23rcd March 2020

Tuples are data structures which are ‘read only lists’. Tuples are immutable variant of lists.

--

Import this -> in Python shell will display the Zen of Python by Tim Peters.

--

**Pylint** is a source-code, bug and quality checker for the Python programming language. It is named following a common convention in Python of a "py" prefix, and a nod to the C programming lint program. It follows the style recommended by PEP 8, the Python style guide.[4] It is similar to Pychecker and Pyflakes, but includes the following features:

Checking the length of each line

Checking that variable names are well-formed according to the project's coding standard

Checking that declared interfaces are truly implemented.[5]

It is also equipped with the Pyreverse module that allows UML diagrams to be generated from Python code.

It can be used as a stand-alone program, but also integrates with IDEs such as Eclipse with PyDev[6] and Visual Studio,[7] and editors such as Atom[8], GNU Emacs and Vim.

--

Numbers: Integral; non-integral

Integral : integers, Booleans

Non-integral: real numbers(floats, decimal(gives precision), complex(real and imaginary part), fractions are rational number

--

Collections are: sequences(lists(mutable), tuples, strings(both immutable), sets(mutable, frozen-sets(immutable), maps(dictionary)

Sets and dictionaries are hash maps.

Callables are anything that can be called: functions, generators, classes, class instances (by implementing \_\_call\_\_), instance methods, built-in functions, built-in methods.

Singletons: None, Notimplemented, Ellipses

--

Physical code > logical code > tokenize / NEWLINE token.

Implicit end of line, explicit end of line. Explicit multiline statement has to be ended using backslash ‘\’.

Comments and docstrings are different. Comments are left out when the code is compiled.

Private objects declared with single underscore in the beginning cannot be imported from a module using this syntax. \_var x = 10 // from module import \* // cannot access \_var

\_\_var is used to mangle class attributes. Useful in inheritance chains.

\_\_var\_\_ system defined names that have special meaning to the interpreter.

Naming conventions: packages: shortones, no underscores, lowercase.

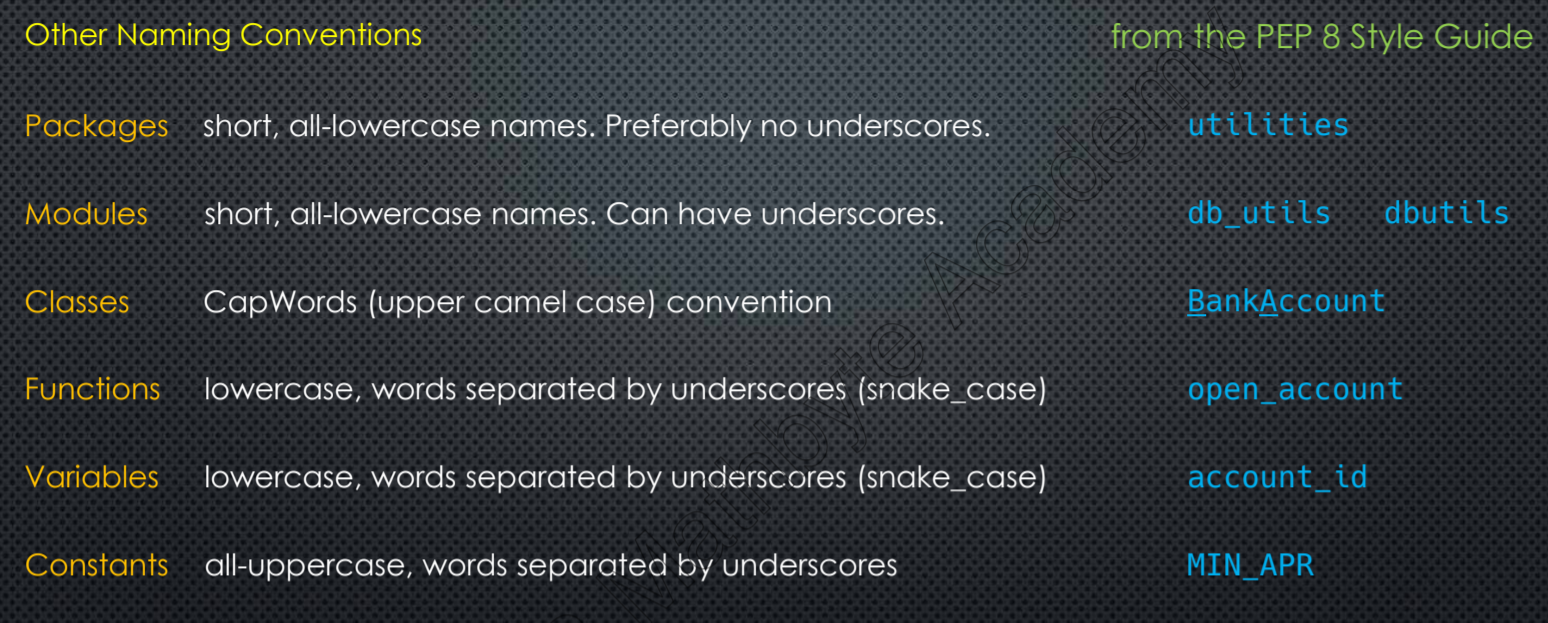
Modules: short, lowercase, can have underscores.

Classes: CapWords or upper camel case

Functions: lowercase, separate words with underscore – snake\_case

Variables: same as functions

Constants: all upper case, words separated by underscore.



Ternary operation: a = 5

b = 'a < 10' if a < 10 else 'a > 10'

print(b)

function name is the variable assign to the function code.

When a variable is assigned to a lambda function, that variable is the name of the lambda function.

Do .. while is not there in Python. Instead use ‘while(true)’

a = 1

while True: # like 'do' while

a += 1

print(a)

if a > 10:

break

‘continue’ statement skips the remaining code and continue with the next iteration.

While .. else – ‘else’ will execute only when the ‘while’ is iterated over without break.

l = [1,2,3]

idx = 0

while idx < len(l):

if l[idx] == 10:

break

idx += 1

else:

l.append(10)

print(l)

The closest resemblance for a ‘switch case’ statement in Python is ‘elif’.

Functions can be defined in Python in 2 ways. Using ‘def’ and ‘lambda’. A name has to be assigned for a lambda function if wanted to reuse it. It’s a single line statement function.

While loop repeats the code block till the condition is true.

‘break’ statement terminates the enclosing loop.

‘continue’ skips the remaining codes and repeat from the beginning.

**Iterable** is an object capable of returning values one at a time.

Lists, tuples, strings are iterables.

# for loop. Here 'i' is requesting for the next value from the iterable object.

for i in [10,15,12,5,20]:

print(i)

for also has an ‘else’ clause, which runs once the loop completes without a break.

Sets and dictionaries are iterables, but they are not ordered and so not indexed.

When referencing a class instance object, ‘str’ will look for the objects class and its memory address by default.

\_\_eq\_\_ to compare objects of a class.

Property setter and getter

class Rectangle:

def \_\_init\_\_(self, width, height):

self.\_width = width

self.\_height = height

def area(self):

return self.width \* self.height

# property getter - method name is same as property name

@property

def width(self):

return self.\_width

@property

def height(self):

return self.\_height

#property setter - method name is same as the property name

@width.setter

def width(self, width):

if width <= 0:

raise ValueError("Width should be greater than zero")

self.\_width = width

@height.setter

def height(self, height):

if height <= 0:

raise ValueError("Height should be greater than zero")

self.\_height = height

# checks if two objects are equal

def \_\_eq\_\_(self, other):

if isinstance(other, Rectangle):

if (self.\_width, self.\_height) == (other.\_width, other.\_height):

return True

return False

return "Can't compare instances of different classes"

# when checking str(object)

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

return "Rectangle object"

# object building details

def \_\_repr\_\_(self):

return "{0}, {1}".format(self.\_width, self.\_height)

obj = Rectangle(5,15)

obj.width = 25

obj.height = 50

print(obj.area())

print(obj.height)

output: 1250

50

**Python is a dynamically typed language. Everything in python are objects**

Id function is used to refer the memory address of the variable. Hex(id(var))

Sys.getrefcount(var) // ctypes.c\_long.from\_address(address).value

When the variable reference to the object having circular reference goes away, the garbage collector will remove the objects with circular references. Memory leak

The gc module. By default, gc is turned on. Object destructor is implemented by \_\_del\_\_

Gc.get\_objects() // gc.disable() // gc.collect

Stack is a linear data structure whereas Heap is a hierarchical data structure. Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed. Stack accesses local variables only while Heap allows you to access variables globally.

An object, whose internal state can be changed, called mutable.

Immutable objects: numbers, strings, tuples, frozen-sets, user defined classes.

Mutable objects: lists, sets, dictionaries, user defined class

**A tuple containing a list elements can be modified.** T = ([1,2],[4,5,6]) t[0].append(3)

lst = [[1,2,3], [5,6,7]]

t = (lst, [8,9])

print(t)

t[1].append(10)

print(t)

t[0][0].append(4)

print(t)

output:

([[1, 2, 3], [5, 6, 7]], [8, 9])

([[1, 2, 3], [5, 6, 7]], [8, 9, 10])

([[1, 2, 3, 4], [5, 6, 7]], [8, 9, 10])

Shared reference is, more than one variable pointing to the same object in the memory.

With mutable objects, python mem manager will not create shared memory references.

Everything in python is passed by reference.

Variable equality in two ways. Memory address and internal data.

To check address equality, use identity operator ‘**is’**. To check data equality, use ‘**==’**

a = 10

b = 10

c = None

d = None

print (a is b)

print(a == b)

print (c is d)

print(c == d)

output: True

True

True

True

**Interning:** Reusing objects on demand. At startup, CPython preloads(caches) a global list of integers into memory(-5 to 256)

**Singleton objects** are classes that are instantiated only once.

When the python code is compiled, identifiers are interned. Function, class, variable names etc. it should share with an \_,letter and can include only an \_.

Sys.intern(string)

import sys

a = sys.intern("hello")

b = sys.intern("hello")

c = a

a is b

b is c

output: true true

peephole optimization happens at compile time.

**Peephole optimization** is a type of Code **Optimization** performed on a small part of the code. It is performed on the very small set of instructions in a segment of code. The small set of instructions or small part of code on which **peephole optimization** is performed is known as **peephole** or window.

Membership tests: mutables are replaced by immutables.

In Python, the int objects use a variable number of bits according to the size of the integer number.

Sys.getsizof(<number>) to know how many bits are used to store the value.

The floor operation ‘//’ is not the same as truncation when deals with negative ‘-‘ numbers.

Math.floor

a = 13

b =3

print(a == b \* (a // b) + a % b )

output: True

the default base of an integer is 10 // changing a number from base 10 to another type. Bin(10), oct(25), hex(34)

Rational numbers are fractions of integers. Any real number with a finite number of digits after the decimal point is also a rational number. Square root of 2 and pi are irrational numbers.

From fractions import Fraction

format(0.23, '.12f')

in bankers rounding, selects ‘even’ least significant digit

decimal.getcontext() // decimal.localcontext()

#tuple passed in Decimal constructor. the first argument in the tuple is the sign

from decimal import Decimal

t = (0,(3,4,5,6,7), -4)

a = Decimal(t)

print(a)

output: 3.4567

Not all functions available in math module is not in Decimal class.

Div and mod operations are not allowed in complex numbers. Math functions will not work with complex numbers; instead use cmath.

Complex class is built-in, which uses a rectangular coordinates as a constructor. A = complex(1,2)

a = complex(3,5)

b = 2 + 4j

c = a + b

d = a \* b

print(c)

print(d)

output: (5+9j) // (-14+22j)

--

bool class is definded PEP285. Bool class is a sub class of integers.

a = issubclass(bool, int)

print(a)

output: True.

True and False are singleton objects of type bool.

All objects in Python has an associated truth value.

Classes define their truth values by defining a special instance method. \_\_bool\_\_.If its not defined, use \_\_len\_\_.

class A():

pass

# def \_\_bool\_\_(self):

# return self != 0

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

return self != 0

a = A()

print(bool(a))

--

Output: True

Boolean operators: not, and, or

Use ‘return’ only when inside a function

Chained comparison: a>b>c>d<e!=f

If a positional parameter is defined with a default value, every positional parameter after that also has to be given with a default value.

‘Named’ arguments are given when the function is called.

Once use a named argument, all arguments follows should also be named.

‘packed values’ are bundled together in some way. Any iterable is considered a packed value.

Dictionaries and sets are unordered types. They can be iterated, but no guarantee of order.

# iterates through all the items in the list

a = ['a','b','c',[10,20,30],[1,2,3]]

for i in a:

for x in i:

print(x)

in an assignment operation, Python evaluates the ‘right handside’ first. So swapping a,b = b,a works.

Slicing works only with iterable sequences.

a = [1,2,3,4,5,6,7,8,9]

#b,c = a[0],a[1:]

#b,c = a[0:3], a[4:]

b, c, \*d = a

b, c, d

output: (1, 2, [3, 4, 5, 6, 7, 8, 9])

In [ ]:

Always unpack into a list

To unpack the key and value of a dictionary, use double star. \*\* on the lhs

\*args return a tuple of positional arguments.

def func(a, b):

return a and b

# return 0 as 'false' becase a is 0. Else it will return 20 from the rhs - when its returning from an 'and'

func(0,20)

#unpacking iterables in to \*args

l = [10,20,30]

def func(a,b,\*args):

print(a)

print(b)

print(args[0],args[2])

func(1,2,\*l)

#unpacking iterables in to \*args

l = [10,20,30,40,50]

def func(a,b,\*args):

print(a)

print(b)

#\*args starts at l[2]

print(args[0],args[2])

#func(1,2,\*l)

func(\*l)

output: 10

20

30 50

Positional argument following \*args are called keyword arguments.

Bare ‘\*’ indicates end of positional arguments.

Positional arguments cannot follow named arguments

\*\* scoops up the keyword arguments. Stores arguments in a dictionary.

Generator expression use parenthesis. ()

Higher order functions are that takes a function as argument or returns a function.

First string in the body of a function is the docstring. Its stored in function.\_\_doc\_\_ property

Annotations are stored in function.\_\_annotations\_\_ property

Type hints are advanced version of annotation

def func(a: 'f name' ="Jaison", b:'l name' = "Jacob") -> 'full name':

"concatenates a name"

return a + ' '+ b

func()

print(func.\_\_doc\_\_)

print(func.\_\_annotations\_\_)

the lambda expression returns a function object

No annotations in lambda.

func = lambda x: x\*\*2 if x > 10 else x

a = func(5)

print(a)

output: 5

--

Sorted uses stable sort

lst = ['N','n','K','J','c','r']

srt = sorted(lst, key=lambda x: x.upper())

print(srt)

output: ['c', 'J', 'K', 'N', 'n', 'r'

---- (review1) how to unpack a dictionary in to variable? Or right handside unpacking. Use {\*\*dic}. a = {\*\*dic}

a = dict(a=10,b=20,c=30)

c = {\*a}

b = {\*\*a}

print(c)

print(a)

Functions are first class objects, that have properties, attributes and methods.

