YCROSS

Tutorialspoint.com – python q&a

improve thinking in programming logic

practice coding @ ◾ Codewars ◾ CodeFights ◾ HackerRank ◾ CodinGame

Is a kind of programming paradigm that is largely based on formal logic. Any program written in the logic programming language is a collection of sentences in logical form, which shows facts and rules about certain problem domains.

If we manage to develop a valid logic, we will be able to move forward in a flexible way through different languages, trying not to depend on the language.

Experts break the following steps to become good at programming logic thinking:

1. Learn programming language very well

Programming is about solving problems, a good technique is to break the big problem in small ones to focus on each problem in a better way, you can use pseudocodes in a program.

You can choose one of these: C++, Java, Python.

You can also choose between other languages, but many people use one of the above languages to do competitive programming or implement many useful applications.

Learn to store data efficiently.

Know standard data structures. C++, it is STL, they are collections in Java.

2. Learn standard algorithms

Learning about structures will provide you a better strategy to focus on your problems and have suitable software.

Searching, Sorting, Hashing, Merging, etc.

Introduction to Algorithms

3. Practice Practice Practice

The most important point is our experts suggest you is one and only one is practice. An algorithm is nothing more than an ordered and finite set of operations that learner does for the individual goal of finding a solution to a problem. So try to practice simpler problems to get better logic.

Practice competitive programming on online judges like:

Sphere Online Judge

Codeforces

4. Learn from experiences (Feedback)

In programming logic we have several ways to solve problems, possibly someone else has solved the problem that you have in an optimal and simple way. It is necessary to look at other people’s minds in order to move forward as a programmer. You have a lot of great projects to watch.

Learn from others

Learn from your mistakes

Always optimize your code for good performance

To improve logic thinking, take a few questions. Next, think about the different ways to solve each problem to solve those problems. Then, take the problem one by one. And then convert your idea into coding. But remember not to be satisfied with just one program for a problem. Try in the best way. Then, finally, compare all the methods.

----

Reference implementation

Python coding style is covered in PEP 8.

Python's name is derived from the British comedy group Monty Python, whom Python creator Guido van Rossum enjoyed while developing the language.

CPython is the reference implementation of Python. It is written in C, meeting the C89 standard with several select C99 features.[107] It compiles Python programs into an intermediate bytecode[108] which is then executed by its virtual machine.[109] CPython is distributed with a large standard library written in a mixture of C and native Python. It is available for many platforms, including Windows and most modern Unix-like systems. Platform portability was one of its earliest priorities.[110]

Other implementations

PyPi is a fast, compliant interpreter of Python 2.7 and 3.5.[111] Its just-in-time compiler brings a significant speed improvement over CPython but several libraries written in C cannot be used with it.[112][113]

Stackless Python is a significant fork of CPython that implements microthreads; it does not use the C memory stack, thus allowing massively concurrent programs. PyPi also has a stackless version.[114]

MicroPython and CircuitPython are Python 3 variants optimized for microcontrollers. This includes Lego Mindstorms EV3.[115]

RustPython is a Python 3 interpreter written in Rust.[116]

Cross-compilers to other languages

There are several compilers to high-level object languages, with either unrestricted Python, a restricted subset of Python, or a language similar to Python as the source language:

Jython enables the use of the Java class library from a Python program.

IronPython follows a similar approach in order to run Python programs on the .NET Common Language Runtime.

The RPython language can be compiled to C, and is used to build the PyPy interpreter of Python.

Pyjs compiles Python to JavaScript.

Cython compiles Python to C and C++.

Numba uses LLVM to compile Python to machine code.

Pythran compiles Python to C++.

Somewhat dated Pyrex (latest release in 2010) and Shed Skin (latest release in 2013) compile to C and C++ respectively.

Google's Grumpy compiles Python to Go.

MyHDL compiles Python to VHDL.

Nuitka compiles Python into C++.[119]

----

Pythonic way of coding // A protocol oriented language // Expert level code: that gives certain clarity as to when an where a python feature to be used.

1. Metaclasses implements lays constraints on user classes from library classes. 2) Python protocols are implemented using underscore methods and their top level functions. 3) Decorators 4) Generators 5) Context managers – sequential, \_\_init\_\_, \_\_class\_\_, \_\_enter\_\_, \_\_exit\_\_.

