Dictionaries:

* Objects retrieved by KEY name
* Unordered and cannot be sorted

Lists:

* Objects retrieved by location
* Ordered sequence can be indexed or sliced

Dictionary example

prices={'apples':1.99,'oranges':2.00,'milk':3.50}

prices['apples']

Tuples

Does not support item re-assignment

Why bother using them? Passing around objects in programs, data integrity, when we need to ensure elements aren’t changed

Sets are unordered collections of unique elements

Myset=set()

Myset.add(1)

No repeatability

If else if statements

loc = 'park'

if(loc== 'bank'):

print('At bank')

elif(loc=="park"):

print('Not at bank')

else:

print("At home")

a=False

if(a==True):

print('~Feed me~')

else:

print("Don't feed me, not hungry")

Tuple unpacking with for loops

Suppose mylist = [(1,2),(3,4),(5,6),(7,8)], list of tuples

For a,b in mylist:

Print(a)

Print(b)

//gives us access to both

Can do tuple unpacking with dictionaries as well

d={'k1':1,'k2':2,'k3':3}

for key,value in d:

print(value)

While loop with python

while x!=10:

print(x)

x+=1

break: breaks out of current enclosing loop

continue: Goes to the top of the closest enclosing loop

pass: does nothing at all

Range function

for num in range(4,13,4):

print(num)

list(range(0,10))

for letter in word:

print(f'the letter is {letter} and the index is {count}')

count+=1

count =0

word='abcde'

for item in enumerate(word):

print(item)

//enumerate does our index count for us automatically

if 'x' in ['y']:

print(True)

else:

print(False)

//in keyword operator

Convert string to list style

mylist=[letter for letter in mystring] or mylist=[x for lx in ‘hello’] or mylist = [num for num in range(0,11)]

can perform actions like squaring from this line too

mylist = [num\*\*2 for num in range(0,11)]

mylist = [num\*\*2 for num in range(0,11) if num%2==0]

fahrenheit = [(9/5)\*x+32 for x in celsius]

defining our own function

def name\_name\_of\_function():

1)def naming(name):

'''

Explaining function

'''

print("Hello "+ name)

2) def add(a,b):

'''

adding integers or concatenating strings

'''

return a+b

to provide default variable value for eg

def naming(name=’default’):

3)def dog\_check(x):

return 'dog' in x.lower()

4)def platin(var):

if var[0] in 'aeiou':

var=var+'ay'

else:

var=var[1:]+var[0]+'ay'

return var

When we want to have an unlimited amount of parameters what do we do?

def sum(\*args):

var =0

for num in args:

var=var+num

return var

def myfunc(\*args):

return sum(\*args)\*.05

The star we include in the parameter line is what defines the limit

def newfunc(\*\*kwargs):

if 'fruit' in kwargs:

print('I would like the fruit {}'.format(kwargs['fruit']))

else:

print('I didnt find anything')

newfunc(fruit='apple',veggie='broccoli')

Using both in conjunction

def myfunc(\*args,\*\*kwargs):

print('I would like {} {}'.format(args[0],kwargs['food']))

def myfunc(var):

var=var.lower()

count=0

toRet=""

for item in var:

if count%2==0:

toRet=toRet+item.lower()

count+=1

else:

toRet=toRet+item.capitalize()

count+=1

return toRet

mao and filter LAMBDA expressions

map, to use a function on a list for each element, use a map

map(name-of-func, list-to-operate-on)

filter can be applied to a list based on if elif etc wheres its not changing the the actual values in the list unlike map

lambda num: num\*\*2

usually use these anonymous functions for mapping and filtering purposes

Reassignment of variables and their scope is not as straightforward as you would think it is

class Dog():

#this is a class object attribute, true for all dogs

species='mammal'

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

self.breed = breed

self.name = name

self.spots = spots

#Operations/Methods

def bark(self):

print("Woof, my name is {}".format(self.name))

class Circle():

pi=3.14159

def \_\_init\_\_(self,radius=1):