**Dir()**is a builtin function, given an object, will return all the attributes of that object.

def func(a=10,\*, kw=1):

c = 10

m = func;

m.new\_var = "Hello world"

print(dir(func))

print(m.\_\_name\_\_)

print(m.\_\_defaults\_\_)

print(func.\_\_code\_\_.co\_varnames)

Objects and classes have attributes bound to the object. An attribute that is callable is called a method.

Import inspect.ismethod, isfunction, isroutine, getsource, getmodule, getcomments, signature,

Getcomments(func) – will return the comments given just above the function definition.

Divmod(x,y,/) – here the ‘/’ says that the preceding parameters are positional only.

Callable(print) – to check if an object is callable. All callables return a value.

A higher order function is one that takes in a function as a parameter and returns a function. Sorted, maps (list comprehension), filter(generator expression)

Map(func,\*iterables) returns an iterator that calculates the function applied to each element of the iterables.

Filter(func, iterable) – if fuction returns true, it will retain the iterable element, if it false, will throw out the element. Returns an iterator, for which all the elements are returned ‘truthy’.

Zip(\*iterables) – takes more than one iterable and returns a tuple zipping up all the iterables

#map

l1 = [10,21,30]

l2 = [5,15,25]

it = map(lambda x, y: x + y, l1,l2)

print(list(it))

#filter

flt = filter(lambda x: x % 2 == 0, l1)

print(list(flt))

#zip

zp = zip(l1,l2)

print(list(zp))

output: [15, 36, 55]

[10, 30]

[(10, 5), (21, 15), (30, 25)]

List comprehension [ xpression1 for variable in iterable if expression2]

l1 = [10,20,30]

l2 = [2,4,6]

#combining filter with map

lst = filter(lambda x: x > 15, map(lambda n,m: n + m, l1,l2))

print(list(lst))

#list expression

a = list([x + y for x,y in zip(l1,l2) if x+y > 20])

#generator expression

gen = (x + y for x,y in zip(l1,l2) if x+y > 20)

print(a)

for i in gen:

print(i)

output: [24, 36]

[24, 36]

24

36

Reducing functions are functions that recombine an iterable recursively and end up return a single value. They are often called aggregators, accumulators or folding functions.

#find maximum of a number from a list

ls = 10,20,30,4,5,7,19,33,55,1

result = ls[0]

for r in ls:

if r > result:

result = r

print(result)

#reduce

from functools import reduce

result = reduce(lambda x,y : x + " " + y, ("Jaiosn", "Jacob", "Valayil"))

print(result)

output: Jaiosn Jacob Valayil

--

Min,max,any,sum,all are builtin reducing functions.

#partial

from functools import partial

def func\_add(a,b,c):

return a + b + c

#->

par = partial(func\_add, 10)

result = par(20,30)

print(result)

output:60

#partial function using function

def func1(a,b,\*args,d,\*\*kwargs):

print(a,b,args,d,kwargs)

def func2(\*args,s,m,\*\*kwargs):

return func1(10,\*args,d=s,\*\*kwargs)

func2(20,30,s=10,m=15,k=19)

output: 10 20 (30,) 10 {'k': 19}

operator module contains arithmetic operators, comparison, sequence/mapping – contains(s,val), concat, countOf(s,val), getitem, setitem,delitem. Itemgetter

#itemgetter sets an index for a sequence and returns a callable

from operator import itemgetter

lst = [1,23,3,4,5,6,7]

f = itemgetter(3,4,5)

print(f(lst))

--

#attrgetter sets an index for properties of an object and returns a callable

from operator import attrgetter

def func():

pass

#setting function attributes

func.a = 10

func.b= 20

res = attrgetter('a','b')(func) #callable is called with a ()

print(res)

#attrgetter calling the callable and another way using methodcaller

from operator import attrgetter, methodcaller

s = 'monte python'

# attrgetter('upper')(s)()

f = attrgetter('upper')

f(s)() # calling 'upper' callabe

x = methodcaller('upper')(s)

print(x)

methodcaller calls the named attribute and calls it aswell.

**Scopes: Global, local, non local, nested**

The portion of the code, where the name/binding is defined is called the lexical scope of the variable. These bindings are stored in namespaces.

Global scope is module scope, spans a single file.

Some built-in’s are available across the modules: True, False, None, dict, print

Built-in has a namespace. Each loaded module has its own namespace.

If a label couldn’t find in a namespace, python look above(upward) for it, not below it.

Masking is redefining a built-in or global variable inside the local scope.

Variables defined inside a function is local to that function. Every time the function is called, the scope is recreated.

There is no ‘code block’ variables in Python.

Inner function modifying the outer function variable has to redefine it as ‘nonlocal’ variable.

def out():

x = 10

print(f"outer {x}")

def out1():

#local x

x = 2

print(f"outer1 {x}")

def out2():

nonlocal x

x = 3

print(f"outer2 {x}")

out2()

out1()

out()

output: outer 10

outer1 2

outer2 3

nonlocal variables referenced are called ‘free’ variables.

Closure is a function plus the extended scope that contains the free variable.

def outer():

x = "Python"

def inner():

print(f"inner {x}")

return inner

fn = outer()

fn()

‘cell’ references the variable in outer scope

Fn.\_\_closure\_\_

Shared free variables can be accessed in the inner functions and can be modified.

def outer():

x = 10

def inner():

nonlocal x

x += 1

print(f"inner {x}")

return inner, inner1

f1, f2 = outer()

f1()

f2()

lambda expressions become closures only if they have a free variable.

--

A decorator takes a function as argument

Returns a closure

Closure takes any combination of parameters

perform some function in the inner code (closure)

Closure function calls the original function passing the arguments passed to the closure

Returns whatever is returned by that function call.

#decorator

from functools import wraps

def decor(fn):

print("decor")

@wraps(fn)

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

print("inner")

return fn(\*args,\*\*kwargs)

#inner = wraps(fn)(inner)

return inner

def add(a,b,\*,d):

"function add"

return a + b + d

add = decor(add)

res = add(10,20,d=5)

print(res)

help(add)

#stacked decorator:

def auth(fn):

def inner():

res = fn()

return res

return inner

def logged(fn):

def inner():

res = fn()

return res

return inner

@auth

@logged

Def add():

Pass

add = auth(logged(add))

maxsize of lru\_cache is 128

when arguments has to be passed to the parameterized decorator, define a decorator **factory** outer to the decorator and return the decorator.

def dec\_factory(num):

def dec(fn):

from functools import wraps

@wraps(fn)

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

print(f"number is {num}")

return fn(\*args,\*\*kwargs)

return inner

return dec

@dec\_factory(3)

def add(a,b,\*,c):

return (a + b) \* c

add = add(10,15,c=2)

print(add)

output: number is 3

50

Add = dec\_factory(3)(add)

--

Decorator class uses \_\_call\_\_ to define the inner function inside – same like decorator factory function

#Decorator class

class Decor\_Class:

def \_\_init\_\_(self,a,b):

self.a = a

self.b = b

def \_\_call\_\_(self,fn):

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

print(f"inner called, a {self.a} b {self.b}")

return fn(\*args, \*\*kwargs)

return inner

@Decor\_Class(10,20)

def hello(c):

print(c)

hello("Hello")

# obj = Decor\_Class(5,7)

# hello = obj(hello)

# hello("world")

Decorate a class (not decorator class) – class decorator

from fractions import Fraction

def dec\_speak(cls):

cls.speak = lambda self, msg: f"class is {self.\_\_class\_\_.\_\_name\_\_}, msg is {msg}"

return cls

decim = dec\_speak(Fraction)

f = decim(100,2)

f.speak("All the very best")

#applying the decorator function on another class

class Person:

pass

p = dec\_speak(Person)

p1 =p()

p1.speak("Hi there")

@total\_ordering is applied to the class// functools singledispatch

**Tuples are read only lists.**

In named tuples, in addition to the value of a position, we give names as well.

Tuples lists strings

Containers containers containers

Order matters order matters order matters

Hetrogeneous/homogeneous hetro/homo(more homo) homo

Indexable indexable indexable

Iterable iterable iterable

**Immutable mutable immutable**

**Fixed length length can change fixed length**

**Fixed order order of lements can fixed order**

**Change – can do in-place**

**Sorts, reversals**

Record = (djia, 2019, 03, 24, 1000,9000,9500,300,100,150)

Symbol, year, month, day, high, low, avg, **\*\_**, d\_avg = record (pythonistic)

#A list of tuple

N\_Y = "US","NA",1000000

TOKYO = "JAPAN","ASIA",1200000

JOHANNESBURG = "SA","AFRICA",800000

SANJOSE = "COSTA RICA", "LA", 500000

cities = [N\_Y,TOKYO,JOHANNESBURG,SANJOSE]

population = sum(city[2] for city in cities)

#population

total = 0

for city in cities:

total+=city[2]

total

review2 class decorator\*

def class\_decor(cls):

cls.speak = lambda x: x

return cls

class Person:

pass

p = class\_decor(Person)

s = p.speak("Hello world")

print(s)

#output Hello world

from collections import namedtuple

namedtuple is a function, which is a class factory, creates a class which inherits from tuple. Also provides named properties to access the elements of the tuple.

# named tuple

from collections import namedtuple

Point2D = namedtuple('Point2D', 'x y')

pt = Point2D(10,20)

print(pt.\_asdict)

print(pt.\_fields)

OrderedDict = Dictionaries that guarantee the ‘key order’.

List.extend(values) – used to append more than a value into the list.

from collections import namedtuple

Person = namedtuple('Person', 'fname lname age')

p = Person("Jaison", "Jacob", 45)

print(p)

#update a value

p = p.\_replace(fname="Esther", lname="Sarah")

print(p.age)

print(hex(id(p)))

#modify a named tuple

values = p[:2]

#a = Person(\*values, 46)

a = Person.\_make(values + (100,))

print(a)

#create a new class from existing class by adding another field (extend a named tuple)

PersonExt = namedtuple('PersonExt', Person.\_fields + ('state',))

print(PersonExt.\_fields)

\_\_defaults\_\_ of function allows to provide default values for a function.

Class\_\_new\_\_.\_\_defaults\_\_(defaults) is right aligned.

Class.\_replace(defaults) is prototype.

--

from collections import namedtuple

Person = namedtuple('Person', 'fname lname age')

p = Person("Jaison", "Jacob", 45)

#set a docstring

Person.\_\_doc\_\_ = "Person class"

Person.fname.\_\_doc\_\_ = "First Name"

print(Person.fname.\_\_doc\_\_)

#defaults prototype

proto1 = Person(fname="FirstName", lname = "LastName", age=10)

pr1 = proto1.\_replace(age=15)

print(pr1)

#setting defaults through constructor

Person.\_\_new\_\_.\_\_defaults\_\_ = ("Jacob",23)

p1 = Person("Emil")

print(p1)

--

from random import randint, random

from collections import namedtuple

Color = namedtuple('Color', 'red green blue orange')

def the\_color():

red = randint(0,255)

green = randint(0,255)

blue = randint(0,255)

orange = round(random(), 2)

return Color(red, green, blue, orange)

color = the\_color()

print(color)

dictionary can be used to create fields in namedtuples.

#Alternative to dictionaries

from collections import namedtuple

data\_list = [{'key1':100, 'key2':200, 'key3' : 300}, {'key3' : 10, 'key1' : 20, 'key2' : 15}, {'key2' : 11, 'key3':23}]

print(data\_list)

def tuplify\_dict(dict\_):

keys = {key for dic in data\_list for key in dic.keys()}

Struct = namedtuple('Struct', sorted(keys))

Struct.\_\_new\_\_.\_\_defaults\_\_ = (None, ) \* len(Struct.\_fields)

return [Struct(\*\*dick) for dick in dict\_]

d\_tpl = tuplify\_dict(data\_list)

print(d\_tpl)

--

Modules are objects of type ModuleType

Globals() – gives all the labels defined in the global / module namespace

Namespaces are dictionaries.

Locals() – provides all labels inside local namespaces (eg – function)

def func(a,b,c):

d = a + b + c

print(locals())

return d

Builtin modules in python are written in ‘c’. Standard library most of it is written in ‘python’ and a few in ‘c’.

When a module is loaded, its reference is stored in a cache in sys.modules and in globals.

Module.\_\_dict\_\_ will contain all the attributes of the module.

Module.\_\_spec\_\_ gives metadata about the module.

Dir(module) gives ‘keys’ from the module.

Namespaces are dictionaries.

Where is python installed? Look at sys.prefix

Where are compiled c binaries located? Look at sys.exec\_prefix

Where does python look for imports? Sys.path

Import importlb

Importlib.import\_module(modulename) - will import the module.

‘pyc’ is a compiled python file.

Finders, loaders, finders + loaders = importer

Loaders can be found at sys.meta\_path

---

import importlib

importlib.util.find\_spec("fractions")

dir(importlib)

#importlib.import\_module("fractions")

import fractions, sys

fractions.\_\_spec\_\_

sys.meta\_path

---

import sys

import importlib

import fractions

sys.prefix

sys.exec\_prefix

sys.path

sys.meta\_path

importlib.util.find\_spec("decimal")

fractions.\_\_spec\_\_

when importing a module, first it adds to sys.modules dictionary. Then adds to module.globals(). If a function is imported from a module, module is loaded in to sys.modules and function is entered in module.global().

--

from cmath import exp

import sys

print('cmath' in sys.modules)

print('cmath' in globals())

print('exp' in globals())

--

Importlib.reload(modulename)

Add files to a zip file: python –m zipfile –c my\_app \_\_main\_\_.py timeit.py

List the zip file contents: python –m zipfile –l my\_app

Frozenimporter is a finder + loader of self contained application. It includes files, libraries, runtime env all required to run the application.

Packages are modules that can contain modules. Package.\_\_path\_\_

\_\_file\_\_ for the module points to the code for the module.

\_\_file\_\_ for the package points to the \_\_init\_\_.py file

No path for the modules.

Package has \_\_package\_\_, \_\_file\_\_, \_\_path\_\_

File has only \_\_package\_\_ and \_\_file\_\_. No \_\_path\_\_

Arranging classes and function – developers perspective and users perspective.

Use \_\_init\_\_.py to import classes and functions in users perspective for easy access.