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time.

Module import time is equivalent to compile time in python. Python compiler is a very small one.

\_\_dumder\_\_ methods are also called data model methods <https://docs.python.org/3/reference/datamodel.html>

A top level function in Python is ‘+’, len … Underscore method protocol is \_\_add\_\_, \_\_len\_\_.

‘!r’ is an object repr() syntax. Dumdr methods use top level function / syntax to implement the protocol very often.

Core patterns in Python: 1) Protocol view of python, 2) Built-in inheritance protocol, 3) Object orientation of Python.

Vim: vsplit <filename> splits window

Correct = wrd == self.word

If correct ..

--

REST stands for REpresentational State Transfer and is an architectural style used in modern web development. It defines a set of rules/constraints for a web application to send and receive data.

<https://www.geeksforgeeks.org/python-build-a-rest-api-using-flask/>

A metaclass is a class whose instances are classes. Like an "ordinary" class defines the behavior of the instances of the class, a metaclass defines the behavior of classes and their instances.

Metaclasses are not supported by every object oriented programming language. Those programming language, which support metaclasses, considerably vary in way the implement them. Python is supporting them.

Some programmers see metaclasses in Python as "solutions waiting or looking for a problem".

There are numerous use cases for metaclasses. Just to name a few:

logging and profiling

interface checking

registering classes at creation time

automatically adding new methods

automatic property creation

proxies

automatic resource locking/synchronization.

--

Library.py

#Implementing user constraints using metaclasses

class BaseMeta(type):

def \_\_new\_\_(cls, name, bases, body):

if name != 'Base' and not 'bar' in body:

raise TypeError('bad user class')

return super().\_\_new\_\_(cls, name, bases, body)

class Base(metaclass=BaseMeta):

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

print('init\_subclass', args, kwargs)

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

def foo(self):

return self.bar()

def bar(self):

pass

#implementing user constraints using \_\_build\_class\_\_

'''old\_bc = \_\_build\_class\_\_'''

'''def my\_bc(fun, name, base=None, \*\*kw):

if base is Base:

print('Check if bar is defined')

if base is not None:

return old\_bc(fun,name,base,\*\*kw)

return old\_bc(fun, name, \*\*kw)'''

'''def my\_bc(\*a, \*\*kw):

print('objects',a,kw)

return old\_bc(\*a, \*kw)

import builtins

builtins.\_\_build\_class = my\_bc'''

class users.py

from Library import Base

assert hasattr(Base,'bar')

class Derived(Base):

def bar(self):

return 'bar'

closure object duality : Decorators in Python make an extensive use of closures as well. On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out. All function objects have a \_\_closure\_\_ attribute that returns a tuple of cell objects if it is a closure function.

Generator is a co-routine that interleaves between user code and library code

Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of frozen set remains the same after creation.

Due to this, frozen sets can be used as key in Dictionary or as element of another set. But like sets, it is not ordered (the elements can be set at any index).

The syntax of frozenset() method is:

Frozenset([iterable as initializer])

# random dictionary

person = {"name": "John", "age": 23, "sex": "male"}

fSet = frozenset(person)

print('The frozen set is:', fSet)

output: The frozen set is: frozenset({'name', 'sex', 'age'})

--

Setattr(Object, attribute, value) – can be used to set the values to attributes. Can use for loops.

Getattr(object, attribute) – can also be used sameway.

The generator stores a definition for (i=0; i<10; i+=1) and computes the next value only when needed (AKA lazy-evaluation).

# this is a generator, creates each x/2 value only when it is needed, uses ()

gen = (x/2 for x in range(5000000))

--

(2) As a function, using yield to return the next value:

# this is also a generator, it will run until a yield occurs, and return that result.

# on the next call it picks up where it left off and continues until a yield occurs...

def divby2(n):

num = 0

while num < n:

yield num/2

num += 1

# same as (x/2 for x in range(5000000))

print divby2(5000000)

--

#zip\_longest

import itertools

lst = [[1,2],[2,3,4],[3,4,['a','b','c','d']]]

m = list(itertools.zip\_longest([1,2],[2,3,4],[3,4],['a','b','c','d'], fillvalue='Hai'))

print(m)

output: [(1, 2, 3, 'a'), (2, 3, 4, 'b'), ('Hai', 4, 'Hai', 'c'), ('Hai', 'Hai', 'Hai', 'd')]

Uses Of Django –

Django is an open-source python web framework used for rapid development, pragmatic, maintainable, clean design, and secures websites. A web application framework is a toolkit of all components need for application development. The main goal of the Django framework is to allow developers to focus on components of the application that are new instead of spending time on already developed components. Django is fully featured than many other frameworks on the market. It takes care of a lot of hassle involved in the web development; enables users to focus on developing components needed for their application.

