#Decorator factory function:

from functools import wraps  
def param\_out(price): #decorator factory  
 def outer(fn): #decorator  
 @wraps(fn)  
 def inner(\*args):  
 print(price)  
 output = fn(\*args)  
 return output  
 return inner  
 return outer  
  
@param\_out(365)  
def mul(a,b):  
 return a \* b  
  
result = mul(10,5)  
print(result)

output:

365

50

# Decorator class

class Decor:  
 def \_\_init\_\_(self, a, b):  
 self.a = a  
 self.b = b  
  
 def \_\_call\_\_(self, fn):  
  
 def inner(\*args, \*\*kwargs):

print(self.a, self.b)  
 return fn(\*args, \*\*kwargs)  
  
 return inner  
  
  
@Decor(10, 20)  
def calc(a, b):  
 return a + b  
  
result = calc(1,2)  
print(result)

output:

10 20

3

# Decorating a class with a function

def dec\_class(cls):  
 cls.speak = lambda self, message: f'{self.\_\_class\_\_.\_\_name\_\_} says: {message}'  
 return cls  
  
  
class Person:  
 pass  
  
  
dec\_class(Person)  
p = Person()  
c = p.speak('Hello world')  
print(c)

output:

Person says: Hello world

# Decorating a class the ‘decorator’ way

def dec\_class(cls):  
 cls.speak = lambda self, message: f'{self.\_\_class\_\_.\_\_name\_\_} says: {message}'  
 return cls  
  
  
@dec\_class  
class Person:  
 pass  
  
  
p = Person()  
c = p.speak('Hello world')  
print(c)

output: Person says: Hello world

# Singledispatch written manually. // from functools import singledispatch. It has register and dispatch functions.

def switcher(fn):  
 registry = {}  
 registry['default'] = fn  
  
 def register(vaue):  
 def inner(fn):  
 registry[vaue] = fn  
 return fn  
 return inner

def decorator(value):  
 fn = registry.get(value, registry['default'])  
 return fn()  
  
 decorator.register = register  
 return decorator  
  
@switcher  
def month():  
 return 'no month exists'  
@month.register(1)  
def jan():  
 return ' January'  
month.register(2) (lambda : 'February')  
month.register(3) (lambda : 'March')  
month.register(4) (lambda : 'April')  
month.register(5) (lambda : 'May')  
month.register(6) (lambda : 'June')  
month.register(7) (lambda : 'July')  
month.register(8) (lambda : 'August')  
month.register(9) (lambda : 'September')  
month.register(10) (lambda : 'October')  
month.register(11) (lambda : 'November')  
month.register(12) (lambda : 'December')  
  
result = month(9)  
print(result)

output: September

python follows stable sort – if two values are equal in sort, it maintains the state of the list where the items are in.

Sorting using builtins sorted: sorted requires an iterable to be passed

employees = {"Jaison":45, 'Swamy':42,"Yasha":43,"Surya":24,"Raju":32,'Girisha':40,"Nag":40,"Rose":40,"Vij":36}  
  
result = sorted(employees, key=lambda e:employees[e], reverse=True) #key takes the value of the items in the dictionary to sort  
print(result)

output: ['Jaison', 'Yasha', 'Swamy', 'Girisha', 'Nag', 'Rose', 'Vij', 'Raju', 'Surya']

# randomize a list:

from random import random  
employees = {"Jaison":45, 'Swamy':42,"Yasha":43,"Surya":24,"Raju":32,'Girisha':40,"Nag":40,"Rose":40,"Vij":36}  
  
result = sorted(employees, key=lambda e: random())  
print(result)

output: ['Vij', 'Jaison', 'Surya', 'Swamy', 'Rose', 'Girisha', 'Nag', 'Yasha', 'Raju']

# using builtins dir function to introspect a function.

def func(a,b,c, \*, d=10,e=20):  
 m='accd'  
  
print(dir(func.\_\_code\_\_))  
print(func.\_\_code\_\_.co\_varnames)

output: ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'co\_argcount', 'co\_cellvars', 'co\_code', 'co\_consts', 'co\_filename', 'co\_firstlineno', 'co\_flags', 'co\_freevars', 'co\_kwonlyargcount', 'co\_lnotab', 'co\_name', 'co\_names', 'co\_nlocals', 'co\_stacksize', 'co\_varnames']

('a', 'b', 'c', 'd', 'e', 'm')

An attribute, that is callable is a method

Inspect.getcomments(myfunc) – returns the ‘#’ comments given preceding the function definition.