Packages has the \_\_dict\_\_ property. It’s the namespace

If a function label has an underscore \_ in front, it will not get exported if imported through ‘import \*’

To specify export, use bare **\_\_all\_\_**= [‘symbol’, ‘symbol’,] in the module. Only these symbols will be exported from the module. Every package has a package.\_\_all\_\_ property.

Copy the globals in for loop: for k in dict(globals()).keys(): print(k)

Implicit namespace package doesn’t contain \_\_init\_\_.py. they are package ‘like’.

Type of namespace package is ‘module’.

Single namespace package can live in multiple directories

Cannot monkey around the path of the namespace package in import

To import from package.zip, just append the package to os.path.append(‘package.zip’)

Pypi has 129k packages as of now.

PEP8 – style guide // PEP 20 – zen of python // pep 484 – type hints

Books: Learning python by Mark Luts pub: O’reilly // Fluent python – Luciano Ramalho – O’reilly \*\*high rating

Python cookbook – David Beazley & Brian K Jones – O’reilly

Effective Python: 59 specific ways to write better Python – Brett Slatkin

Python in a nutshell: Alex Martelli, Anna – O’reilly

Twitter: @raymonh – Raymond Hettinger co developer of Python and implemented the latest dictionary.

#zip() with \*argument is great for transposing 2d-data

m = [(1,2,3),(4,5,6)]

print(list(zip(\*m)))

output: [(1, 4), (2, 5), (3, 6)]

Yoututbe: PyCon videos, GvR, RaymondHettinger, AlexMartelli

Blog: <http://planetpython.org>

<https://Stackoverflow.com>

python3.6 new items: Dictionaries maintain sort/key order, preserving order of \*\*kwargs in function arguments, underscores in numeric literals, fstrings, type annotations, asynchronous enhancements.

a = 1000000232

b = 2344

c = f"numerator {a} / denominator {b} = {a/b:0.7f}"

print(c)

ide: mypy, pycharm.

From 3.6, ordereddict is not required.

#update/merge dictionaries preserve order and remove duplicates from the updated dictionary

d = {"a":10,"b":20,"c":30}

e = {"d":40, "e":50, "a":100}

d.update(e)

print(d)

output: {'a': 100, 'b': 20, 'c': 30, 'd': 40, 'e': 50}

ordereddict methods: move\_to\_end(key, last=true), popitem(last=true), reversed()

#move\_to\_end

d = {"a":10,"b":20,"c":30,"d":40,"e":50,"f":60}

for i in range(len(d)-1):

key = next(iter(d.keys()))

d[key] = d.pop(key)

print(d)

#popfirst

key = next(iter(d.keys()))

d.pop(key)

print(d)

#poplast

d.popitem()

print(d)

output:

{'f': 60, 'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50}

{'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50}

{'a': 10, 'b': 20, 'c': 30, 'd': 40}

dictionary implementation: <http://code.activestate.com/recipes/578375/>

#namedtuple factory method

from collections import namedtuple

def named\_dict\_factory(class\_name, \*\*field\_names):

Struct = namedtuple("Struct", field\_names.keys())

Struct.\_\_new\_\_.\_\_defaults\_\_ = tuple(field\_names.values())

return Struct

Person = named\_dict\_factory("Person", name="Jaison",state="Kerala",country="India",age=45)

p = Person("Jacob")

print(p)

Struct(name='Jacob', state='Kerala', country='India', age=45)

In [ ]:

#format number with underscore

a = "format {:\_}".format(1000000)

print(a)

output: format 1\_000\_000

#lambda expression inside print statement.

a = 10

print(f"{(lambda x: x \*\* 2)(a) if a >5 else a}")

# print(f"{(lambda x: x \*\* 2 if a >5 else a)(a)}")

by default, system time is set for the random seed.

Random.seed(0), random.randint(10,20), random.random(), random.shuffle(“abcd”), random.randrange(5), random.choice(sequence), random.choices(), random.sample()

randoms = []

l = [10,20,30,40]

import random

randoms = [random.choice(l) for \_ in range(10)]

print(randoms)

from timeit import timeit (timeit has its own name space) timeit(stmt=’..’, globals=globals(), number=10)

sys.argv for commandline arguments

--

import argparse

parser = argparse.ArgumentParser(description="Add two numbers")

parser.add\_argument("a", help="Enter a number", type=int)

parser.add\_argument("b", help="Enter a number", type=int)

p = parser.parse\_args()

a = p.a

b = p.b

print(f"a + b = {a + b} ")

--

#like switch statement

def dow\_switch(dow):

dow\_dict = {

1: lambda: print("Sunday"),

2: lambda: print("Monday"),

3: lambda: print("Tuesday"),

4: lambda: print("Wednesday"),

5: lambda: print("Thursday"),

6: lambda: print("Friday"),

7: lambda: print("Saturday"),

'default': lambda: print("Incorrect day entered")

}

return dow\_dict.get(dow, dow\_dict['default'])()

dow\_switch(55)

1st Apr ’20 – Iteration, Generators

Protocols: sequence protocol, iterator protocol, iterable protocol, generator protocol

<https://github.com/fbaptiste/python-deepdive>

a sequence of ordered elements. Position is important. Index start at 0

mutable sequence types: lists, byte arrays

immutable sequence types: tuple, strings, frozen sets, range, bytes. Tuples are also data structures.

Other standard types from standard library: namedtuple, deque, array module (array object)

Review3

Any sequence type is iterable. A ‘set’ is an iterable, but not a sequence. Set is not ordered.

Common sequence methods: a in s, a not in s, a + s (its concatenation), s \* n (repetition – n is an integer), len(s), min(s), max(s), s.index(x), s.index(x, i), s.index(x, I, j)

S[i], s[i:j], s[i:j:k] – k is the step. Slicing will return the same container type.

Mutable sequence types does NOT support hashing. Hash(s)

#when a mutable sequence is concatenated (applies to repetition also), its memory address is referenced in the copied object

a = [[10,20]]

b = a + a

print(b)

print(hex(id(b[0][0])))

print(hex(id(b[1][0])))

b[0][0] = 100

print(b)

a = {1:10,2:"a",3:"b","c":"c"}

'c' in a

Output: True

Cannot mix types in concatenation.

--

a = ":::".join(["a","1","@","b"]) // output: 'a:::1:::@:::b'

enumerate a sequence / s= list(enumerate(s))

s = list(enumerate((10,20,30))) output: [(0, 10), (1, 20), (2, 30)]

--

# slice from end.

s = "python"

s[5:2:-1] // output : noh

---

Slicing always creates a new object

#repetition problem of a new object in mutable object. Shallow copy and deep copy can get rid of this problem

a = [[10,20]]

b = a \* 2

a[0][0] = 100

print(b) // output: [[100, 20], [100, 20]]

concatenation does not mutate objects. It just creates another object for the variable to point to.

Mutating an object means changing the objects state without changing the object itself.

#mutating a list element

a = ["Jaiosn","Jacob"]

print(hex(id(a)))

print(hex(id(a[0])))

a[0] = 100

a.append("Valayil")

print("---------\n",hex(id(a)))

print(hex(id(a[0])))

output:

0x165de9d3048

0x165deb356b0

---------

0x165de9d3048

0x7ffea80dadf0

Mutable sequence mthods:

s.clear() // s.append(x), s.insert(i,x), extend(iterable), pop(i), remove(x), s.reverse() – inplace reversal of the elements, s.copy() – shallow copy

a=[2,3,4,5,6]

a[1:3] = {'a','b','c','d','e','f'}

print(a)

output: [2, 'c', 'a', 'd', 'f', 'b', 'e', 5, 6]

--

Constant folding is the process of recognizing and evaluating constant expression at compile time rather than computing them at runtime.

#disassemble

from dis import dis

dis(compile("(1,2,3,4,'a')", 'string', 'eval'))

compiling a list takes more effort than a tuple, because tuples are immutable

for any sequence s, the index range is given by: 0 based: 0 <= n < len(s)

shallow copy uses the same memory references in the new list from the copied list.

Deep copies require a recursive approach. Deepcopy is applicable to all objects in general.

Standard library ‘copy’ module provides generic copy and deep copy operations

Pythonic way of copying an iterable is to use list comprehension

lst = list([1,2,3,4])

cp =[e for e in lst]

print(cp)

--

Slicing only works with sequences. Slicing rely on indexing. With mutable sequence types, can assign data.

S = slice(0,2) object

lst = list([1,2,3,4,5,6,7,8,9])

s = slice(0,4)

a = lst[s]

print(a)

step value of -1 will begin from end

slice.indices()

lst = list([1,2,3,4,5,6,7,8,9])

s = slice(0,3).indices(9)

lst[-3:-9:-1] # k of step -1 iterate back. j of -9 is the 8th element from the end. i of -3 is the 3rd element from end.

Output: [7, 6, 5, 4, 3, 2]

--

Custom sequence:

An immutable sequence type should support: return the length of the sequence. Given an index, return the element of that index. Pythons looping mechanisms(for loop, comprehensions) should be able to iterate through the elements.

Sequences should implement the protocols \_\_len\_\_ and \_\_getitem\_\_.

\_\_getitem\_\_ should raise an IndexError when the index is out of bounds.

Pythons list object implements \_\_len\_\_, \_\_getitem\_\_. Mylist.\_\_getitem\_(index)

---

my\_list = [1,2,3,4,5,6,7,8,9,100,12,13]

index = 0

while True:

try:

idv = my\_list.\_\_getitem\_\_(index)

except IndexError:

break

index += 1

print(idv)

custom sequence:

class Silly:

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

self.n = n

def \_\_getitem\_\_(self,n):

if n > self.n or n < 0:

raise IndexError

else:

return f"This is a silly return"

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

return self.n

silly = Silly(5)

for item in silly:

print(item)

when the class object is created, the \_\_getitem\_\_ is called, returns a sequence.

+= works true for the immutable objects. It will change the address

a = 1000

print(hex(id(a)))

a += 100

print(hex(id(a)))

b = [10,20,30]

print(hex(id(b)))

b += b

print(hex(id(b)))

output:

0x1f7be3fc830

0x1f7be6263b0

0x1f7be4251c8

0x1f7be4251c8

List += tuple can be done

#mutable object assignment through slicing

a = [1,2,3,4,5,6,7,8,9]

a[slice(2,7,1)] = 10,11,12,13,14,15

print(a)

#output: [1, 2, 10, 11, 12, 13, 14, 15, 8, 9]

a[2:7:2] = 10, 11,12 #the mutating element and assignment element should be equal in number

print(a)

#output: [1, 2, 10, 4, 11, 6, 12, 8, 9]

--

Deleting a slice means assigning empty elements. A[0:2] = “” or []

#insertion using slice

a = [10,20,30,40,50]

a[1:1] = 'abc'

#output: [10, 'a', 'b', 'c', 20, 30, 40, 50]

print(a)

--

Assignment operation requires an iterable object on the rhs

Length of the extended slice and assignment iterable must be of same size.

In normal concatenation of objects using +, both the objects should be of same type. Inplace concatenation using +=, 2nd object can be any other iterable.

To assign elements in a custom sequence, use \_\_setitem\_\_

Protocol \_\_contains\_\_ matches ‘in’, \_\_delitem\_\_ for ‘del’

\_\_rmul\_\_ has to be implemented in custom sequence class inorder to perform.

#numbers modules provides various number classes

import numbers

def check\_num(num):

if isinstance(num, numbers.Number):

print(num)

check\_num(10.998)

def check\_real(num):

if isinstance(num, numbers.Real):

print(num)

check\_real(10.998)

--

Sorted(iterable, key=None, reverse=False). Sorted function makes a copy of the iterable and returns the sorted elements in a list. Algorithm used: TimSort of Tim Peters.

A stable sort is one that maintains the relative order of items that have equal keys.

List.sort(iterable,key,reverse) – makes an in-place sorting.

d = {"c":100,"d":150,"f":400,"a":75,"b":25}

k = sorted(d, key=lambda x: d[x]) # sorting based on the value of the dictionary

print(k)

output: ['b', 'a', 'c', 'd', 'f']

--

In-place sorting:

lst = [0,10,5,2,3,4,1,3,9]

lst.sort()

print(lst)

output: [0, 1, 2, 3, 3, 4, 5, 9, 10]

st = ["a","bird","parrot","ginger","zen of python","hehehe"]

n = sorted(st, key=lambda x: x[-1])

print(n)

output: ['a', 'bird', 'hehehe', 'zen of python', 'ginger', 'parrot']

--

#disasseble

st = ["a","bird","parrot","ginger","zen of python","hehehe"]

n = sorted(st, key=lambda x: x[-1])

from dis import dis

a = compile('sorted(st, key=lambda x: x[-1])',"string", "eval")

dis(a)

--

List comprehension is generating a list by transforming, optionally filtering another iterable.

New\_lst = [transformation iteration filtering]

Comprehensions have their own local scope. Comprehensions are basically functions.

Comprehensions can be nested within one.

n\_st = [[i \* j for j in range(5)] for i in range(5)] # [[outer] inner]

print(n\_st)

output: [[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8], [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]

--

#comprehension

l = [[(i,j,k) for i in range (3)] for j in range(2) for k in range(1)] # [[[far outer]outer]inner]

print("comprehension ",l)

#for loop

lst = []

for k in range(1):

for j in range(2):

for i in range(3):

lst.append((i,j,k))

print("function ", lst)

output: comprehension [[(0, 0, 0), (1, 0, 0), (2, 0, 0)], [(0, 1, 0), (1, 1, 0), (2, 1, 0)]]

function [(0, 0, 0), (1, 0, 0), (2, 0, 0), (0, 1, 0), (1, 1, 0), (2, 1, 0)]

#returning function objects

a = [lambda x, p=i: x \* p for i in range(5)]

print(a) # creates a list of 5 functions.

print(a[2](10)) #access the list index of functions by value

output:

[<function <listcomp>.<lambda> at 0x00000211B8154168>, <function <listcomp>.<lambda> at 0x00000211B8154D38>, <function <listcomp>.<lambda> at 0x00000211B81540D8>, <function <listcomp>.<lambda> at 0x00000211B8154E58>, <function <listcomp>.<lambda> at 0x00000211B81543A8>]

20

List comprehension, generator expression, set comprehension, dictionary comprehension

ITERABLES AND ITERATORS:

Iterator protocol. All itereable objects implements \_\_iter\_\_ protocol, which makes it an iterable object.

Sets are unordered collection of items. Sets are not indexable.