Django follows the MVT framework for architecture.

M stands for Model

V stands for View

T stands for Template

Exception: no catch block in try block

json.loads() takes in a string and returns a json object.

json.dumps() takes in a json object and returns a string.

GUI Python

Tkinter: 1) define the widgets, 2) pack it or grid it in the screen. Columns are relative to eachother

***Globals() from tkinter:*** {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x00C19E38>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_file\_\_': 'C:/Users/jaisojac/PycharmProjects/Fred1/A1.py', '\_\_cached\_\_': None, 'enum': <module 'enum' from 'C:\\Users\\jaisojac\\AppData\\Local\\Programs\\Python\\Python38-32\\lib\\enum.py'>, 'sys': <module 'sys' (built-in)>, 'TclError': <class '\_tkinter.TclError'>, 'constants': <module 'tkinter.constants' from 'C:\\Users\\jaisojac\\AppData\\Local\\Programs\\Python\\Python38-32\\lib\\tkinter\\constants.py'>, 'NO': 0, 'FALSE': 0, 'OFF': 0, 'YES': 1, 'TRUE': 1, 'ON': 1, 'N': 'n', 'S': 's', 'W': 'w', 'E': 'e', 'NW': 'nw', 'SW': 'sw', 'NE': 'ne', 'SE': 'se', 'NS': 'ns', 'EW': 'ew', 'NSEW': 'nsew', 'CENTER': 'center', 'NONE': 'none', 'X': 'x', 'Y': 'y', 'BOTH': 'both', 'LEFT': 'left', 'TOP': 'top', 'RIGHT': 'right', 'BOTTOM': 'bottom', 'RAISED': 'raised', 'SUNKEN': 'sunken', 'FLAT': 'flat', 'RIDGE': 'ridge', 'GROOVE': 'groove', 'SOLID': 'solid', 'HORIZONTAL': 'horizontal', 'VERTICAL': 'vertical', 'NUMERIC': 'numeric', 'CHAR': 'char', 'WORD': 'word', 'BASELINE': 'baseline', 'INSIDE': 'inside', 'OUTSIDE': 'outside', 'SEL': 'sel', 'SEL\_FIRST': 'sel.first', 'SEL\_LAST': 'sel.last', 'END': 'end', 'INSERT': 'insert', 'CURRENT': 'current', 'ANCHOR': 'anchor', 'ALL': 'all', 'NORMAL': 'normal', 'DISABLED': 'disabled', 'ACTIVE': 'active', 'HIDDEN': 'hidden', 'CASCADE': 'cascade', 'CHECKBUTTON': 'checkbutton', 'COMMAND': 'command', 'RADIOBUTTON': 'radiobutton', 'SEPARATOR': 'separator', 'SINGLE': 'single', 'BROWSE': 'browse', 'MULTIPLE': 'multiple', 'EXTENDED': 'extended', 'DOTBOX': 'dotbox', 'UNDERLINE': 'underline', 'PIESLICE': 'pieslice', 'CHORD': 'chord', 'ARC': 'arc', 'FIRST': 'first', 'LAST': 'last', 'BUTT': 'butt', 'PROJECTING': 'projecting', 'ROUND': 'round', 'BEVEL': 'bevel', 'MITER': 'miter', 'MOVETO': 'moveto', 'SCROLL': 'scroll', 'UNITS': 'units', 'PAGES': 'pages', 're': <module 're' from 'C:\\Users\\jaisojac\\AppData\\Local\\Programs\\Python\\Python38-32\\lib\\re.py'>, 'wantobjects': 1, 'TkVersion': 8.6, 'TclVersion': 8.6, 'READABLE': 2, 'WRITABLE': 4, 'EXCEPTION': 8, 'EventType': <enum 'EventType'>, 'Event': <class 'tkinter.Event'>, 'NoDefaultRoot': <function NoDefaultRoot at 0x014CD928>, 'Variable': <class 'tkinter.Variable'>, 'StringVar': <class 'tkinter.StringVar'>, 'IntVar': <class 'tkinter.IntVar'>, 'DoubleVar': <class 'tkinter.DoubleVar'>, 'BooleanVar': <class 'tkinter.BooleanVar'>, 'mainloop': <function mainloop at 0x014CDA90>, 'getint': <class 'int'>, 'getdouble': <class 'float'>, 'getboolean': <function getboolean at 0x03100148>, 'Misc': <class 'tkinter.Misc'>, 'CallWrapper': <class 'tkinter.CallWrapper'>, 'XView': <class 'tkinter.XView'>, 'YView': <class 'tkinter.YView'>, 'Wm': <class 'tkinter.Wm'>, 'Tk': <class 'tkinter.Tk'>, 'Tcl': <function Tcl at 0x03100190>, 'Pack': <class 'tkinter.Pack'>, 'Place': <class 'tkinter.Place'>, 'Grid': <class 'tkinter.Grid'>, 'BaseWidget': <class 'tkinter.BaseWidget'>, 'Widget': <class 'tkinter.Widget'>, 'Toplevel': <class 'tkinter.Toplevel'>, 'Button': <class 'tkinter.Button'>, 'Canvas': <class 'tkinter.Canvas'>, 'Checkbutton': <class 'tkinter.Checkbutton'>, 'Entry': <class 'tkinter.Entry'>, 'Frame': <class 'tkinter.Frame'>, 'Label': <class 'tkinter.Label'>, 'Listbox': <class 'tkinter.Listbox'>, 'Menu': <class 'tkinter.Menu'>, 'Menubutton': <class 'tkinter.Menubutton'>, 'Message': <class 'tkinter.Message'>, 'Radiobutton': <class 'tkinter.Radiobutton'>, 'Scale': <class 'tkinter.Scale'>, 'Scrollbar': <class 'tkinter.Scrollbar'>, 'Text': <class 'tkinter.Text'>, 'OptionMenu': <class 'tkinter.OptionMenu'>, 'Image': <class 'tkinter.Image'>, 'PhotoImage': <class 'tkinter.PhotoImage'>, 'BitmapImage': <class 'tkinter.BitmapImage'>, 'image\_names': <function image\_names at 0x031066E8>, 'image\_types': <function image\_types at 0x03110BF8>, 'Spinbox': <class 'tkinter.Spinbox'>, 'LabelFrame': <class 'tkinter.LabelFrame'>, 'PanedWindow': <class 'tkinter.PanedWindow'>}