Inspect.getsource, getmoudule, signature

def func(a:int,c:'string obhect',b:str='abcd',m=10,\*, d:'defaults values',e=20):  
 m='accd'  
  
import inspect  
a = inspect.getsource(inspect.getsource)  
b = inspect.getmodule(inspect.getsource)  
print(b)  
  
for param in inspect.signature(func).parameters.values():  
 print(param.name)  
 print(param.default)  
 print(param.annotation)  
 print(param.kind)

Higher order functions are those takes a function as a parameter and returns a function

#builtins map

import inspect  
  
l1 = [1,2,3]  
l2=[4,5,6]  
  
l3 = map(lambda x,y:x+y, l1,l2)  
print(\*l3)

output: 5 7 9

#filter – filter out the iterable based on the function

#filter takes only 1 argument as iterable, returns an iterator  
l1 = [1,2,3,4,5]  
  
a = filter(lambda x: x % 2 == 0, l1)  
print(\*a)

Output: 2 4

#zip not a higher order function (does not take a function)

#zip is not a higherorder function. it takes a list of iterables and return an iterator  
  
l1 = [1,2,3]  
l2 = [10,20,'k']  
a = zip(l1,l2)  
print(\*a)

Output: (1, 10) (2, 20) (3, 'k')

#list comprehension

#list comprehension - returns a list, NOT an iterator  
l1 = [1,2,3]  
  
a = [x\*\*2 for x in l1]  
print(a)

Output: [1, 4, 9]

#list comprehension - returns a list, NOT an iterator  
l1 = [1,2,3]  
l2 = [10,22,33]  
  
a = [x+y for x, y in zip(l1,l2)] # adding the iterables  
  
print(a)

Output: [11, 24, 36]

#list comprehension - returns a list, NOT an iterator  
l1 = [1,2,3,4,5,6,7,10,11]  
a = [x for x in l1 if x % 2 == 0] # filtering  
print(a)

Output: [2, 4, 6, 10]

A reducing function is also called aggregators, folding functions or accumulators. It recombine an iterable recursively ending up in a single value.

# reduce function to find minimum value from a list  
l = [10,20,30,12,24,2,55,33]  
from functools import reduce  
result = reduce(lambda x, y: x if x < y else y, l)  
print(result)

Output:2

# reduce function to find highest value from a list  
l = [10,20,30,12,24,2,55,33]  
from functools import reduce  
result = reduce(lambda x, y: x if x > y else y, l)  
print(result)

Ouput: 55

#Partial function

#partial function - passes partial parameters to functions  
from functools import partial  
def add(a,b,c=100):  
 return a+b+c  
  
c = partial(add, b=10)  
d = c(5)  
print(d)

Ouput: 115

#operator module

from operator import mul  
  
a=10  
c = a.\_\_mul\_\_(10)  
d = a.\_\_rmul\_\_(5)  
e = mul(c,d)  
print(c)  
print(d)  
print(e)