#iterating a collection

class MyClass:

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

self.i = 0

self.length = length

def \_\_next\_\_(self):

self.i += 1

if self.i >= self.length:

raise StopIteration

return self.i

seq = MyClass(10)

#for i in range(15):

while True:

try:

next(seq)

print(seq.i, end=" ---")

except StopIteration:

break

output: 1 ---2 ---3 ---4 ---5 ---6 ---7 ---8 ---9

**iterator protocol has two methods: \_\_iter\_\_ (**returns the object itself), **\_\_next\_\_ (**returns next element from the collection)

if an object is an iterator, we can use it with for loops, comprehension etc, but it exhausts after single iteration.

#implementing the iterator protocol (\_\_iter\_\_ and \_\_next\_\_) to make the object an iterator.

class MyClass:

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

self.i = 0

self.length = length

def \_\_next\_\_(self):

self.i += 1

if self.i >= self.length:

raise StopIteration

return self.i

def \_\_iter\_\_(self):

return self

seq = MyClass(10)

#uses list comprehension

/\* it = [item for item in seq]

print(it) \*/

for item in seq:

print(item, end="--")

output: 1--2--3--4--5--6--7--8--9--

**the iterable objects implements only the \_\_iter\_\_(self) protocol, which return the iterator object.**

**The iterator object implements both \_\_iter\_\_ and \_\_next\_\_ protocols.**

class Cities:

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

self.\_cities = "mumbai","kolkota","chennai","Ahmedabad","bengaluru","indore"

self.\_index = 0

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

return len(self.\_cities)

# def \_\_iter\_\_(self):

# print("Cities iter called")

# return City\_Iterator(self) #----------- returns the iterator class object

def \_\_getitem\_\_(self,s):

return self.\_cities[s]

class City\_Iterator:

def \_\_init\_\_(self,city\_obj):

self.\_city\_obj = city\_obj

self.\_index = 0

#def \_\_iter\_\_(self):

# print("City iterator iterator called")

# return self

def \_\_next\_\_(self):

if self.\_index >= len(self.\_city\_obj):

raise StopIteration

print("next called")

pick = self.\_city\_obj.\_cities[self.\_index]

self.\_index += 1

return pick

cities = Cities()

for city in cities:

print(city)

print(list(cities))

print("sorted ",sorted(list(cities)))

output:

mumbai

kolkota

chennai

Ahmedabad

bengaluru

indore

['mumbai', 'kolkota', 'chennai', 'Ahmedabad', 'bengaluru', 'indore']

sorted ['Ahmedabad', 'bengaluru', 'chennai', 'indore', 'kolkota', 'mumbai']

first, python will check if the \_\_iter\_\_ is implemented in the iterable object. If not, it will check if the sequence protocol \_\_getitem\_\_ is implemented. If yes, it will return.

Itertools.cycle

Whenever an iterabale is called, it returns a new iterator. When we call on an iterator, it return itself, so get exhausted.

--

In computer programming, a **sentinel value** [..] is a special **value** whose presence guarantees termination of a loop that processes structured (especially sequential) data. The **sentinel value** makes it possible to detect the end of the data when no other means to do so (such as an explicit size indication) is provided

--

#delegating an iterator:

from collections import namedtuple

Person = namedtuple('Person', "fname, lname")

class Iter\_Delegator:

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

try:

self.\_persons = [person.fname.capitalize() + " " + person.lname.capitalize() for person in persons]

except (TypeError, AttributeError):

print("Error!")

self.\_persons = []

def \_\_iter\_\_(self):

return iter(self.\_persons) #delegation

p = [Person("Jaison", "Jacob"), Person("Sonia", "Suresh"), Person("Esther", "Sarah"), Person("Emil", "Jacob")]

d = Iter\_Delegator(p)

for name in d:

print(name)

output:

Jaison Jacob

Sonia Suresh

Esther Sarah

Emil Jacob

#reversing an iterator:

from collections import namedtuple

Person = namedtuple('Person', "fname, lname")

class Iter\_Delegator:

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

try:

self.\_persons = [person.fname.capitalize() + " " + person.lname.capitalize() for person in persons]

except (TypeError, AttributeError):

print("Error!")

self.\_persons = []

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

return len(self.\_persons)

def \_\_getitem\_\_(self,n):

return self.\_persons[n]

""" def \_\_reversed\_\_(self): # if reversed is not implemented, it will check for \_\_getitem\_\_ for sequences

return self.\_persons

def \_\_iter\_\_(self):

return iter(self.\_persons) #delegation

"""

p = [Person("Jaison", "Jacob"), Person("Sonia", "Suresh"), Person("Esther", "Sarah"), Person("Emil", "Jacob")]

d = Iter\_Delegator(p)

for name in reversed(d):

print(name)

output:

Emil Jacob

Esther Sarah

Sonia Suresh

Jaison Jacob

Generators are a type of iterators. Generator functions are factories, they create generators when called.

Generator expressions are like comprehensions syntax, a concise way of creating generators.

‘yield’ does three things. Emit, pause and resume.

Once the function ‘yields’, it suspends function. Calling ‘next’ will resume the function from the preserved state after the ‘yield’ statement . Once the function returns, it raises a StopIteration exception.

def song():

print("Line 1")

print("Line 2")

yield "'yielded' I am a lumberjack ok1"

print("Line after yield1")

yield("'yielded' I am a natural man")

print(" Line after yield 2")

line = song() # calling the function returns a ‘generator’ object. Here it is ‘line’. The function ‘song’ is a #generator factory.

yld = next(line)

print(yld)

yld = next(line)

print(yld)

output:

Line 1

Line 2

'yielded' I am a lumberjack ok1

Line after yield1

'yielded' I am a natural man

**A function that uses ‘yield’ statement is a generator function**

generators implement the ‘\_\_iter\_\_’ protocol, so that ‘next’ can be called upon it.

Generators are lazy iterators. It can be passed to for loop, comprehension. It exhausts when encounter the function return.

# factorial of a number using generator function

import math

def func(num):

for n in range(num):

yield math.factorial(n)

gen = func(5)

for i in gen:

print(i)

--

The **factorial of a number** is the product of all the integers from 1 to that **number**. For example, the **factorial** of 6 (denoted as 6!) is 1\*2\*3\*4\*5\*6 = 720. **Factorial** is not defined for negative **numbers** and the **factorial** of zero is one, 0! = 1

#creating an iterable from a generator

def gen\_factory(n):

for i in range(n):

yield i

class Iter\_Gen:

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

self.n = n

def \_\_iter\_\_(self):

return gen\_factory(self.n) # returns the iterator making iterable

it = Iter\_Gen(10)

for item in it:

print(item)

next(iter(it)) # fixing an interator on the iterable

#creating an iterable from a generator / static method

class Iter\_Gen:

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

self.n = n

def \_\_iter\_\_(self):

return Iter\_Gen.gen\_factory(self.n) # returns the iterator making iterable

@staticmethod

def gen\_factory(n):

for i in range(n):

yield i

it = Iter\_Gen(10)

for item in it:

print(item)

next(iter(it)) # fixing an interator on the iterable

list comprehension is ‘iterable’ and are eager. Generator expression is ‘iterator’ and is lazy

#difference between nested loops and nested list comprehension

a = "a","b",'c',"d","e"

b = "x","y",'z'

x = [n + m for n in a for m in b] # [outer loop, inner loop]

print(x)

x = [[n + m for n in a] for m in b] #[[inner comprehension]outer comprehension]

print(x)

ouput:

['ax', 'ay', 'az', 'bx', 'by', 'bz', 'cx', 'cy', 'cz', 'dx', 'dy', 'dz', 'ex', 'ey', 'ez']

[['ax', 'bx', 'cx', 'dx', 'ex'], ['ay', 'by', 'cy', 'dy', 'ey'], ['az', 'bz', 'cz', 'dz', 'ez']]

#yield from

def matrix(n):

gen = ((i \* j for j in range(1, n+1)) for i in range(1, n+1))

return gen

def matrix\_gen(n):

for row in matrix(n):

# for item in row:

# yield item

yield from row # 'yield from' directly return the value

for item in matrix\_gen(3):

print(item, end=”,”)

output: 1,2,3,2,4,6,3,6,9,

iterate module tools:

slicing: islice

selecting and filtering: dropwhile, takewhile, compress, filterfalse

chaining and teeing: chain, tee

mapping and reducing: starmap, accumulate

infinite iterators: count, cycle, repeat

zipping: zip\_longest

combinatorics: product, perumtations, combinations, combinations\_with\_replacement.

Built-in functions: iter, reversed, next, len, slice, zip, filter, sorted, enumerate, min, max, sum, all, any, map

From functools: reduce

--

Aggregators are functions that iterate through every element of the iterable and returns a single element.

A function that takes a single argument and returns true or false is called a **predicate**.

Slice general iterables and iterators using islice. Islice(iterable, start, stop, step). Islice returns a lazy iterator.

#islice

l = 1,2,3,4,5,6,7,8,9,0

from itertools import islice

a = islice(l, 1,5)

list(a)

output: [2, 3, 4, 5]

--

Filterfalse retains elements where the predicate evaluates to false.

#filterfalse

l = 1,2,False,3,4,5,0,6,7,8,9,0

from itertools import filterfalse

#a = filterfalse(lambda x: x < 5, l)

a = filterfalse(None,l)

list(a)

output: [False, 0, 0]

--

Compress(data, selector) – returns data only where the corresponding selector element is True.

from itertools import compress

a\_data = "Jaison", "Jacob",45,"Valayil"

a\_selector = 1,2,0,False

b = compress(a\_data, a\_selector)

list(b)

output: ['Jaison', 'Jacob']

takewhile(predicate, iterable) – returns values as long as the elements are True.

from itertools import takewhile

a = 1,2,3,4,5,6

b = takewhile(lambda x: x < 5, a)

list(b)

output: [1, 2, 3, 4]

--

Dropwhile(predicate, iterable) – returns remaining elements after the predicate returns false

from itertools import dropwhile

a = 1,2,9,3,4,5,6

b = dropwhile(lambda x: x < 5, a)

list(b)

output: [9, 3, 4, 5, 6]

--

Review:

Count(start, step) – has not ‘stop’ – takes any form of numbers

from itertools import takewhile, count

a = list(takewhile(lambda x: x < 20, count(10, 1)))

print(a)

output: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

cycle loops over a finite iterable/iterator indefinitely.

from itertools import islice, cycle

def color():

yield "red"

yield "green"

yield "blue"

yield "orange"

yield "yellow"

col = color()

g = cycle(col)

a = list(islice(g,10))

print(a)

output: ['red', 'green', 'blue', 'orange', 'yellow', 'red', 'green', 'blue', 'orange', 'yellow']

Repeat yields the value repeatedly.

from itertools import islice, repeat

a = "Hello"

b = repeat(a)

c = list(islice(b,5))

print(c)

output: ['Hello', 'Hello', 'Hello', 'Hello', 'Hello']

Chain

#chain iterates through each iterator/iterable in the arguments

from itertools import chain

a = 1,2,3,4,5

b = 10,11,12,13,14,15,16

c = {'a','b','k'}

for item in chain(a,b,c):

print(item, end=” “)

output: 1 2 3 4 5 10 11 12 13 14 15 16 a b k

unpacking of an iterable is ‘eager’.

Tee(iterable, number) – is copying of iterables.it return a tuple of lazy iterators, NOT iterables.

#tee

from itertools import tee

a = [1,2,3,4]

b = tee(a,3)

for it in b:

for item in it:

print(item, end=" ")

output: 1 2 3 4 1 2 3 4 1 2 3 4

--

#from\_iterable - will unpack the iterable

from itertools import chain

def func():

yield (i\*\*2 for i in range(0,4))

yield (i\*\*2 for i in range(4,7))

yield (i\*\*2 for i in range(7,9))

k = chain.from\_iterable(func())

for i in k:

print(i, end=" ")

output: 0 1 4 9 16 25 36 49 64

Mapping is applying a callable to each element of an iterable.

Starmap is similar to map, where it unpacks every sub-element of the iterable and passes that to the map function. Starmap returns an iterator.

Reduce is (fn, iterable) where as accumulate is (iterable,fn) – fn is optional. Default is add.

Accumulate will return the iterator and intermediate results before returning the final result.

from itertools import starmap

def add(x,y):

return x + y

k = starmap(add, [(0,0),[1,2]])

print(list(k))

output: [0,3]

--

a = 1,2,3,4,5

a = iter(a) # set an iterator on an iterable to iterate

print(next(a))

print(next(a))

--

Looping with iterators

The iterator protocol is used by for loops, tuple unpacking, and all built-in functions that work on generic iterables. Using the iterator protocol (either manually or automatically) is the only universal way to loop over any iterable in Python.

--

Zip zipup according to the shortest length of the iterable

Zip\_longest fillup to the lontest iterable

#zip

a = 1,2,3,4,5

b = 6,7

c = "a","b","c","d"

x = zip(a,b,c)

print(list(x))

#zip\_longest

from itertools import zip\_longest

x = zip\_longest(a,b,c, fillvalue="#")

print(list(x))

output:

[(1, 6, 'a'), (2, 7, 'b')]

[(1, 6, 'a'), (2, 7, 'b'), (3, '#', 'c'), (4, '#', 'd'), (5, '#', '#')]

--

#groupby - next on group and next of sub iterator

from itertools import groupby

a = (("a",1,2,3),("a",1,2,3),("c","e","f","m",10),("c",1,3,5),(1,"k",2,"f",3),(1,10,20),(1,40,60,80))

b = groupby(a, key=lambda x: x[0])

for key, sub in b:

print(key, list(sub))

output:

a [('a', 1, 2, 3), ('a', 1, 2, 3)]

c [('c', 'e', 'f', 'm', 10), ('c', 1, 3, 5)]

1 [(1, 'k', 2, 'f', 3), (1, 10, 20), (1, 40, 60, 80)]

In [ ]:

Combintorics are mainly used in statistics

Product(l1,l2) – does all the possible combinations

Permutations(lst) // combinations([1,2,3],2) // combinations\_with\_replacement([1,2,3,4,5,6],4)

A context in python is a state surrounding a section of the code. review

‘open’ statement creates a context. ‘with’ enters the context. When exited from the context, the context manager will close the file.

Context is useful for open/close a file, start a db transaction, commit / abort the transaction, set a decimal precision and then revert back.

Context managers are explained in PEP 343.

Context management protocols: \_\_enter\_\_, \_\_exit\_\_

Patterns: open – close, enter – exit, change – reset, lock – release, start – stop

With MyClass() as obj – instance of MyClass is a handle to Python. ‘with’ enters the context created by the \_\_enter\_\_. ‘obj’ is the object returned by \_\_enter\_\_.

Once the ‘with’ block is exited or some exception happens within the bloc, \_\_exit\_\_ is called.

\_\_exit\_\_ runs even if an exception happens within the ‘with’ block.