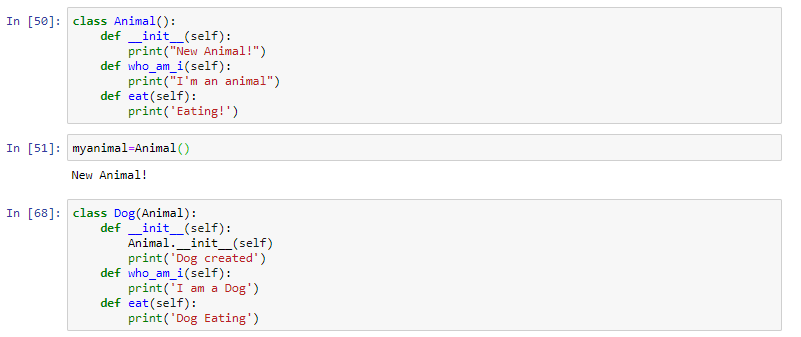
self.radius = radius

def getCircum(self):

return 2\*self.pi\*self.radius

def getArea(self):

return self.radius\*\*2 \* self.pi



class Animal():

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

self.name = name

def speak(self):

raise NotImplementedError('Subclass must implement this abstraction')

When we only want specifically defined classes, we are expected to write speak for our created sub animals from the animal base class

Special Methods

class Book():

def \_\_init\_\_(self,title,author,pages):

self.title=title

self.author=author

self.pages=pages

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

return f"{self.title} by {self.author}"

def \_\_len\_\_(self):

return self.pages

def \_\_del\_\_(self):

print("Book object has been deleted")

when we print out the object, the \_\_str\_\_ function gives us the return output instead of the address

class Account():

def \_\_init\_\_(self,owner,balance):

self.owner=owner

self.balance=balance

def deposit(self,x):

newBalance=self.balance+x

self.balance=newBalance

return "Deposit Accepted"

def withdraw(self,x):

if x>self.balance:

print("Not enough funds")

else:

self.balance=self.balance-x

return "Withdrawal accepted"

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

return f"{self.owner} has {self.balance}"

Module building in github under modules

What is name and main

When we run a py script, the name we give it such as python “one.py” will be our main. This is useful for our logic, we can say if \_\_name\_\_ = “\_\_main\_\_”, which mean its running directly, execute all these following commands, else (when we are for example importing some py files), we can just do a pass or not do anything at all, since al we need them for is a library.

#one.py

def func():

print("FUNC IN ONE.PY")

print(\_\_name\_\_)

print("TOP LEVEL ONE.PY")

if \_\_name\_\_ == "\_\_main\_\_":

print("ONE.PY HAS BEEN RUN DIRECTLY")

#Here we will want to write all the stuff

else:

print("ONE.PY HAS BEEN IMPORTED")

ERROR HANDLING////////

try:

f = open('testfile','w')

f.write("Write a test line")

except TypeError:

print("There was a type error")

except OSError:

print("OSERROR, No permissions for action")

except:

print("all other exceptions")

finally:

print("End of block")

using Error Handling in definitions

def inputInt():

while True:

try:

result = int(input("Input a number"))

except:

print("\nThats not a number")

continue

else:

print("Cool, that works")

break

finally:

print("End of def")

PyLint and unittest libraries which helps us test code and its compatibility

After installing the pylint library, run the command pylint examplefile.py -r y to get a breakdown of what its score is

Using Python decorators to add/remove functionality

Def hello():

Return “hello”

Greet = hello

//after deleting hello

Del hello,

Greet still exists so its not pointing at the old func

def hello(name= "Jose"):

print('The hello function has been executed')

def greet():

return '\t This is the greet function inside hello'

def welcome():

return '\t This is the welcome function inside hello'

print(greet())

print(welcome())

greet and welcome are only defined withing the scope of hello

genrators:

def create\_cube(n):

for x in range(n):

yield x\*\*3

much more efficient, calculating value on call instead of storying memory



Cannot do this inherently with a list

Is s = “hello”

Then we say some\_var = iter(s)

And then we can do the next(some\_var), not possible with s