output: 100

50

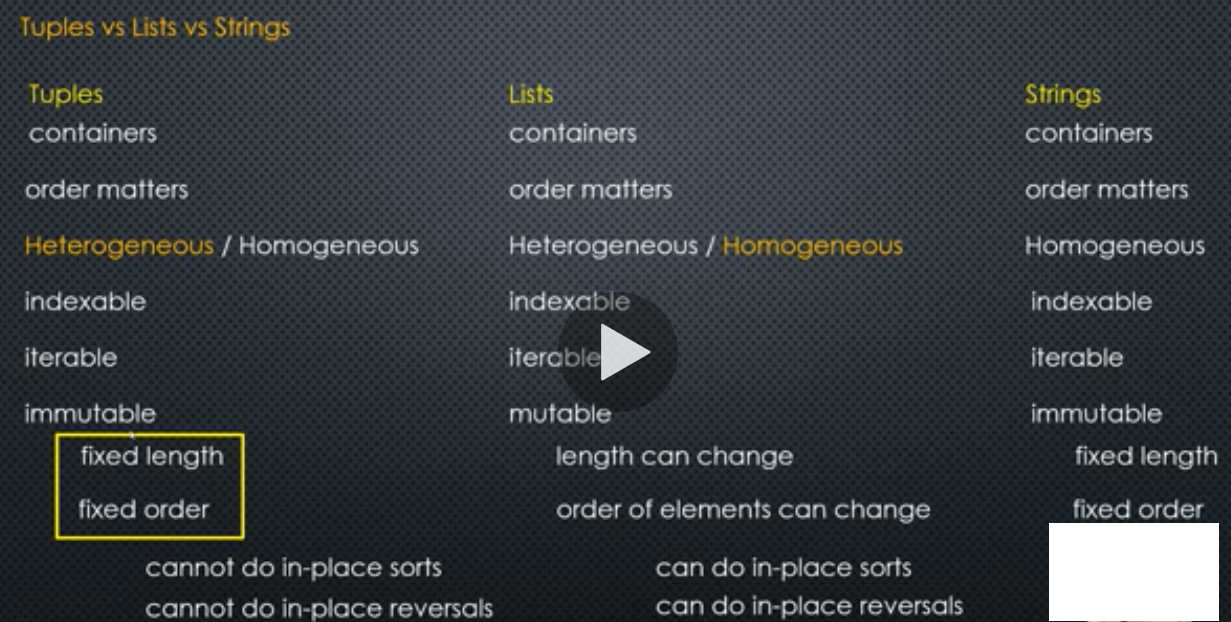
5000

Itemgetter returns a callable

from operator import itemgetter  
s = 1,2,3,45,44,4  
f=itemgetter(4,3,2)  
r=f(s)  
print(r)

ouptu: (44, 45, 3)

methodcaller – calls another callable. Methodcaller(‘upper’)(‘phthon’) – output: PYTHON



Namedtuple is function it generates a class. So it’s a class factory. From collections import namedtuple . the generated class is a type of tuple.

Namedtuple class instance is immutable. Class.\_fields gives the field names of the class. Class.\_source gives source code that generated the class.

Instance method – instance.\_asdict() returns a dictionary of key-value pairs.

Slicing an object returns a tuple. Unpacking returns a list.

Class.\_make(single\_iterable) – generates a new tuple object.

Instance method \_replace[keyword=value] – replaces the filed valued and returns a new instance.

Extend a namedtuple by using new\_fields = class.\_fields + (‘new\_field’) // then NewClass = namedtuple(‘NewClass’, new\_fields)

Default values can be passed to a namedtuple in 2 ways. One by defining a prototype instance of the class initiated with default keyword arguments and the using that prototype.\_replace(newkeyword\_values) to create new instance. 2nd method is to use class.\_\_new\_\_.\_\_defaults\_\_ = (kwyword\_values) and then create the object.

Random() generates a float, randint() generates int, choice(list) generates random items based on that list. Shuffle, randrange, seed

Default seed is system time. Choices() takes more than 1 parameter.

from collections import namedtuple  
  
mydict = dict(red=100, green=150, yellow=175, orange=200)  
#using a dictionary to create namedtuple  
Rainbow = namedtuple('Rainbow', mydict.keys())  
# r = Rainbow(\*mydict.values())  
r = Rainbow(\*\*mydict)  
print(r)  
# --------  
Color = namedtuple('Color', 'red, green, yellow, orange')  
Color.\_\_new\_\_.\_\_defaults\_\_ = (0,) \* len(mydict) # setting default values  
c = Color()  
'''  
# replacing a default value in the tuple class  
\*a,\_ = c  
a1 = Color(\*a,111)  
print(a1)  
#replacing a value in an instance  
a2 = a1.\_replace(yellow=444)  
print(a2)  
'''  
# creating new default values for the particular object only  
new\_value = 1, 2, 3, 4  
c1 = Color.\_make(new\_value)  
print(c1)  
  
# getting the values out of the namedtuple as dictionary  
d = c1.\_asdict()  
print(d)

ouput: Rainbow(red=100, green=150, yellow=175, orange=200)

Color(red=1, green=2, yellow=3, orange=4)

OrderedDict([('red', 1), ('green', 2), ('yellow', 3), ('orange', 4)])

# Random

#using random  
from random import random, randint, randrange, choice, choices, seed  
# seed('abc') # if the seed is set, the random numbers will not change  
print(random()) # from 0 to 1 range  
print(randint(10, 20)) # from within the given range  
l = list(range(100))  
c = choice(l) # generate random number from list  
c1 = [choice(l) for i in range(5)] # generates 5 random numbers  
c2 = choices(l, k=3) # genrates 3 random numbers of length specified in key:k  
print((c2))

Ouput: 0.9769134916477327

15

[57, 88, 99]

#using random  
from random import choices  
  
# choices using weight and cumulative weight to generate values based on weightage  
l = 'a','b','c'  
weight = [8,1,1]  
cum\_weight = [2,3,3]  
result = choices(l, k=5, weights=weight)  
cum\_result = choices(l, k=10, cum\_weights=cum\_weight)  
print(result)  
print(cum\_result)

Output: ['a', 'a', 'c', 'a', 'a']

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

Methods supported by list. Clear, remove(x), pop(i), append(x), insert(I,x), extend(iterable), s.copy(), reverse()

# slice

#slice is a type  
a= [10,20,30,40]  
b= slice(0,2)  
print(a[b])

Output: [10, 20]

The custom sequence should return the element given an index. Should know the length of the sequence. Should be able to loop over the sequence.