\_\_exit\_\_(type, value, traceback)

If the ‘return’ value of \_\_exit\_\_ is set True, then silence the exception. If set to ‘False’, then propagate the exception.

‘with’ does not have its own scope. Its scope is wherever its defined.

‘yield from’ a context will not return from the context like the ‘return’ from a function.

#Generator function with context manager class

def open\_file(file\_name, mode):

file\_handle = open(file\_name, mode)

print("Opening file and returning file handle to the context manager")

try:

yield file\_handle

finally:

file\_handle.close()

pass

class Context\_Mgr:

def \_\_init\_\_(self, fn, \*args, \*\*kwargs):

self.file = fn(\*args, \*\*kwargs) #yielding file\_handle

def \_\_enter\_\_(self):

print("From \_\_enter\_\_ Context created, returning file handle to the context object")

return next(self.file)

def \_\_exit\_\_(self, exc\_type, exc\_value, exc\_tb):

try:

next(self.file)

except StopIteration:

pass

finally:

print("From \_\_exit\_\_ closing the file")

return False

with Context\_Mgr(open\_file, "test.txt","w") as obj:

x = "The rowdy baby"

obj.write(x)

print("the file written")

print("File closed ?", obj.closed)

f = open("test.txt")

p = f.readline()

print(p)

output:

From \_\_enter\_\_ Context created, returning file handle to the context object

Opening file and returning file handle to the context manager

the file written

From \_\_exit\_\_ closing the file

File closed ? True

The rowdy baby

#Context manager decorator

def context\_manager\_dec(gen\_fn):

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

gen = gen\_fn(\*args, \*\*kwargs)

ctx = GenContextManager(gen)

return ctx

return helper

@context\_manager\_dec

def open\_file(fname, mode='r'):

print('opening file...')

f = open(fname, mode)

try:

yield f

finally:

print('closing file...')

f.close()

class GenContextManager:

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

self.gen = gen

def \_\_enter\_\_(self):

return next(self.gen)

def \_\_exit\_\_(self, exc\_type, exc\_value, exc\_tb):

print('calling next to perform cleanup in generator')

try:

next(self.gen)

except StopIteration:

pass

return False

file\_gen = open\_file('test.txt', 'w')

with GenContextManager(file\_gen) as f:

f.writelines('Sir Spamalot')

Contextlib.contextmanager – which will convert the generator into a context manager

from contextlib import contextmanager

@contextmanager

def open\_file(fname, mode='r'):

print('opening file...')

f = open(fname, mode)

try:

yield f

finally:

print('closing file...')

f.close()

with open\_file('test.txt') as f:

print(f.readlines())

ouput:

opening file...

['The rowdy baby']

closing file...

Nested context managers can be opened within another. Indenting out each one will close the context.

from contextlib import contextmanager

@contextmanager

def open\_file(f\_name):

print(f'opening file {f\_name}')

f = open(f\_name)

try:

yield f

finally:

print(f'closing file {f\_name}')

f.close()

f\_names = 'file1.txt', 'file2.txt', 'file3.txt'

enters = []

exits = []

for f\_name in f\_names:

ctx = open\_file(f\_name)

enters.append(ctx.\_\_enter\_\_)

exits.append(ctx.\_\_exit\_\_)

files = [enter() for enter in enters]

while True:

try:

rows = [next(f).strip('\n') for f in files]

except StopIteration:

break

else:

row = ','.join(rows)

print(row)

for fn in exits[::-1]:

fn(None, None, None)

output:

opening file file1.txt

opening file file2.txt

opening file file3.txt

file1\_line1,file2\_line1,file3\_line1

file1\_line2,file2\_line2,file3\_line2

file1\_line3,file2\_line3,file3\_line3

closing file file3.txt

closing file file2.txt

closing file file1.txt

concurrent (run A, run B, again run A, run B) / parallel multitasking (multi processor runs both tasks together.)

cooperative (task A yields, task B yields - coroutines) / preemptive (scheduler preempts task A, task B)

python programs run on a sigle thread. Generators - coroutines use ‘yield’ , asyncio.

Native coroutines use – async – await

Generator based coroutintes:

A deque is a double ended queue. A generator is called using next()

Dq.appendleft, dq.popleft, dq.pop, dq.append, dq.len, dq.maxlen

#coroutine using generator

from collections import deque

def producer(dq,n):

for i in range (1,n):

dq.appendleft(i)

if len(dq) == dq.maxlen:

print("Q is full - yielding")

yield

def consumer(dq):

while True:

while len(dq) > 0:

print("Processing ..", dq.pop())

print("Q empty, yielding")

yield

def cordinator():

dq = deque(maxlen=5)

prod = producer(dq,15)

con = consumer(dq)

while True:

try:

next(prod)

except StopIteration:

break

finally:

next(con)

cordinator()

output:

Q is full - yielding

Processing .. 1

Processing .. 2

Processing .. 3

Processing .. 4

Processing .. 5

Q empty, yielding

Q is full - yielding

Processing .. 6

Processing .. 7

Processing .. 8

Processing .. 9

Processing .. 10

Q empty, yielding

Processing .. 11

Processing .. 12

Processing .. 13

Processing .. 14

Q empty, yielding

Generator states – created, running, suspended(yields), closed(return)

Inspect.getgeneratorstate()

Yield can (a) yield a value and suspend, (b) receive a value from outside a generator.

def func\_gen():

recieved = yield "hello from generator" #yielding value to invoker

print(recieved)

def get\_gen():

gen = func\_gen()

try:

a = next(gen) # yielded value is received

print(a)

gen.send("Hai hawai.. reply from invoker") # sending to yield

except StopIteration:

pass

get\_gen()

output:

hello from generator

Hai hawai.. reply from invoker

‘priming a generator’ is to make the generator in a suspended state before sending the data to it >(yield).

Calling gen.close() – will raise a GeneatorExit exception and will run finally clause.

GeneratorExit does not inherit from Exception.

After catching the GE exception, either should return or raise another exception. Otherwise, Python will raise an exception.

Throw(Exception) to the suspended state of yield. Generator catches the exception, handles and yields.

#coroutine decorator

from inspect import getgeneratorstate

def coroutine(fn):

def inner():

gen = fn()

next(gen)

return gen

return inner

@coroutine

def gen():

try:

while True:

rec = yield

print( rec \*\* 2)

except TypeError:

pass

a = gen()

a.send(10)

status = getgeneratorstate(a)

print(status)

inspect.getgeneratorlocals()

#delegator

def squares(n):

for i in range(n):

yield i \*\* 2

def delegator(n):

for value in squares(n):

yield value

gen = delegator(5)

for \_ in range(5):

print(next(gen), end=” “)

output: 0 1 4 9 16

when the data is a list, use ‘yield from’

#pulling data through pipeline using iterators

from itertools import islice

import csv

# def parse\_data(f\_name):

# f = open(f\_name)

# try:

# dialect = csv.Sniffer().sniff(f.read(2000))

# f.seek(0)

# next(f)

# yield from csv.reader(f, dialect = dialect)

# finally:

# f.close()

# a = parse\_data("cars.csv")

# for i in a:

# print(i)

#produce data

def parse\_data(f\_name):

with open(f\_name) as f:

dialect = csv.Sniffer().sniff(f.read(2000))

f.seek(0)

next(f)

yield from csv.reader(f, dialect=dialect)

#filter data

def filter\_data(rows, contains):

for row in rows:

if contains in row[0]:

yield row

# data = parse\_data("cars.csv")

# filtered\_data = filter\_data(data, "Dodge")

# for i in filtered\_data:

# print(i)

# data.close()

# filtered\_data.close()

#final puller

def output(f\_name, \*args):

data = parse\_data(f\_name)

for arg in args:

data = filter\_data(data, arg)

yield from data

from itertools import islice

result = output("cars.csv", "Chevrolet","Carlo","Landau")

for i in result:

print(i)

output:

['Chevrolet Monte Carlo Landau', '15.5', '8', '350.0', '170.0', '4165.', '11.4', '77', 'US']

['Chevrolet Monte Carlo Landau', '19.2', '8', '305.0', '145.0', '3425.', '13.2', '78', 'US']

#pushing and filtering data through coroutines.

#decorator to autoprime the coroutine

def autoprime\_dec(coroutine):

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

gen = coroutine(\*args, \*\*kwargs)

next(gen)

return gen

return inner

#coroutine printing data

@autoprime\_dec

def print\_data():

while True:

received = yield

print(received)

#transmitter

@autoprime\_dec

def power\_up(nxt\_gen,n):

import math

while True:

received = yield

output = math.pow(received,n)

nxt\_gen.send(output)

@autoprime\_dec

def filter\_even(nxt\_gen):

while True:

received = yield

if received % 2 == 0:

nxt\_gen.send(received)

#print\_dt = print\_data()

filtered = filter\_even(print\_data())

gen2 = power\_up(filtered,3)

gen1 = power\_up(gen2,2)

for i in range(1,6):

gen1.send(i)

output:

64.0

4096.0

**Deepdive 3 HashMaps** (04/09/20)

Itertools.chain chains iterables. Chainmap maps dictionaries.

Dictionaries are also known as associative arrays. They are abstract data structures, means can be implemented in more than one way. Here, it will be implemented using hash tables(hash maps).

Key-sharing dictionaries – Mark Shannon. Compact dictionaries (key order) – Raymond Hettinger

Dictionary keys must be hashable. Key order is maintained in order of insertion.

In a dictionary, object or value is associated to a key. So it’s an associated array.

A = Person([hash(key)] // Person([hash(key)] = 1000

Key points to a hash, where the value will be stored. Key1->0(is index of the array) = 10

Hash of a key is found by considering the key length or value with the size of the table. Probing is used to find the empty slot to store the ‘key and value’ and also to extract the value given the key. Hash of a key should not be changed.

Compact dictionaries has an index and value array. Hash value will look in the index array index, which in turn will provide the index value of the value array where the actual key-value is stored.

Hash values will change from each run, but will remain same in one run.

Sys.hash\_info.width // all immutable objects are hashable as keys

Hash of the object must be an integer value

Dictionaries can be created using: dict(key=value), dict\_a = {“a”:10}, comprehension a = {k : k\*\*2 for k in range(10)}, dict.fromkeys(iterable, defaultvalue) dict.fromkeys([1,(1,0)], “N/A”)

#passing function and its arguments as key-value in dictionary

def add(a,b):

return a + b

def square(a):

return a \*\* 2

def mult(a,b):

return a \* b

funcs = {add:(10,20), square:(5,), mult:(5,3)}

for func, args in funcs.items():

result = func(\*args)

print(result)

output: 30 25 15

to shallow copy a dictionary:

a= {"one":1, "two":2}

b = dict(a)

print(a)

--

d.get(key,[default]) d[‘key’] , len(d), d.clear(), del d[‘key’], d.pop(key, default), d.popitem(), d.setdefault(key,value) // KeyError

Python string | strip()

strip() is an inbuilt function in Python programming language that returns a copy of the string with both leading and trailing characters removed (based on the string argument passed).

Syntax:

string.strip([chars])

Parameter:

There is only one optional parameter in it:

1)chars - a string specifying the set of characters to be removed.

If the optional chars parameter is not given, all leading and trailing whitespaces are removed from the string.

Return Value: Returns a copy of the string with both leading and trailing characters removed.

--

import string

d = dict.fromkeys(string.ascii\_lowercase,"lower")

print(d)

d.keys(), d.values(), d.items() – all are views and iterable. Dictionary views are sets, set operations can be performed.

Set operations: union, intersection, difference. Sets do not have a guaranteed order.

Set union – intersection = symmetric difference –(^)

D1.update(d2/iterable(key,value)/key-vlue pair)

Shallow copy methods: d1 = d.copy(), d = {\*\*d1}, d = dict(d1), d = {k:v for k,v in d.items()}

a = {"a":1,"b":2,"c":3}

b = {k:v for k,v in a.items()}

print(b)

output: {'a': 1, 'b': 2, 'c': 3}

--

d = deepcopy(d1)

By default, custom classes compared == if they have same id(is).

Custom objects are made hashable by using their id’s.

In a custom class if an \_\_eq\_\_ is defined, then Python backs off from hashing the object.

When the \_\_eq\_\_ method is defined in custom class, python automatically sets the attribute \_\_hash\_\_ = None

If \_\_eq\_\_ method is defined, then \_\_hash\_\_ method also has to be defined to make the class object hashable.

#getting the hash of a custom object

class Person:

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

self.name = name

p = Person("Jaison")

print(hash(p))

# class object is unhashable since \_\_eq\_\_ is defined

class Person:

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

self.name = name

def \_\_eq\_\_(self, other):

if self.name == other.name:

print("both from same object")

p = Person("Jaison")

print(hash(p))

# class object is unhashable since \_\_hash\_\_ attribute is set to None

class Person:

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

self.name = name

\_\_hash\_\_ = None

p = Person("Jaison")

print(hash(p))

#class object is hashable since \_\_eq\_\_ returns true and \_\_hash\_\_() method is implemented

class Person:

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

self.name = name

def \_\_eq\_\_(self, other):

return self.name == other.name

def \_\_hash\_\_(self):

return hash(self.name)

p = Person("Jaison")

print(hash(p))

p1 = Person("Jaison")

print(hash(p1))

d = {p:100}

print(d[p1]) # retrieving value with the other object

if the class object is immutable, its safe to use inside the dictionary as key. To make the object immutable, define some properties in the class as readonly, use \_variablename to make it private.

Use that property for equality in \_\_eq\_\_() and hashvalue under \_\_hash\_\_()

**Set:** a set is a gathering together into a whole of definite, distinct objects of our perception or of our thought – which are called the elements of the set.

A set is an unordered collection of distinct object. Its distinct, un ordered elements.

Sets are implemented as hashmaps in python. Set elements should be hashable.

Element equality (==) should be false. Sets maintain a hash table of its elements

Set methods: union (|), intersecton(&), difference (-), symmetric difference

S1 | s2 // s1.union(s2,s3, ..) // s1 & s2 // s1.intersection(s2,s3..) // s1 – s2 // s1.difference(s2)

Symmetric difference is the (union of s1 with s2) – (intersection of s1 with s2) // s1 ^ s2 // s1.symmetric\_dfference(s2)

Cardinality of a set is the len(set).

Two sets are said to be ‘disjoint’ if their intersection is empty. S1.disjoint(s2)

Subset – if all the elements of s2 is there in s1, then s1 is a subset of s2. S1 <= s2 // s1.issubset(s2)

Proper subset is when s1 contains all the elements of s2 and s2 has additional elements. S1 < s2

S1 is superset if it has additional elements than s2. S1 >= s2 // s1.issuperset(s2)

Set is a mutable collection. Since set is not hashable. (set elements must be hashable)

A frozenset is hashable (immutable). A set can be converted into a frozenset. No literal notation for frozenset.