PIL (Python image library)// from PIL import imageTk, image // pip install pillow // root.quit

Label( sticky=E+W) // a = IntVar() RadioButton(variable= a, value=1)

From PIL import ImageTk, image // from Tkinter import messagebox

Messagebox.showinfo(‘message..’), showwarning, showerror, askquestion, askokcancel, askyesno

Top = TopLevel() – for new window // filedialog.askfileopen(initialdir= , titlt=, filetype=)

Root.iconbitmap() // slider is Scale // checkButton.deselect() onvalue, offvalue

Database: import sqlite3 connect(database.db) connect.cursor connect.execute connect.close connect.commit

API install requests import requests, Jason

Jason.jcontent(api)

Chart: install Matplotlib

Import numpy, import matplotlib.pyplot

X = numpy.random.normal(1000,20000,300)

Pyplot.hist(x)

Pyplot.show()

import sqlite3

db = sqlite3.connect("citizen.db")

conn = db.cursor()

'''conn.execute("""create TABLE address (

first\_name text,

last\_name text,

address text,

city text,

zip\_code integer

)""")'''

print('Table created')

conn.execute("INSERT INTO address (first\_name, last\_name, address, city, zip\_code) VALUES ('Jaison', 'Jacob', '#Nagarbhavi', 'Bengaluru', 560071)")

conn.execute('SELECT \* FROM address')

records = conn.fetchmany(5)

print(records)

conn.execute("UPDATE address SET address = 'BDA Layout, Kadugodi' WHERE first\_name == 'Jaison'")

conn.execute('SELECT \* FROM address')

records = conn.fetchone()

print(records)

db.commit()

conn.close()

print('Database closed')