Should implement \_\_getitem\_\_, optionally \_\_len\_\_. It has to raise IndexError when reached out of bounds.

‘list’ by default implements \_\_getitem\_\_.

# Inplace sort on list with sort mutates the list.

# sort methods in list is inplace sort  
lst = ["brids","are", "beaurtiful","creatures"]  
lst.sort()  
print(lst)

Output: ['are', 'beaurtiful', 'brids', 'creatures']

#timeit

from timeit import timeit  
  
l = [10,20,30,50,40,55,23]  
  
t = timeit(stmt='l.sort()', globals=globals(), number=10)  
print(t)  
print(l)

ouput: 1.0400000000000686e-05

[10, 20, 23, 30, 40, 50, 55]

For sorting on custom objects, implement either \_\_gt\_\_ or \_\_lt\_\_ taking an int value.

# Custom sorting  
class MyClass:  
 def \_\_init\_\_(self, name, val):  
 self.val = val  
 self.name = name  
  
 def \_\_repr\_\_(self):  
 return f'object has {self.name} and {self.val}'  
 def \_\_lt\_\_(self, other):  
 return self.val < other.val  
  
c1 = ('c1', 10)  
c2 = ('c2', 10)  
c3 = ('c3', 20)  
c4 = ('c3', 15)  
c5 = ('c4', 17)  
  
l = [c1,c2,c3,c4,c5]  
r = sorted(l)  
print(r)

Output: [('c1', 10), ('c2', 10), ('c3', 15), ('c3', 20), ('c4', 17)]

Comprehensions has their own local scope like in a function.

Nested comprehensions and nested loops in comprehensions are different.

Compile(‘code to compile’, filename=’filename’, mode=’eval’)

# assert raises Assertion error if the expression evaluates to False.

assert 1 == 10, ‘custom eror msg’

AssertionError custom eror msg

The \_\_iter\_\_ returns the object itself. Return self. While running the iterator through for loop or by while loop, it first calls the sq\_iter = iter(sq) once, then the next(sq\_iter).

Iterable is an object that implements \_\_iter\_\_, which returns the iterator class(self).

# iterable and iterator

# iterable is the container class and iterator is another class  
  
class Cities:  
 def \_\_init\_\_(self, cities):  
 self.\_cities = cities  
  
 def \_\_iter\_\_(self):  
 return CitiesIterator(self)  
  
 def \_\_len\_\_(self):  
 return len(self.\_cities)  
  
  
class CitiesIterator:  
 def \_\_init\_\_(self, city\_obj):  
 self.\_city\_obj = city\_obj  
 self.index = 0  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def \_\_next\_\_(self):  
 if self.index >= len(self.\_city\_obj):  
 raise StopIteration  
 result = self.\_city\_obj.\_cities[self.index]  
 self.index += 1  
 return result  
  
  
world\_cities = ["New York", 'London', 'Tokyo', 'Shanghai', 'California', 'Mumbai']  
  
cities = Cities(world\_cities)  
for city in cities:  
 print(city)

Output: New York

London

Tokyo

Shanghai

California

Mumbai

If both sequence protocol \_\_getitem\_\_ and iterator protocols are defined in a class, python will run it as an iterator.

“**constant folding**“. It just means that when we have a **constant** expression **Python** evaluates the value of that expression at compile time so that when you actually run the program it doesn't take as long to run because **Python** uses the already computed value.

2nd form of iter(callable, sentinel).

Deletating an iterator 150?

Reverse iteration using reversed. \_\_reversed\_\_. For item in reversed(seq)

# iter

def count\_down(num):  
 def down():  
 nonlocal num  
 num -= 1  
 return num  
 return down  
  
c = count\_down(10)  
  
'''for \_ in range(10):  
 print(c())  
'''  
k = count\_down(10)  
m = iter(k,3) # passing an iterator  
  
for i in m:  
 print(i, end=' ')

output:

9 8 7 6 5 4

# iter delegation

from collections import namedtuple  
  
Person = namedtuple('Person', 'first last')  
  
person\_names = [Person("jaison", 'jacob'), Person("sonia", 'jaison'), Person("enil", 'jaiosn'),  
 Person("esther", 'jaison')]  
  
  
class PersonNames:  
 def \_\_init\_\_(self, namel\_lst):  
 self.p\_names = [name.first.capitalize() + ' ' + name.last.capitalize() for name in namel\_lst]  
  
 def \_\_iter\_\_(self):  
 return iter(self.p\_names) # Delegation - an iterator is returned for the list  
  
p = PersonNames(person\_names)  
print(\*p)  
for n in p:  
 print(n)

ouput:

Jaison Jacob Sonia Jaison Enil Jaiosn Esther Jaison

Jaison Jacob

Sonia Jaison

Enil Jaiosn

Esther Jaison

Generators are iterators aswell. Generator functions are are generator factories, which returns a generator when called.