List is not hashable.

#Creating set

a = "the quich brown fox jumps over the lazy dog"

b = {c for c in a}

c = set("pthon")

d = {"Python"}

print(b)

print(c)

print(d)

output:

{'f', 'm', 'b', 'z', 'v', 'h', 'w', ' ', 'y', 'a', 'o', 'g', 't', 'n', 'e', 'p', 'q', 'l', 'u', 'd', 'c', 'i', 'r', 'j', 'x', 's'}

{'o', 't', 'n', 'h', 'p'}

{'Python'}

Common set method: len, in, not in. s1.add(), s1.remove(), s1.discard(), s.pop(), s.clear()

Set update methods: s1 |= s2 (s1.update(s2)) // s2 &= s2 (s1.intersection\_update(s2)) // s1.difference\_update(s2), s1.symmetric\_difference\_update(s2)

Shallow copy: s1.copy(), s1 = set(s2), s1 = {\*s1} // s1.deepcopy(s2)

#creat a dictionary from 2 sets

a = {k:v for k,v in zip({"a","b","c","d","e"},{1,2,3,4,5})}

print("set",a)

#creating a dictionary from a frozenset

a = {k:v for k,v in zip(frozenset(["a","b","c","d","e"]),{1,2,3,4,5})}

print("frozenset",a)

output:

set {'d': 1, 'c': 2, 'a': 3, 'b': 4, 'e': 5}

frozenset {'d': 1, 'c': 2, 'a': 3, 'b': 4, 'e': 5}

a = {"a","b","c","d","e"}

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

c = {\*a,\*b}

d = {k:v for k,v in zip(a,b)}

print(c) #set

print(d) #dictionary

output:

{1, 2, 3, 'd', 'c', 'a', 4, 5, 'b', 'e'}

{'d': 1, 'c': 2, 'a': 3, 'b': 4, 'e': 5}

Keys of a dictionary always behave like a set.

Python does not allow modifying the size of a dictionary while iterating over a view. But, …

#modifying a dictionary during iteration over a view

d = dict(zip("abc",[1,2,3]))

for k,v in d.items():

print(k,v)

d['c']=100 # modified even before the view reaching 'c'

output:

a 1

b 2

c 100

**Serialization and de-serialization:** is useful for persistence and transmission of data.

Serializing: creating a persistent representation of data

Deserializing: reconstructing the object from the serialized data

Pickling and unpickling: Python mechanism for serializing and deserializing objects using binary mechanism.

JSON is a long string of text representation

Serializing and deserializing together called marshalling.

Import pickle

Dump > pickle an object to a file

Load > unpickle a file into an object

Dumps > pickle an object to a string

Loads > unpickle a string to an object

#Serialization / pickling / unpickling using dump and load

import pickle

from datetime import datetime

d = dict(a=100,b="jaison",c=datetime.utcnow())

with open("mypickle.txt","wb") as f:

pickle.dump(d,f)

with open("mypickle.txt","rb") as f:

reader = pickle.load(f)

print(reader)

output:

{'a': 100, 'b': 'jaison', 'c': datetime.datetime(2020, 4, 10, 22, 24, 22, 912135)}

#Serialization / pickling / unpickling using dumps and and loads

import pickle

m=dict(C="a", D="c",k="m")

a = pickle.dumps(m)

print("Pickling",a) # binary format of pickling

b = pickle.loads(a)

print("Unpickling",b) # unpickling

output:

Pickling b'\x80\x03}q\x00(X\x01\x00\x00\x00Cq\x01X\x01\x00\x00\x00aq\x02X\x01\x00\x00\x00Dq\x03X\x01\x00\x00\x00cq\x04X\x01\x00\x00\x00kq\x05X\x01\x00\x00\x00mq\x06u.'

Unpickling {'C': 'a', 'D': 'c', 'k': 'm'}

JSON is a text based object serialization. It’s a common format for web API’s and general data interchanged between systems.

JSON is strings. Uses very limited datatypes: strings use “”, all numbers are considered as floats, Booleans as true / false, list arrays, dictionaries – ‘keys’ must be strings, empty value is null.

Infinity, -Infinity, NaN

A tuple in python will be converted to a list in JSON

‘vars’ function returns the attributes of an object as a dictionary.

class A:

def \_\_init\_\_(self,a,b):

self.a = a

self.b = b

self.c = None

a = A(10,20)

vars(a)

output: {'a': 10, 'b': 20, 'c': None}

datetime standard format is ISO8601 – YYYY-MM-DDTHH:MM:SS.SSSSS+-05:30 (year-month-dateTimehour:minute:second.millisecond+-offset from UTC)

date.isoformat() – will convert date into a isoformatted string. Date.strftime(‘%Y%m%dT%H%M%S)

dump/dumps(object, default=func) – default function is the serializer function for the unknown datatypes in the object. Func takes only one argument for the unknown data type to be passed by Python.

from datetime import datetime

from decimal import Decimal

from functools import singledispatch

import json

class Person:

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

self.name = name

self.age = age

self.create\_date = datetime.utcnow()

def toJSON(self):

return {self.name, self.age, self.create\_date}

def \_\_repr\_\_(self):

return f"Person name = {self.name} age = {self.age}"

class Point:

def \_\_init\_\_(self,x,y):

self.x = x

self.y = y

def \_\_repr\_\_(self):

return f"Point x = {self.x} y = {self.y}"

@singledispatch

def format(arg):

try:

print("\ttrying toJson")

return arg.toJSON()

except AttributeError:

try:

print("\ttrying vars")

return vars(arg)

except TypeError:

print("\ttrying str")

return str(arg)

@format.register(datetime)

def to\_date(arg):

return arg.isoformat()

@format.register(set)

def from\_set(arg):

return list(arg)

@format.register(Decimal)

def to\_decimal(arg):

return f"Decimal {str(arg)}"

p = Person("Jaison",45)

pt1 = Point(0,5)

log\_record = {"a":10,"b":{1,2},"c":(1,2,3),"date":datetime.utcnow(),"Person":p,"Point":pt1,"time":datetime.utcnow()}

d = json.dumps(log\_record, indent=2, default=format)

print(d)

output:

trying toJson

trying toJson

trying vars

{

"a": 10,

"b": [

1,

2

],

"c": [

1,

2,

3

],

"date": "2020-04-11T01:33:47.281022",

"Person": [

"2020-04-11T01:33:47.281022",

"Jaison",

45

],

"Point": {

"x": 0,

"y": 5

},

"time": "2020-04-11T01:33:47.281022"

}

#custom encoder overriding the 'default' and specifying the default kw in super().\_\_init\_\_()

from json import JSONEncoder

from datetime import datetime

class CustomJsonEncoder(JSONEncoder):

def \_\_init\_\_(self,\*args,\*\*kwargs):

super().\_\_init\_\_(skipkeys=True, allow\_nan=True, separators=(' / ', ' = '))

def default(self, arg):

if isinstance(arg, datetime):

return arg.isoformat()

else:

return super().default(arg)

d = {

"date":datetime.utcnow(),

"abc":"text",

1+1j:"complex",

"inf":float('Infinity'),

"nan": float('NaN')

}

ds = json.dumps(d, cls=CustomJsonEncoder)

print(ds)

output:

{"date" = "2020-04-11T03:49:17.087459" / "abc" = "text" / "inf" = Infinity / "nan" = NaN}

JSONDecoder: load(object\_hook / object\_pairs\_hook, parse\_float=function)

Object\_hook passes the deserialized dictionary to the callable – no ordering. Object\_pairs\_hook passes it as a list of tuple ordered.

Object\_hook and object\_pairs\_hook receives a parsed object.

#object decoder uses loads

import json

j='''

{

"int":100,

"float":0.5,

"nan":NaN,

"inf":-Infinity,

"car": {

"wheel":"round",

"steer":"pinion"

}

}

'''

x = '''{

"a1":"a1",

"b1":10.05,

"c1":100,

"d1":[10,20,30],

"f1":{"s1":10,"s2":["x1","x2"]},

"g1":{"m1":{"r1":10,"r2":20}}

}

'''

def parse\_int(arg):

return int(arg)

def parse\_float(arg):

return float(arg)

def parse\_constant(arg):

return None

def obj\_hook(arg):

return str(arg)

js = json.loads(j,object\_hook=obj\_hook, parse\_int=parse\_int, parse\_float=parse\_float, parse\_constant=parse\_constant)

print(js,"\n")

js = json.loads(x,object\_hook=obj\_hook, parse\_int=parse\_int, parse\_float=parse\_float, parse\_constant=parse\_constant)

print(js)

output:

{'int': 100, 'float': 0.5, 'nan': None, 'inf': None, 'car': "{'wheel': 'round', 'steer': 'pinion'}"}

{'a1': 'a1', 'b1': 10.05, 'c1': 100, 'd1': [10, 20, 30], 'f1': "{'s1': 10, 's2': ['x1', 'x2']}", 'g1': '{\'m1\': "{\'r1\': 10, \'r2\': 20}"}'}

JSONDecoder: decode method

Regular expression. ‘r’ raw string indicates python that to leave the escape characters in the string.

Pattern = r’”pattern”’ // re.compile(pattern) // pattern.search(“text”) || re.search(pattern, “text”)

#JSONDecoder

import json

class Point:

def \_\_init\_\_(self,x,y):

self.x = x

self.y = y

def \_\_repr\_\_(self):

return(f"(Point x = {self.x} y = {self.y})")

return f'Point(x={self.x}, y={self.y})'

class Custom\_Decoder(json.JSONDecoder):

def decode(self,arg):

obj = json.loads(arg)

if 'points' in obj:

obj['points'] = [Point(x,y) for x,y in obj['points']]

return obj

j ='''{

"points":[

[10,20], [199,299]

]

}'''

result = json.loads(j, cls=Custom\_Decoder)

print(result)

--

Output:

{'points': [(Point x = 10 y = 20), (Point x = 199 y = 299)]}

JSON schema. We can describe the structure of a JSON object, using JSON schema. Json-schema.org

Pip install jsonschema – a library, which can be used to validate a Jason document with a json schema.

#JSONSCHEMA

from jsonschema import validate

from jsonschema.exceptions import ValidationError

from json import dumps, loads, JSONDecodeError

#jsonschema

my\_schema = {

"type":"object",

"properties":{"firstName":{"type":"string","minlength":1},

"middleName":{"type":"string","minLength":1,"maxLength":1},

"lastName":{"type":"string","minLength":1},

"age":{"type":"integer","minimum":1,"maximum":100},

"eyeColor":{"type":"string","enum":["amber","blue","green","red","orange","white"]}

},

"required":["firstName","lastName"]

}

#the JSON object must be str, bytes or bytearray, not dict

p\_data = '''{

"firstName":"Jaison",

"middleName":"v",

"lastName":"Jacob",

"age":45,

"eyeColor":"blue"

}'''

try:

a = validate(loads(p\_data), my\_schema)

except JSONDecodeError as ex:

print(ex)

except ValidationError as ex:

print(ex)

else:

print("data is conforms to schema")

print(loads(p\_data))

-----

Output:

data is conforms to schema

{'firstName': 'Jaison', 'middleName': 'v', 'lastName': 'Jacob', 'age': 45, 'eyeColor': 'blue'}

Marshmallow can validate, serialize, deserialize objects, not only JSON, but others also.

from marshmallow import Schema, fields

from datetime import date

class Person:

def \_\_init\_\_(self,fname,lname,dob):

self.fname = fname

self.lname = lname

self.dob = dob

class Person\_Schema(Schema):

fname = fields.Str()

lname = fields.Str()

dob = fields.Date()

p1 = Person("Jaison","Jacob",date(1975,3,23))

person\_schema = Person\_Schema()

a = person\_schema.dumps(p1)

print(a)

output: {"dob": "1975-03-23", "lname": "Jacob", "fname": "Jaison"}

Specialized dictionaries:

defaultdict – for creating keys with default value if it doesn’t exists.

Orderedict – for insertion key order.

Counter: specialized tools for dealing with counters – multi set

Chainmap: for mapping of dictionaries

Userdict: alternative for subclassing ‘dict’

#defaultdict

from collections import defaultdict #automatic insertion of default value to the key if not specified

a = defaultdict(lambda :0) #factory funciton

txt = 'Malayalamalayalam'

for c in txt:

a[c] += 1

print(a)

output: defaultdict(<function <lambda> at 0x0000024CA5565EE8>, {'M': 1, 'a': 8, 'l': 4, 'y': 2, 'm': 2})

from collections import defaultdict, namedtuple

from datetime import datetime

from functools import wraps

def function\_stats():

d = defaultdict(lambda: {"count": 0, "first\_called": datetime.utcnow()})

Stats = namedtuple('Stats', 'decorator data')

def decorator(fn):

@wraps(fn)

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

d[fn.\_\_name\_\_]['count'] += 1

return fn(\*args, \*\*kwargs)

return wrapper

return Stats(decorator, d)

stats = function\_stats()

@stats.decorator

def func\_1():

pass

@stats.decorator

def func\_2(x, y):

pass

func\_1()

func\_2(10,20)

stats.data

#dict(stats.data)

Output:

defaultdict(<function \_\_main\_\_.function\_stats.<locals>.<lambda>()>,

{'func\_1': {'count': 1,

'first\_called': datetime.datetime(2020, 4, 12, 21, 59, 17, 834753)},

'func\_2': {'count': 1,

'first\_called': datetime.datetime(2020, 4, 12, 21, 59, 17, 835130)}})

OrderedDict uses 1) reverse iteration, 2) pop first, last item in a dictionary, 3) move item to beginning or to end of a dictionary.

#reverse iteration in OrderedDict

from collections import OrderedDict

d = OrderedDict()

d["one"]=1

d["two"]=2

d["three"]=3

for k in reversed(d):

print(k)

output:

three

two

one

d.popitem() – removes from end // d.popitem(last=False) – removes from front.

d.move\_to\_end(item) – moves item to end. // d.move\_to\_end(item, last=False)

in OrderedDict, ordering does matter.