Pygame

#inititalizing pygame

pygame.init()

#winidow title

pygame.display.set\_caption("Book")

#load the image icon

surface\_icon = pygame.image.load("book.png")

#set the surface icon

pygame.display.set\_icon(surface\_icon)

#surface object

display\_surface = pygame.display.set\_mode((400,400))

#create a pygame font

#font = pygame.font.Font('freesansbold.ttf',32)

#create a system font

sys\_font = pygame.font.SysFont(‘arial',32)

#create text

text = sys\_font.render("GeeksforGeeks",True,(0,0,128),(200,150,25)#, boarder size as int

)

#create rectangular text surface

text\_rect = text.get\_rect()

text\_rect.center = (400//2, 400//2)

#draw a rectangle

rect = pygame.Rect((50,50), (50, 100))

#start the loop

while True:

#fill the surface with background color

display\_surface.fill((255,255,255))

#draw a rectangle on the surface

pygame.draw.rect(display\_surface, (10,100,175), ([100,20],[25,125]))

#add the text surface to the display surface

display\_surface.blit(text, text\_rect)

#looping the pygame events to look for exit

for event in pygame.event.get():

if event.type == pygame.QUIT:

pygame.quit()

# update the display surface to show the latest updates

pygame.display.update()

--

Turtle

import turtle

wn = turtle.Screen()

wn.bgcolor("blue")

wn.setup(width=500, height=250)

wn.title('Gaming unwind.. '+ str(wn.window\_height()))

wn.tracer(0)

pen = turtle.Turtle()

pen.write('Hellow world', move=False, align="left", font=("Ariel", 24, "bold"))

wn.mainloop()

Python 3: 1 Fred Baptise

[pep20.org](https://pep20.org/) was created by [Nik Kantar](http://nkantar.com/).  
[PEP 20](https://www.python.org/dev/peps/pep-0020/) was written by Tim Peters.

The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Review1

Variables: Are symbols for memory addresses

Numbers: Integers, rationales, floats, decimals, complex numbers – *cmath* standard library

Numeric types: Booleans – every object has one associated Truth value (In python everything is an object, including functions)

Tuples: Are not just read-only lists

You can also create a Python tuple without parentheses. This is called tuple packing. b= 1, 2.0, 'three'

A list item inside a tuple is mutable

t = tuple(lst)

t[4][0]= 'Jai'

print(t)

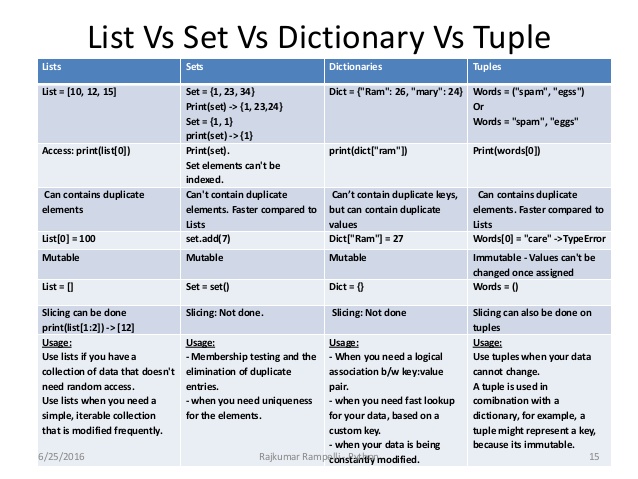
del t[4][0]

print(t)

output: (1, 2, 3, 4, ['Jai', 11, 12])

(1, 2, 3, 4, [11, 12])

--



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

Numbers: Integral – Integers, Booleans

Non Integral – floats, complex(has real and imaginary part), decimal, fractions(rationales)

Collections: Sequences – Mutable (lists) Immutable (Tuples, strings). Sets has only keys – Mutable(sets) Immutable(Frozen sets). Mappings(Dictionaries) has key value pairs

Callables – User defined functions, generators, classes, built in functions, built in methods, instance methods, class instances. Any object which can be called using a parenthesis ().