Generator expressions use comprehension syntax

#Generator yield

# yield  
def song():  
 print("first line of song '' rarara")  
 print("2nd lind of song '''lalal")  
 yield 'song stopper...a'  
 print("resumed .. 'sasasa")  
 yield 'song pauser again'  
 print('kalalka')  
  
line = song()  
  
a = next(line)  
print(a)  
b = next(line)  
print(b)  
#next(line) # this will signal a stopiteration  
try:  
 next(line)  
except StopIteration:  
 print('no more yields')

Output:

first line of song '' rarara

2nd lind of song '''lalal

song stopper...a

resumed .. 'sasasa

song pauser again

kalalka

no more yields

when the generator function return ‘None’, it raises a StopIteration exception.

Generators implement iterator protocol

#yield  
import math  
  
def fact(num):  
 for i in range(num):  
 yield math.factorial(i)  
  
f = fact(10)  
  
for i in f:  
 print(i)

Ouput:

1

1

2

6

24

120

720

5040

40320

362880

# generator

def squares(n):  
 for i in range(n):  
 yield i \*\* 2  
  
class Squares:  
 def \_\_init\_\_(self, n):  
 self.n = n  
  
 def \_\_iter\_\_(self):  
 return squares(self.n) # returning a generator / iterator  
  
sq = Squares(5)  
print(list(sq))

output:

[0, 1, 4, 9, 16]

def squares(num):  
 for i in range(num):  
 yield i \*\* 2  
  
sq = squares(5)  
print(next(sq))  
print(next(sq))  
s = enumerate(sq) # enuerate continues from where next stopped  
for i in s:  
 print(i)

output:

0

1

(0, 4)

(1, 9)

(2, 16)

A generator expression or comprehension returns an iterator. A list comprehension returns an iterable.

a = (i \*\* 2 for i in range(10))  
  
for i in a:  
 print(I, end=’ ‘)

output: 0 1 4 9 16 25 36 49 64 81

# reading from files

*'''  
with open('file1.txt') as f:  
 r = f.readlines()  
 for row in r:  
 s = row.strip('\n')  
 print(s)  
  
files = 'file1.txt','file2.txt'  
def brand(\*files):  
 for file in files:  
 with open(file) as f:  
 for line in f:  
 yield line.strip('\n')  
   
s = brand(\*files)  
  
for a in s:  
 print(a)  
'''*files = 'file1.txt','file2.txt'  
  
def brand(\*files):  
 for file in files:  
 yield from cleaned\_up(file)  
  
def cleaned\_up(file):  
 with open(file) as f:  
 for row in f:  
 yield row.strip('\n')  
  
  
print(list(brand(\*files)))

#yield from

def subgen():

for i in range(3):

yield i

def delegator():

print('calling subgen')

yield from subgen()

#s = subgen()

'''for i in s:

yield i

'''

#yield from s

yield 'subgen closed'

d = delegator()

next(d)

next(d)

---

A predicate is a function that takes a single argument and returns True or False.

When working with infinite iterator tolls, use islice to limit.

#islice

l = 1,2,3,4,5,6,7

from itertools import islice

b= islice(l,2,7,2) #iterable, start, stop, step

list(b)

output: [3, 5, 7]

In [ ]:

#filter

def gen\_cube(num):

for i in range(num):

yield i \*\* 3

a = gen\_cube(10)

from itertools import filterfalse, takewhile, dropwhile

def gen\_odd(num):

if not num % 2 == 0:

return num

c = filter(gen\_odd, gen\_cube(5)) # function, iterator

list(c)

output: 1, 27

--

Dropwhile is just opposite of takewhile

#cycle

from itertools import islice, cycle

def color():

yield 'red'

yield 'green'

yield 'oange'

yield 'white'

c = color()

c1 = cycle(c)

list(islice(c1,6)) # cycle cycle through until it reaches the limit set by islice

output:

['red', 'green', 'oange', 'white', 'red', 'green']

In [ ]:

#chain

from itertools import islice, cycle, tee, chain, repeat

def color():

yield 'red'

yield 'green'

yield 'oange'

yield 'white'

c = color()

l =1,2,3,4

k = chain(l,color())

list(k)

output: [1, 2, 3, 4, 'red', 'green', 'oange', 'white']

In [ ]:

#tee

from itertools import islice, cycle, tee, chain, repeat

def color():

yield 'red'

yield 'green'

yield 'oange'

yield 'white'

c = color()

l =1,2,3,4

k = tee(l,3)

for i in k:

print(\*i)

output: 1 2 3 4

1 2 3 4

1 2 3 4

#chain.from\_iterable

from itertools import islice, cycle, tee, chain

a = [1,2,3]

b = 'a','b','c'

c = "first", "second", "third"

d = a,b,c

e = chain.from\_iterable(d)

d

output: ([1, 2, 3], ('a', 'b', 'c'), ('first', 'second', 'third'))

In [ ]:

--

#starmap

from itertools import starmap

l = [[2,4],[3,6]]

a = starmap(lambda x,y: x \* y, l)

list(a)

output: [8, 18]

In [ ]:

If there is no iter on an iterable, can’t call next on it.

l = [1,2,3,4]

a = iter(l)

next(a)

output: 1

--

#groupby

data = ((1,"john"), (1, "Rema"), (2,"Rohan"), (2,"Soman"),(2,"Sujoy"),(3,"Ruk"),(3,"Sujoy"),(3,"Rakma"),(3,"Jinger"))

from itertools import groupby

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

for group\_key, sub\_iter in a:

print(group\_key, list(sub\_iter))

output: 1 [(1, 'john'), (1, 'Rema')]

2 [(2, 'Rohan'), (2, 'Soman'), (2, 'Sujoy')]

3 [(3, 'Ruk'), (3, 'Sujoy'), (3, 'Rakma'), (3, 'Jinger')]

In [ ]:

#contextmanager decorator

from contextlib import contextmanager

@contextmanager

def open\_file(fname, mode):

f = open(fname, mode)

try:

yield f

finally:

f.close()

with open\_file('jjfile.txt','r') as fl:

r = fl.readlines()

print(r)

print(fl.closed)

output: ['Python is a beautiful language Python is a powerful language\n', 'Python is charming']

True

When a GeneratorExit occurs, either should return or reraise the exception when the close() call on the generator is made. Simply cannot ignore it. If ignored, Python will raise RuntimeError.

If there are no more yield in a generator function, priming will cause StopIteration exception.

Exception can be thrown at generator function using .throw(exceptiontype)

#generator

from inspect import getgeneratorstate

def gen():

print("starting gen..")

count = 0

while True:

try:

greet = yield 'hello',count

count += 1

print(greet)

except (GeneratorExit, ValueError) as ex:

print(ex)

g = gen()

a = next(g)

print(a)

#g.throw(ValueError, 'my msg')

#g.close() # closing raises GeneratorExit error at the gen func

Output: starting gen..

('hello', 0)

Exceptions are not necessarily errors. That can be used to control the flow of execution. Ex. Throwing exceptions at a generator to mimic database commit, rollback and abort.

No coroutine decorator is built into python. Need to write our own.

#Yield from delegator

def subgen():

yield "hello"

yield "friends"

def delegator():

yield from subgen()

yield "finished"

g = delegator()

for i in range(3):

print(next(g))

output: hello

friends

finished

#decorator and yieldfrom

def coro():

output = " "

while True:

received = yield output

output = received[::-1]

def decor(gen):

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

g = gen()

next(g)

return g

return inner

@decor

def gen1():

yield from gen2()

def gen2():

yield from gen3()

def gen1():

yield from coro()

sg = gen1()

next(sg)

sg.send("coorona")

output: 'anorooc'

In [ ]:

--

Counter is Pythons implementation of multi-set.

The hash value of the key decides the ‘index of the dictionary key’ by the hash function.

2 conditions for a key to be hashable in a dictionary. A) it should return an integer. B) if two objects compare equal (==), then their hashes should also be equal.

#itemgetter

"""

Return a callable object that fetches item from its operand using the operand’s \_\_getitem\_\_() method. If multiple items are specified, returns a tuple of lookup values.

"""

from operator import itemgetter

l = []

l.append([1,2,3])

l.append([4,5,6])

l.append((4,5))

f = itemgetter(2)

f(l)

output: (4, 5)

In [ ]:

Common dictionary operations. D.get(key, default), d.clear(), d.pop(), d.popitem(), del d[key], d[key]= value, d.copy(), d.setdefault(key,value), d.update(d1)

#fromkeys

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

d = dict()

x = d.fromkeys(l,'N/A')

x

output: {1: 'N/A', 2: 'N/A', 3: 'N/A', 4: 'N/A', 5: 'N/A'}

In [ ]:

#update a dictionary

d = dict()

d={1:'a',2:'b'}

d.update({'a':10}) #add

d.update({1:10}) # modify

d

output: {1: 10, 2: 'b', 'a': 10}

In custom classes, if two objects compare equal (==) then the hash of the two objects also should be equal.