Keys view keys() acts like a set. No ordering.

def popitem(d, last=True):

if last:

return d.popitem()

else:

first\_key = next(iter(d.keys()))

return first\_key, d.pop(first\_key)

d2 = dict(a=1, b=2, c=3, d=4)

print(d2)

print(popitem(d2))

print(d2)

d2 = dict(a=1, b=2, c=3, d=4)

print(d2)

print(popitem(d2, last=False))

print(d2)

d2 = dict(a=1, b=2, c=3, d=4)

print(d2)

key = 'b'

d2[key] = d2.pop(key)

print(d2)

d = dict(a=1, b=2, c=3, d=4, e=5, f=6)

key = 'c'

print(d.keys())

# first move desired key to end

d[key] = d.pop(key)

print(d.keys())

keys = list(d.keys())[:-1]

for key in keys:

d[key] = d.pop(key)

print(d.keys())

print(d)

def move\_to\_end(d, key, \*, last=True):

d[key] = d.pop(key)

if not last:

for key in list(d.keys())[:-1]:

d[key] = d.pop(key)

d = dict(a=1, b=2, c=3, d=4, e=5, f=6)

move\_to\_end(d, 'c')

print(d)

move\_to\_end(d, 'c', last=False)

print(d)

Counter is a specialized class in the collections module.

#Counter counts the number if elelments in the iterable

from collections import Counter

cnt = Counter("Hello my dear friends, lets party tonight")

print(cnt)

output: Counter({' ': 6, 'e': 4, 't': 4, 'l': 3, 'r': 3, 'o': 2, 'y': 2, 'd': 2, 'a': 2, 'i': 2, 'n': 2, 's': 2, 'H': 1, 'm': 1, 'f': 1, ',': 1, 'p': 1, 'g': 1, 'h': 1})

Counter frequently counting the keys.

Counter.most\_common(list), counter.elements()

cnt = Counter("Hello my dear friends, lets party tonight")

for c in cnt.elements():

print(c)

c1.update(c2), c1.subtract(c2), to find the minimum of 2 counters: c1 & c2, to find maximum of 2 counters: c1 | c2

unary –c1, +c1 operators operate with the signed values in the counter object.

The result of a chainmap will not have repeated keys.

ChainMap.parents, d.new\_child(d4), d.maps, d.maps.append(d5), d.maps.insert(0,d6), del d.maps[1]

#adding new configurations without modifying the previous one

from collections import ChainMap

config = {"root":"d","bin":"bin","pwd":"123@abc"}

b = ChainMap({},config)

b["root"]="c"

b["pwd"]="1@Abcd"

b["user"]="Kelvin"

print(list(b.items()))

print(b)

output: [('root', 'c'), ('bin', 'bin'), ('pwd', '1@Abcd'), ('user', 'Kelvin')]

ChainMap({'root': 'c', 'pwd': '1@Abcd', 'user': 'Kelvin'}, {'root': 'd', 'bin': 'bin', 'pwd': '123@abc'})

Dict’s have the \_\_getitem\_\_ and \_\_setitem\_\_ methods.

Collections.UserDict is not technically a subclass of ‘dict’. It uses regular dict for its datastructure and implements all the functionalities of a dict. Its not a dict. It’s a mapping type.

No gurantee that the built in class will use its default methods when called through inheritance.

class LimitedDict(UserDict):

def \_\_init\_\_(self, keyset, min\_value, max\_value, \*args, \*\*kwargs):

self.\_keyset = keyset

self.\_min\_value = min\_value

self.\_max\_value = max\_value

super().\_\_init\_\_(\*args, \*\*kwargs)

def \_\_setitem\_\_(self, key, value):

if key not in self.\_keyset:

raise KeyError('Invalid key name.')

if not isinstance(value, int):

raise ValueError('Value must be an integer type.')

if value < self.\_min\_value or value > self.\_max\_value:

raise ValueError(f'Value must be between {self.\_min\_value} and {self.\_max\_value}')

super().\_\_setitem\_\_(key, value)

d = LimitedDict({'red', 'green', 'blue'}, 0, 255, red=10, green=10, blue=10)

output: {'red': 10, 'green': 10, 'blue': 10}

**OOPS:** 04/14/20 review

Properties are hybrids between functions and data.

Object is a container that contains the data (state) and functions (behavior). Both data and functions are called the attributes of the object.

Class is a template (type) or blueprint to create the container objects

Meta classes allows us to create classes.

The type of a class is ‘type’

class A:

pass

type(A)

output: type

--

Python also creates attributes to the objects.

Getattr(object\_symbol, attribute\_name, default\_value\_if\_attributed\_doesn’t\_exist) // class.\_\_dict\_\_[‘attribute’]

setattr(object\_symbol, attribute\_name, value)

delattr(object\_symbol, attribute\_name) // del class.attribute

class A:

pass

setattr(A,"name", "Jaison")

getattr(A,"name")

the ‘state’ of a class attributes are stored in the class \_\_dict\_\_ attribute. It returns a ‘mappingproxy’ object.

Class instance objects has their own attribute \_\_dict\_\_

Class.\_\_dict\_\_ returns a mappingproxy. Object.\_\_dict\_\_ returns a dictionary, which can be mutated.

When an attribute is accessed through the instance, python first checks the objects dictionary, then looks for class’s dictionary – getattr also does the same.

Methods are objects that combine instance and function. It has attribute \_\_self\_\_, \_\_func\_\_

\_\_init\_\_ is called as a bound method. Obj.\_\_init\_\_(args) – after creating the object.

\_\_new\_\_ function creates the object.

#binding a function to an instance object

from types import MethodType

class A:

def func(self):

print(self)

obj = A()

obj.obj\_func = MethodType(lambda self: "obj\_func", obj) # it’s a method

obj.say = lambda: "Say hello" # its just a function

obj.obj\_func()

output: 'obj\_func'

‘property’ class constructor: property(fget=function, fset=function, fdel=function, doc=’docstring’)

#decorator

def decor(fn):

print("decor called")

def inner(\*arg,\*\*kwarg):

print("inside inner")

result = fn(\*arg,\*\*kwarg)

return result

return inner

@decor

def add\_(\*args):

l = len(args)

result = 0

for i in range(l):

result += args[i]

return result

add\_(10,20,30,10)

output: decor called

inside inner

70

Property is used to access the data attributes of an object.

Urllib module. Requests is a 3rd party library.

A classmethod will be bound to the ‘class’ irrespective of it calling from a class or from an instance. ‘cls’ is the default argument.

Staticmethod will ‘never’ bound to any object when called, either it is called from a class or an instance. No default argument in staticmethod definition.

3rd party library to work with time. Pytz.

‘types’ module contains function, module, generator types.

The ‘scope’ of a symbol defined inside the function in a class is ‘module’ scope, not ‘class scope’.

Functions scope is not nested within the ‘class scope’

All comprehensions works like functions and in ‘scope’ levels.

Polymorphysm:

We can iterate over an object with function iter(), only if the object implements the iterator protocol.

Useful special methods:

\_\_init\_\_ for instantiation of objects

\_\_enter\_\_, \_\_exit\_\_ for contexts

\_\_getitem\_\_, \_\_setitem\_\_, delitem\_\_ for sequences

\_\_iter\_\_, \_\_next\_\_ for iterables and iterators

\_\_len\_\_ for len() and \_\_contains\_\_ for ‘in’

\_\_str\_\_ str() for users \_\_repr\_\_ (for developers) repr() - both used to create the string representation of the instance object. Python will first look for \_\_str\_\_ if not, will see \_\_repr\_\_.

\_\_matmul\_\_ is for ‘@’ in support for numpy.

When an operation is not supported or method not implemented, return ‘NotImplemented’.

Reflected operators use the right hand side argument on left and use the ‘r’ version of the operators, if the function returns NotImplemented and the operands are of different types. If, a\_\_add\_\_b doesn’t work, use b\_\_radd\_\_a.

Rich comparison operators will flip the operands and operator if it return NotImplemented.

Every class inherits from ‘object’ class.

To make an instance hashable, only the equality and hashable attribute of the class need to be made immutable, by making it a private ‘and property’.

If a class doesn’t implement \_\_bool\_\_ or \_\_len\_\_, its considered True.

‘partial’ is a callable class.

Check an object is callable by using function ‘callable(obj)’

Format(value, format-spec) if ‘format-spec’ is not specified, then it is str(value) // \_\_format\_\_

Python allows multiple class inheritance.

Isinstacnce, issubclass, type

Type of an object is its class. Type of a class is ‘type’

Every class in python implicitly inherits from ‘object’ class.

Some builtin and standard library classes are in lowercase, because it follows ‘c’.

Import types // dir(types)

Binding of methods to ‘which object?’ is important in calling the overridden methods.

Hasattr(object, attribute)

Delegating to a parent – calling the method specifically in the ancestry hierarchy.

Delegatations works its way up in the inheritance hierarchy until it finds what it needs.

The whole point of object oriented programming is to reduce code.

Super().\_\_init\_\_ inside the current class \_\_init\_\_ will redefine the instantiation of the current object.

#super

class A:

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

self.name = name

self.age = age

self.sex = sex

class B(A):

def \_\_init\_\_(self,\*args):

self.name = args[0] + "'s"

super().\_\_init\_\_(\*args) # super surpasses the value set by this class's init

b = B("Thomas",'M',50,'India')

b.\_\_dict\_\_

output: {'name': 'Thomas', 'age': 50, 'sex': 'M'}

Delegated methods are bound to the calling instance

Predefined attributes can be stored in \_\_slots\_\_. If a class has \_\_slots\_\_ defined, then the instance.\_\_dict\_\_ will not be created. ‘Only’ the predefined attributes in the slots can be used in the class. Slots saves memory.

class A:

\_\_slots\_\_ = ("x","y")

def \_\_init\_\_(self, x,y):

self.x = x

self.y = y

A.\_\_slots\_\_

Output: ('x', 'y')

Subclass will use slots from the parents (if present), and will also use an instance dictionary.

‘properties’ are stored in class dictionary.

Specifiying \_\_dict\_\_ inside \_\_slots\_\_ as a attribute will enable the instance to have its own dictionary to store extra attributes.

\_\_slots\_\_ = (“name”, “\_\_dict\_\_”)

Datadescriptors are classes (also protocol) that implement certain methods: \_\_get\_\_, \_\_set\_\_, \_\_delete\_\_, \_\_set\_name\_\_

If a class inheriting another class has a \_\_slots\_\_, and the parent class has attributes, (but no slots), the inherited class will have both \_\_solts\_\_ and instance \_\_dict\_\_

Datadescriptors allow us to learn how properties, methods, functions and \_\_slots\_\_ work. Its is the underpinning mechanism.

Data descriptors, non-data descriptors. Properties use descriptor protocol methods.

Non-data descriptors implement only the \_\_get\_\_ method.

For non-data descriptors, it will first look for item in the instance dictionary.

#non-data descriptor

from datetime import datetime

class MyTime:

def \_\_get\_\_(self,instance, owner\_class): #non-data descriptor protocol methods

return datetime.utcnow().isoformat()

class Logger:

time = MyTime()

Logger.time

\_\_set\_\_(self, instance, value) // setters and deletters are always called from instance.

Non-data descriptor instance is usually the class attribute of the implementer class.

‘weak reference’ to the object does not affect the reference count as far as the memory manager is concerned.

weakref.ref(object) // from weakref import WeakKeyDictionary

most of the built-in types does not support weakref. But custom class do

import ctypes

import weakref

class A:

pass

def ref\_count(address):

return ctypes.c\_long.from\_address(address).value

a = A()

b = a

weak = weakref.ref(b)

print(ref\_count(id(a)))

sys.getrefcount(a)

--

weakref.getweakrefcount() // d.keyrefs()

\_\_hash\_\_ returns hash(self.attribute)

#uing weakref in descriptor

import weakref

class IntegerValue:

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

self.value = weakref.WeakKeyDictionary()

def \_\_set\_\_(self, instance, value):

if isinstance(value, int):

self.value[instance] = value

else:

raise ValueError

def \_\_get\_\_(self, instance, owner\_class):

if instance is None:

return self

else:

return self.value.get(instance)

class Logger:

x = IntegerValue()

a1 = Logger()

a2 = Logger()

a3 = Logger()

a1.x = 105

a2.x = 95

Logger.x.value.keyrefs()

Output: [<weakref at 0x0000025C4E24FE58; to 'Logger' at 0x0000025C4E1A3A08>,

<weakref at 0x0000025C4E24F2C8; to 'Logger' at 0x0000025C4E1A31C8>]

# passing a callable to a weakref

import weakref

def object\_destroy(obj):

print(f"object {obj} destroyed")

class A:

pass

a = A()

w = weakref.ref(a, object\_destroy) #callable

del a

output: object <weakref at 0x0000025C4E1A8458; dead> destroyed

\_\_set\_name\_\_ was introduced in python 3.6. This method is called when the descriptor is first(once) instatiated.

class ValidString:

def \_\_set\_name\_\_(self, owner\_class, property\_name):

print(f'\_\_set\_name\_\_ called: owner={owner\_class}, prop={property\_name}')

self.property\_name = property\_name

def \_\_get\_\_(self, instance, owner\_class):

if instance is None:

return self

else:

print(f'\_\_get\_\_ called for property {self.property\_name} '

f'of instance {instance}')

class Person:

first\_name = ValidString()

last\_name = ValidString()

output:

\_\_set\_name\_\_ called: owner=<class '\_\_main\_\_.Person'>, prop=first\_name

\_\_set\_name\_\_ called: owner=<class '\_\_main\_\_.Person'>, prop=last\_name

class ValidString:

def \_\_init\_\_(self, min\_length):

self.min\_length = min\_length

def \_\_set\_name\_\_(self, owner\_class, property\_name):

self.property\_name = property\_name

def \_\_set\_\_(self, instance, value):

if not isinstance(value, str):

raise ValueError(f'{self.property\_name} must be a string.')

if len(value) < self.min\_length:

raise ValueError(f'{self.property\_name} must be at least '

f'{self.min\_length} characters'

)

instance.\_\_dict\_\_[self.property\_name] = value

def \_\_get\_\_(self, instance, owner\_class):

if instance is None:

return self

else:

print (f'calling \_\_get\_\_ for {self.property\_name}')

return instance.\_\_dict\_\_.get(self.property\_name, None)

class Person:

first\_name = ValidString(1)

last\_name = ValidString(2)

p = Person()

p.first\_name = 'Alex'

p.\_\_dict\_\_

output:

{'first\_name': 'Alex'}

Data descriptor (using get and set or delete) always overrides the instance dictionary.

For non-data descriptors, it will first look in the instance dictionary.

class ValidString:

def \_\_init\_\_(self, min\_length):

self.min\_length = min\_length

def \_\_set\_name\_\_(self, owner\_class, prop\_name):

self.prop\_name = prop\_name

def \_\_set\_\_(self, instance, value):

print('calling \_\_set\_\_ ...')

if not isinstance(value, str):

raise ValueError(f'{self.prop\_name} must be a string.')

if len(value) < self.min\_length:

raise ValueError(f'{self.prop\_name} must be '

f'at least {self.min\_length} characters.'

