```

In [73]:
```
%%writefile test.txt
First Line
Second Line
```

```
Overwriting test.txt
```

In [74]:
```
for line in open('test.txt'):
    print(line)
```

```
First Line

Second Line
```

if, elif, else

In [75]:
```
person = 'George'

if person == 'Sammy':
    print('Welcome Sammy!')
elif person =='George':
    print('Welcome George!')
else:
    print("Welcome, what's your name?")
```

```
Welcome George!
```

for Loop

In [76]:
```
list1 = [1,2,3,4,5,6,7,8,9,10]

for num in list1:
    if num % 2 == 0:
        print(num)
    else:
        print('Odd number')
```

```
Odd number
2
Odd number
4
Odd number
6
Odd number
8
Odd number
10
```

In [77]:
```
for letter in 'This is a string.':
    print(letter)
```

```
T
h
i
s

i
s

a

s
t
r
i
n
g
.
```

In [78]:
```
list2 = [(2,4),(6,8),(10,12)]
for tup in list2:
    print(tup)
```

```
(2, 4)
(6, 8)
(10, 12)
```

In [79]:
```
d = {'k1':1,'k2':2,'k3':3}
for item in d:
    print(item)
```

```
k1
k2
k3
```

In [80]:
```
# Dictionary unpacking
for k,v in d.items():
    print(k)
    print(v)
```

```
k1
1
k2
2
k3
3
```

In [81]:
```
list(d.keys())
```

Out[81]:
```
['k1', 'k2', 'k3']
```

In [82]:
```
sorted(d.values())
```

Out[82]:
```
[1, 2, 3]
```

while Loop

In [83]:
```
x = 0
```

```
while x < 10:
    print('x is currently: ',x)
    x+=1
    if x==3:
        print('Breaking because x==3')
        break
    else:
        print('continuing...')
        continue
```

```
x is currently:  0
continuing...
x is currently:  1
continuing...
x is currently:  2
Breaking because x==3
```

In [84]:
```
list(range(0,12))
```

Out[84]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [85]:
```
list(range(0,101,10))
```

Out[85]:
```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [86]:
```
for i,letter in enumerate('abcde'):
    print("At index {} the letter is {}".format(i,letter))
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

In [87]:
```
mylist1 = [1,2,3,4,5]
mylist2 = ['a','b','c','d','e']
list(zip(mylist1,mylist2))
```

Out[87]:
```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

In [88]:
```
'x' in ['x','y','z']
```

Out[88]:
```
True
```

In [89]:
```
mylist = [10,20,30,40,100]
min(mylist)
```

Out[89]:
```
10
```

In [90]:
```
from random import shuffle
# This shuffles the list "in-place" meaning it won't return
# anything, instead it will effect the list passed
shuffle(mylist)
mylist
```

Out[90]:
```
[100, 10, 20, 30, 40]
```

In [91]:
```
from random import randint
# Return random integer in range [a, b], including both end points.
randint(0,100)
```

Out[91]:
```
12
```

Methods

In [92]:
```
lst = [1,2,3,4,5]
```

The methods for a list are:

append count extend insert pop remove reverse sort

Functions

In [93]:
```
def greeting(name):
    print(f'Hello {name}')
greeting('Jose')
```

```
Hello Jose
```

In [94]:
```
def add_num(num1,num2):
    return num1+num2
add_num(4,5)
```

Out[94]:
```
9
```

map function

The map function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list.

In [95]:
```
def splicer(mystring):
    if len(mystring) % 2 == 0:
        return 'even'
    else:
        return mystring[0]

mynames = ['John','Cindy','Sarah','Kelly','Mike']

list(map(splicer,mynames))
```

Out[95]:
```
['even', 'C', 'S', 'K', 'even']
```

filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function

In [96]:
```
def check_even(num):
    return num % 2 == 0

nums = [0,1,2,3,4,5,6,7,8,9,10]

list(filter(check_even,nums))
```

Out[96]:
```
[0, 2, 4, 6, 8, 10]
```

lambda expression

lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs.

In [97]:
```
square = lambda num: num **2
square(2)
```

Out[97]:
```
4
```

In [98]:
```
my_nums = [1,2,3,4]
list(map(lambda num: num ** 3, my_nums))
```

Out[98]:
```
[1, 8, 27, 64]
```

LEGB Rule

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,..

In [99]:
```
# x is local here:
f = lambda x:x**2
```

In [100…
```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

    def hello():
        print('Hello '+name)

    hello()

greet()
```

```
Hello Sammy
```

In [101…
```
print(name)
```

```
This is a global name
```

*args and **kwargs

When a function parameter starts with an asterisk, it allows for an arbitrary number of arguments, and the function takes them in as a tuple of values.

In [102…
```
def myfunc(*args):
    return sum(args)

myfunc(40,60,20)
```

Out[102]:
```
120
```

Similarly, Python offers a way to handle arbitrary numbers of keyworded arguments. Instead of creating a tuple of values, **kwargs builds a dictionary of key/value pairs.

In [103…
```
def myfunc(*args, **kwargs):
    if 'fruit' and 'juice' in kwargs:
        print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit']}")
        print(f"May I have some {kwargs['juice']} juice?")
    else:
        pass

myfunc('eggs','spam',fruit='cherries',juice='orange')
```

```
I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Object Oriented Programming

Attributes

In [104…
```
class Dog:
    def __init__(self,breed):
        self.breed = breed
```

```
sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')

sam.breed
```

Out[104]:
```
'Lab'
```

Methods

In [105…
```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

Inheritance

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

In [106…
```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
```

```
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

In [107…
```
d = Dog()
```

```
Animal created
Dog created
```

In [108…
```
d.eat()
```

```
Eating
```

Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, polymorphism refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in.

In [109…
```
class Dog():

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

    def speak(self):
        print(self.name + " says woof")
```

In [110…
```
class Cat():

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

    def speak(self):
        print(self.name + " says meow")
```

In [111…
```
d = Dog("Lab")
c = Cat("Zex")
```

In [112…
```
d.speak()
```

```
Lab says woof
```

In [113…
```
c.speak()
```

```
Zex says meow
```

In [114…
```
for pet in [d,c]:

    print(type(pet))
    print(type(pet.speak()))
```

```
<class '__main__.Dog'>
Lab says woof
<class 'NoneType'>
<class '__main__.Cat'>
Zex says meow
<class 'NoneType'>
```

Abstract Class

An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals

In [115…

```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Special Methods

The **init**(), **str**(), **len**() and **del**() methods These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

In [116…

```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.page

    def __len__(self):
        return self.pages
```

```
    def __del__(self):
        print("A book is destroyed")
```

In [117…

```
book = Book("Python Rocks!", "Jose Portilla", 159)

#Special Methods
print(book)
print(len(book))
del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```