#custom class dictionary key

class Person:

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

self.id = id

self.name = name

self.age = age

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

if isinstance(other, Person):

return self.id == other.id

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

return hash(self.id)

p = Person('a', 'Jaison', 29)

d = {Person('b','Jacob',38): 'one',p:'two',p:'three'}

d[Person('b','Jacob',38)]

output: 'one'

In [ ]:

Sets are distinct, unordered collection of items. Set elements must be hashable.(lists, set and dictionaries are not hashable). Sets are implemented as hashmaps. A set is a mutable collection, so cannot be used as a key in a dictionary. A set cannot be an element of another set.

Tuple is an immutable equivalent of a list. Froenset is an immutable equivalent of a set. No literal for creating a frozen set. Frozenset({1,2,3})

Set operations: union |, intersection &, difference -, symmetric\_difference ^.

Disjoint = if two sets does not have a common element, then the sets are disjoint. Len(set1 & set2) == 0

Containment: issubset <=, ispropersubset <, issuperset >, ispropersuperset >=.

To unpack an ‘iterable’, use single star \*. To unpack keys in a dictionary, use single \*. To unpack a dictionary completely, use \*\*. Sets cannot be created using an empty literal. {}

a = {1,2,3}

b = set(['a','b','c'])# argument must be an iterable

c = {\*a,\*b}

type(c)

output: set

--

#set comprehension

s = {c for c in 'Python'} # use curly braces for a set

s

output: {'P', 'h', 'n', 'o', 't', 'y'}

cardinality of a set is the number of elements in the set. i.e, len.

To remove an element from a set, use s.remove(‘z’). KeyError if key is not present. To avoid exception, use s.discard(‘z’).

To add an element, use s.add(‘z’). s.pop, s.clear. All these functions mutates the set.

Set update operations: |= s1.update(), &= s1.intersection\_update(), -= difference\_update(), ^= symmetric\_difference\_update()

----

Serialization is the creation of the persistent representation of the object. Deserialization is the reconstruction of the object from the serialized data.

Python dictionaries are objects, json dictionaries are string. There is no order of keys in a json string.

Set is not serializable in JSON.

#encode

from json import JSONEncoder, dumps

custom\_encoder = JSONEncoder()

d = {'a':float('nan'),'b':float('inf')}

c = custom\_encoder.encode(d)

s = json.loads(c)

c, s

output: ('{"a": NaN, "b": Infinity}', {'a': nan, 'b': inf})

--

Json dumps arguments: json.dumps(default, skipkeys=False, separators=False, Indent=False, allow\_nan=False, sort\_keys=False, cls=False). Separator has 2 arguments; between items and between key-value pairs.

Schema is a structure for serializing and deserializing json data.

Json.load, loads(object\_hook=func) – object\_hook works deserialization of objects for python dictionaries.

Loads(object\_hook=func, [object\_pairs\_hook=func // normally to pass the objects as a tuple to keep ordering, parse\_int, parse\_float, parse\_constant) both object\_hook and object\_pairs\_hook cannot be passed at a time. If passed, object\_paris\_hook takes perecedence.

Object\_hook function receives the parsed object.

#time

import pytz

from pytz import timezone

from datetime import datetime

local\_t = datetime.now() #localtime

utc\_t = datetime.utcnow() #utctime

timezone\_t = datetime.now().astimezone(timezone('Asia/Kolkata'))#timezone offset

t\_zone = timezone('Europe/London') # timezone set

t\_time = datetime.now(t\_zone)

in JSONDecoder, override the decode function.

#object serialization with date

import json

from datetime import datetime

d = {'date\_object':datetime.now(), "name":"Jaiosn", "sex":"Male", 'age':45}

def convert\_date(arg):

return arg.strftime('%Y-%m-%d %H:%M:%S')#onvert date to JSON string

s = json.dumps(d, default=convert\_date)

s

output: '{"date\_object": "2020-05-17 10:31:17", "name": "Jaiosn", "sex": "Male", "age": 45}'

#convert JSON string to datetime

import json

from datetime import datetime

def convert\_str\_to\_date(arg):

if "object\_type" in arg and arg['object\_type'] == 'datetime':

return datetime.strptime(arg['date'], "%Y-%m-%dT%H:%M:%S")

else:

return arg # this is very important

j = """

{

"name":"Jaison",

"age":45,

"DOB":{"object\_type":"datetime", "date":"1975-3-23T06:15:45"}

}

"""

l = json.loads(j, object\_hook=convert\_str\_to\_date)

l

output: {'name': 'Jaison', 'age': 45, 'DOB': datetime.datetime(1975, 3, 23, 6, 15, 45)}

standard library is mostly written in python and sometimes in c. built-in library is written in c. cpython.