)

setattr(instance, self.prop\_name, value)

def \_\_get\_\_(self, instance, owner\_class):

if instance is None:

return self

else:

return instance.\_\_dict\_\_.get(self.prop\_name, None)

class Person:

name = ValidString(1)

p = Person()

p.name = 'Alex'

calling \_\_set\_\_ ...

calling \_\_set\_\_ ...

calling \_\_set\_\_ ...

property objects are data descriptors. It has \_\_get\_\_, \_\_set\_\_, \_\_delete\_\_ methods.

class MakeProperty:

def \_\_init\_\_(self, fget=None, fset=None):

self.fget = fget

self.fset = fset

def \_\_set\_name\_\_(self, owner\_class, prop\_name):

self.prop\_name = prop\_name

def \_\_get\_\_(self, instance, owner\_class):

print('\_\_get\_\_ called...')

if instance is None:

return self

if self.fget is None:

raise AttributeError(f'{self.prop\_name} is not readable.')

return self.fget(instance)

def \_\_set\_\_(self, instance, value):

print('\_\_set\_\_ called...')

if self.fset is None:

raise AttributeError(f'{self.prop\_name} is not writable.')

self.fset(instance, value)

class Person:

def get\_name(self):

return self.\_name

def set\_name(self, value):

self.\_name = value

name = MakeProperty(fget=get\_name, fset=set\_name)

p = Person()

p.\_\_dict\_\_

p.name = 'Guido'

output: \_\_set\_\_ called...

the key to using descriptors is its reusability. Good for applying validations also.

Functions implement the non-data descriptor \_\_get\_\_ protocol, which makes it a function or a method depends on from where it is called.

class Person:

def say\_hello(self):

print("Hellow")

p = Person()

bound\_method = Person.say\_hello.\_\_get\_\_(p,Person)

bound\_method()

output: Hellow

#Functions are non-data descriptors in Python. It shows how they associate with a class as a function and as a method with an instance

import types

class MyFunc:

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

self.\_func = func

def \_\_get\_\_(self, instance, owner\_class):

if instance is None:

print("called from class")

return self.\_func

else:

print("callef from instance")

return types.MethodType(self.\_func, instance)

def say\_hello(self):

print("Hellow")

class Person:

hello = MyFunc(say\_hello)

p = Person()

Person.hello

p.hello()

output: called from class

callef from instance

Hellow

Enumerations: how multiple related constants are stored.

Python enumerations are introduced in 3.4 PEP435 enum module, Enum type, IntEnum, IntFlag, Flag

Enumerators are created by subclassing Enum.

Once an enumerator members are defined, the member list and values become immutable, class cannot be subclassed.

Enumeration class attributes are instances of the class.

Enumeration members are always hashable. Enumerations are iterable. Defninition order of members are preserved.

Once an enumeration is declared, its immutable.

An enum class member will have ‘name’ and ‘value’ properties.

Cannot subclass an enumerate class that have members

Enumclass.\_\_members\_\_ gives the members list.

@enum.unique decorator for enum class will generate error for aliases.

By default, every member of the enum class has a ‘truthy’ irrespective of the member value.

Implement truthiness in eumerations by using \_\_bool\_\_

To check if an attribute is a member of an object, hasattr(obj, ‘\_\_iter\_\_’)

To ensure unique values in Enum class, decorate class with: @enum.unique

import enum

class MyClass(enum.Enum):

RED =1

BLUE=2

GREEN=3

def \_\_bool\_\_(self):

return bool(self.value)

def pure\_color(self,value):

return {self: value}

print(bool(MyClass.RED))

MyClass.RED.pure\_color(255) # acceccing the method through class instance (member)

Output: True

{<MyClass.RED: 1>: 255}

import enum

class MyClass(enum.Enum):

RED =1

BLUE=2

GREEN=3

d = {MyClass.RED : "One", MyClass.GREEN: "Green"} # USING THE CLASS MEMBER AS KEY IN DICTIONARY(MEMBERS ARE ALWAYS HASHABLE)

d

output:

{<MyClass.RED: 1>: 'One', <MyClass.GREEN: 3>: 'Green'}

Enum.auto() – to assign automatic values to members of the enumeration. Behind the scenes, uses \_generate\_next\_value\_( name,start,count,last\_value)

import enum

class MyEnum(enum.Enum):

def \_generate\_next\_value\_(name,start,count,last):

if count % 2 == 1:

return last[-1]

else:

return last[-1]+1

AZURE = 1

RED = enum.auto()

BLUE = enum.auto()

CARMINE = 7

GREEN = enum.auto()

AQUAMARINE = enum.auto()

list(MyEnum)

output:

[<MyEnum.AZURE: 1>, <MyEnum.BLUE: 2>, <MyEnum.CARMINE: 7>, <MyEnum.GREEN: 8>]

Exceptions are objects of some classes. They are raised/called when specific events happens in the application.

Compilation exception is syntax errors

Execution exception is the other type of exception

BaseException (SystemExit (sys.exit()), KeyboardInterrupt, GeneratorExit, (Exception )-> Other exceptions)

Bare or naked exception is specifying only the except: under try block.

If a bare exception is provided, as except:, use sys.exc\_info() to get a handle of the exception.

An exception object has minimum 2 attribtes: args, \_\_traceback\_\_. traceback module.

It is easier to ask for forgiveness than it is to get permission. – Grace Hopper

It is easier to ask for permission [EAFP principal]

In a multi except statement, specify the most relevant exception at the top.

Else: clause executes if no exception occurs in the try: except clause. And, if ‘finally’: clause is there, then ‘else’ clause will not execute.

try:

#a=10/'a'

a = 10/1

except TypeError as ex:

print(ex, type(ex))

else:

print("else")

ouput: else

#casefold

a = "AbcdaBCDfghijkLMAN"

b = a.casefold()

c = a.lower()

print(b,c)

output: 'abcdabcdfghijklman'

'abcdabcdfghijklman'

Collection has sequence types(string,list,tuple,range) and mapping types(set,dict)

To hide exception traceback, use ‘from none’ in the raise Exception

try:

raise ValueError("level1")

except ValueError as ex1:

try:

raise ValueError("level2")

except ValueError as ex2:

raise ValueError("level3") from ex1

Meta programming: code to modify code.

Decorators, descriptors, metaclasses (act as class factory) – unless writing a library or framework, metaclasses are not useful.

Object.\_\_new\_\_(cls, \*args,\*\*kwargs) – is a static method. Here, \*args and \*\*kwargs are ignored.

Inorder to work the inheritance properly, use the super().\_\_new\_\_ to create object from the custom class.

If we implement both \_\_new\_\_ and \_\_init\_\_ inside our class, the arguments should match.

class A:

#@staticmethod

def \_\_new\_\_(cls,name):

#return object.\_\_new\_\_(cls)

instance = object.\_\_new\_\_(cls)

print("new called")

return instance

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

self.name = name

print("init called")

a = A("Jaison")

output: new called

init called

class A:

def \_\_new\_\_(cls,w,l):

cls.area = lambda self: w \* l

instance = super().\_\_new\_\_(cls)

instance.w = w

instance.l = l

return instance

a = A(2,3)

a.area()

output: 6

a ‘class’ is an instance of ‘type’ class.

The ‘type’ creates the class instance. Type(class\_name, base\_classes, class\_dictionary)

#using type

class\_name = "Circle"

class\_body = """

def \_\_init\_\_(self,a,b):

self.a = a

self.b = b

self.c = a \*\* b

def area(self):

return self.a \* self.b

"""

class\_dict = {}

class\_base = ()

exec(class\_body,globals(),class\_dict)

t = type(class\_name, class\_base, class\_dict)

t1 = t(10,20)

t1.area()

output: 200

Python executes the class body inside the class \_\_dict\_\_ dictionary

Bydefault, ‘type’ is the metaclass for all the classes. When we inherit type, then the inherited class will become the ‘custom meta class’ to create other classes. Class A(metaclass = MyMetaClass)

#custom metaclass

class MyMetaClass(type):

def \_\_new\_\_(mcls, name, bases, class\_dict): #mcls is MyMetaClass, name is the newclass

#tweak

new\_class = super().\_\_new\_\_(mcls,name,bases,class\_dict)

#tweak

return new\_class

class A(metaclass=MyMetaClass):

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

self.name = name

a = A("Jaison")

A.\_\_dict\_\_

a.\_\_dict\_\_

#class decorator // decorator functions decorate classes

def savings(cls): # class is the argument

cls.savings = "monthly"

return cls

@savings

class Account:

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

self.name = name

Account.\_\_dict\_\_

#decorating a class. decorator takes a parameter

def decor\_a\_class(apr):

def inner(cls):

cls.savings = "yearly"

cls.interest = apr

return cls

return inner

@decor\_a\_class(9)

class Account:

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

self.name = name

Account.\_\_dict\_\_

‘static’ and ‘class’ methods are descriptors, not functions (callable)

class A:

a = 10

b = 20

c = {"a":10,"b":20}

s = set([1,2,3])

def hello(self):

print("Hello")

for name, value in vars(A).items():

print(name, "value", value)

class decorator to decorate a function or a class

when a class decorated is written to decorate a function inside another class, \_\_get\_\_ has to be implemented.

\_\_prepare\_\_(name,bases,\*\*kwargs)->dict - of the metaclass creates the class dictionary. It’s a staticmethod

1. Python calls the \_\_prepare\_\_. 2) prepare returns a dictionary. 3) python manipulates the dictionary. 4) python calls the \_\_new\_\_

If \_\_prepare\_\_ is defined in a class, \_\_new\_\_ also has to be defined. Otherwise, if any arguments are passed in the inherited class, it cannot be stored in the class dictionary.

class MyMeta(type):

@staticmethod

def \_\_prepare\_\_(name, bases, \*\*kwargs):

print('MyMeta.\_\_prepare\_\_ called...')

print('\tname:', name)

print('\tkwargs:', kwargs)

return {'a': 100, 'b': 200}

def \_\_new\_\_(mcls, name, bases, cls\_dict, \*\*kwargs):

print('MyMeta.\_\_new\_\_ called...')

print('\tcls: ', mcls, type(mcls))

print('\tname:', name, type(name))

print('\tbases: ', bases, type(bases))

print('\tcls\_dict:', cls\_dict, type(cls\_dict))

print('\tkwargs:', kwargs)

return super().\_\_new\_\_(mcls, name, bases, cls\_dict)

class MyClass(metaclass=MyMeta, kw1=10, kw2=20):

pass

class CustomDict(dict):

def \_\_setitem\_\_(self, key, value):

print(f'Setting {key} = {value} in custom dictionary')

super().\_\_setitem\_\_(key, value)

def \_\_getitem\_\_(self, key):

print(f'Getting {key} from custom dictionary')

return int(super().\_\_getitem\_\_(key))

class MyMeta(type):

def \_\_prepare\_\_(name, bases):

return CustomDict()

def \_\_new\_\_(mcls, name, bases, cls\_dict):

print('metaclass \_\_new\_\_ called...')

print(f'\ttype(cls\_dict) = {type(cls\_dict)}')

print(f'\tcls\_dict={cls\_dict}')

class MyClass(metaclass=MyMeta):

pass

instance initializing steps: Type\_\_prepare\_\_ prepares the dictionary for the class. Python creates the bases, executed the code to populate the class dictionary. Type\_\_call\_\_ calls the Person\_\_new\_\_. Person returns the instance to the Type\_\_call\_\_. Type\_\_call\_\_ calls the Person\_\_init\_\_ for the new instance. New instance of Person is created.

The metaclass of ‘type’ is type itself.

type(type) is type.

Attribute accessing in python: \_\_getattribute\_\_(‘attribute), \_\_getattr\_\_ super().\_\_getattribute\_\_(‘attr’)

Attribute write accessor: \_\_setattr\_\_(self,name,value)

5th April 5, 2021

**Python closures**

A *closure* is a nested function which has access to a free variable from an enclosing function that has finished its execution. Three characteristics of a Python closure are:

* it is a nested function
* it has access to a free variable in outer scope
* it is returned from the enclosing function

A *free variable* is a variable that is not bound in the local scope. In order for closures to work with immutable variables such as numbers and strings, we have to use the nonlocal keyword.

Python closures help avoiding the usage of global values and provide some form of data hiding. They are used in Python decorators.

Sequence types are indexable. Mutable: lists, byte arrays. Immutable: strings, tuples, range, bytes.

Sets and dictionary are not indexable, so are not sequences.

Array module has an array object.

From collections package, nametuple and deque

homogeneous sequences: string. Heterogeneous: lists

All sequences are iterables, but all iterables are not sequences. (set is an iterable, but not a sequence)

No concatenation / repetition of range objects.

Concatenation of mutable objects (lists) causes the address to be copied.

Mutating an object means changing the state of an object without changing the address or creating a new object.

Constant folding is the process of recognizing and evaluating constant expression at the time of compiling rather than at computing them at run time.

**Tuples has pointers directly into its elements, where as lists has to access another array which has pointers to the list elements.**

In Shallow copy of the sequence, it copies the container, but not the elements of the container.

When we update the value of an immutable object of a shallow copied object, it creates a new memory reference for that value. (only when updates the element).

‘copy’ module has copy and deepcopy operations.

Slice object has a start and end also a step value. A = slice(0,4,2)

Custom sequences (classes) must implement \_\_len\_\_ and \_\_getitem\_\_ methods.

In-place concatenation is +=, repetition \*=. These are different from normal concatenation

In-place concatenation / repetition of mutable objects will not change the memory address of the mutable object. But immutable object does change the memory address.

Only concatenate similar sequence types

The sorted function uses TimSort algorithm

A list comprehension has 3 parts. Transformation, iteration and an optional filter.

Comprehensions are like functions and has its scope.

Nested loops inside the list comprehension and nested comprehension inside the comprehension.

Iterators are tools of consumable in nature (once used, can’t be reused), used to get an item out of a collection one by one.

Iterables are collections that implement iterators.

Iterator protocol implements \_\_iter\_\_ (just return the object itself) and \_\_next\_\_

Iterable objects implement only the \_\_iter\_\_ protocol. It returns a new instance of the iterator object used to iterate over the iterable.

Sequence protocol only necessarily to implement \_\_getitem\_\_ and optionally, \_\_len\_\_

If a class implements both iterator and sequence protocols, python uses the iterator protocol.

‘its easier to ask for forgiveness that to ask for permission’ – Grace Hoper

The iter function has a 2nd form. Iter(callable, sentinel)

StopIteration is the iter exception

Lazy evaluation: <https://www.udemy.com/course/python-3-deep-dive-part-2/learn/lecture/10059714#overview>

Generators are a type of iterators too. They implement the iterator protocol.

Generator functions generate generators (returns generators)

Generators are lazy iterators

xcross