# module namespaces

import pytz, sys

a = id(sys.modules['pytz'])

b = globals()['pytz']

pytz.\_\_dict\_\_['\_\_version\_\_']

dir(pytz)

--

Python installed @ and c binaries located: sys.exec\_prefix, sys.prefix

Python imports look @ sys.path

Importlib.import\_module[‘math’]

Sys.meta\_path contains the finder objects

Marshmallow and pyaml does both serialization and deserialization.

Defaultdict(callable,[optional kwargs]) – callable returns a value when a the value of a non existing key is requested.

Dict.get(key, value) does not create a key if it doesn’t exist. But defaultdict does it.

Ordereddict has a function as reversed. An ordereddict and a normal dict is equal if both contains same key-value pairs irrespective of the order.

Multisets can have repeated elements.

In counter, fromkeys is not implemented.

#normal dict and defaultdict counter

from collections import defaultdict

a = "The last dew drops in the Martian soil was beyond billions of Sun years"

d = dict()

for c in a:

d[c] = d.get(c,0) + 1

d

#-------

d = defaultdict(int)

for c in a:

d[c] += 1

d

unary operation ( + / - ) with Counter, will remove the opposte signed items from the Counter.

--

#format:

"class Person:\n",

" def \_\_init\_\_(self, name, dob):\n",

" self.name = name\n",

" self.dob = dob\n",

" \n",

" def \_\_repr\_\_(self):\n",

" print('\_\_repr\_\_ called...')\n",

" return f'Person(name={self.name}, dob={self.dob.isoformat()})'\n",

" \n",

" def \_\_str\_\_(self):\n",

" print('\_\_str\_\_ called...')\n",

" return f'Person({self.name})'\n",

" \n",

" def \_\_format\_\_(self, date\_format\_spec):\n",

" print(f'\_\_format\_\_ called with {repr(date\_format\_spec)}...')\n",

" dob = format(self.dob, date\_format\_spec)\n",

" return f'Person(name={self.name}, dob={dob})'"

---------

Slots will not affect the creation of class dictionary. Only instance dictionary will not be created if slots are defined.

class A:

\_\_slots\_\_ = ('name', '\_longitude', '\_latitude')

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

self.name = name,

self.\_longitude = longi

self.\_latitude = lati

@property

def longitude(self):

return self.\_longitude

@property

def latitude(self):

return self.\_latitude

@property

def latitude():

del self.\_latitude

a = A('Mumbai',100,200)

a.\_longitude

del a.\_latitude

properties and \_\_slots\_\_ are essentially data descriptors. They implement the descriptor protocols. (\_\_get\_\_, \_\_set\_\_, \_\_delete\_\_, \_\_set\_name\_\_)

declare \_\_slots\_\_ = ‘\_\_dict\_\_’, ‘other\_slots’ if wanted to get an instance dictionary for an instance object along with the slots.

#Descriptor

class A:

def \_\_set\_name\_\_(self,owner\_class,prop\_name):

print(f'setname called, owner class = {owner\_class}, property\_name = {prop\_name}')

self.prop\_name = prop\_name # important to set this

def \_\_set\_\_(self, instance, value):

print("setting value")

instance.\_\_dict\_\_[self.prop\_name] = value

def \_\_get\_\_(self,instance,owner\_class):

print("getting value")

return instance.\_\_dict\_\_.get(self.prop\_name, None)

class B:

\_\_slots\_\_ = '\_\_dict\_\_' # if the class implements slots, it has to add dict in order to add properties

name = A()

b = B()

b.name = 'Jaiosn'

b.name

decorator, descriptor, metaclasses comes under metaprogramming.

A class inherits from ‘type’ and its \_\_new\_\_ returns an object from super().\_\_new\_\_(cls,class\_name, base\_name, dict\_name)

Staticmethods and classmethods are descriptors. The are not callables, they are descriptors. Callable(staticmethod) = false.

#meta class

class MyMetaClass(type):

def \_\_new\_\_(mcls, name, base, dict):

obj = super().\_\_new\_\_(mcls,name,base,dict)

return obj

class MyClass(metaclass=MyMetaClass):

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

self.x = x

self.y = y

m = MyClass(10,20)

m.x

--

Class type defines the \_\_prepre\_\_ method.

1. Python calls \_\_prepare\_\_, prepare returns \_\_dict\_\_. Python injects certain things to \_\_dict\_\_, calls \_\_new\_\_.

A class is a callable, because its metaclass type implements \_\_call\_\_

Attribute access is a comibination of \_\_getattribute\_\_ and \_\_getattr\_\_ methods.

When the exception is defined as except: , the sys.exc\_info() will return exc\_type, exc\_value, exc\_traceback.

Exception object has 2 properties: \_\_traceback\_\_ and args. Traceback module has print\_tb and print\_exception methods.

xcross