Singletons – None, notimplemented, Ellipsis

Comments cannot be part of a statement, not even a multi-line statement

Variant names must start with an underscore or a letter.

\_var // (starting with a single underscore) means it’s internal or private (python has nothing as private)

\_\_var\_ // double underscore – class attribute mangling in inheritance.

\_\_my\_var\_\_ // double underscore on both ends – system defined names for the interpreter

Pep 8 style guide

Package names – singleword, small letter, no underscore

Modules - singleword, small letter, can have underscore

Classes – CamelCase

Functions, variables – snake\_case

Constants – UPPERCASE, underscore

Conditionals

If: elif: else:

Ternary operator – (return) x if (condition is true) else (return) y

X = ‘a > b’ if a >5 else ‘b >= a’

Functions:

Functions has to be defined before it is called

Lambda is a nameless function, which returns to the variable it passed – lambda x: x \*\*2

fn = lambda x: x \*\*4 // fn(4) . Its an inline or anonymous function, which can be assigned to another function.

a = lambda x: x \*\* 2 if x > 2 else 0

print(a(2))

print(a(3))

output: 0, 9

While:

While true: (replaces do: while in other langs) // break, continue.

While loop executes else clause if it does not encounter any break statement in its code block

a = 100

while a > 95:

print(a)

a -=1

if a == 97:

continue

print(a)

# break can be used to break out of the loop

else:

print('exiting')

--------

Finally code block will run even though it encounters continue, break statements in a try block inside a while loopwhile True:

try:

a = int(input('Enter the number'))

b = int(input('Enter the number'))

c = a / b

print(c)

except (ZeroDivisionError, ValueError):

print(f'Division by zero error / Value error {a} b {b} = {c}')

finally:

print('Cleans up after exception')

For Loop: An iterable is an object, which is capable of returning a value, one at a time. (list, collection, range, string). Iterable is an object, which defines \_\_iter\_\_ protocol in its class. An iterable classs defines \_\_iter\_\_ returns an iterator object, which class defines \_\_next\_\_ in it.

Iterator class defines \_\_next\_\_ and \_\_iter\_\_ (returns ‘self’) in it.

When a class defines \_\_len\_\_ and \_\_getitem\_\_, a for loop can be used to retrieve items from that class.

Generators are Iterators, in that an Iterator doesn't have to generate new items, it can also be iterating over an existing collection of items.

Enumerate() function returns the index and the element.

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

x = list(enumerate(lst,10))

print(x)

for cnt, ele in enumerate(lst):

print(cnt, ele)

Class

Hex(id(instanceobject) – to find the memory address

Magic function: dundrr \_\_str\_\_ (replaces builtin func) // \_\_repr\_\_ // \_\_eq\_\_

The official Python documentation says \_\_repr\_\_() is used to compute the “official” string representation of an object. The repr() built-in function uses \_\_repr\_\_() to display the object. \_\_repr\_\_()  returns a printable representation of the object, one of the ways possible to create this object.  \_\_repr\_\_() is more useful for developers while \_\_str\_\_() is for end users.

The following code shows how \_\_repr\_\_() is used.

class Point:

   def \_\_init\_\_(self, x, y):

     self.x, self.y = x, y

   def \_\_repr\_\_(self):

     return 'Point(x=%s, y=%s)' % (self.x, self.y)

p = Point(3, 4)

print p

This gives the output

Point(x=3, y=4)

Isinstance(object, class) // monkey patching. Giving a value to an undefined property of an object at runtime

class Rectangle:

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

self.w = width

self.h = height

def area(self):

return self.w \* self.h

def perimeter(self):

return 2 \* (self.w \* self.h)

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

return 'Rectangle w = {0}, height = {1}'.format(self.w, self.h)

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

return 'Rectangle repr w = {0}, repr height = {1}'.format(self.w, self.h)

r1 = Rectangle(10, 20)

print('Width = {0}, Height = {1}'.format(r1.w, r1.h))

print('Area = ', r1.area())

print('Perimeter = ', r1.perimeter())

print(str(r1))

print(r1)

Decorators - @property getter def property\_name() // @propertyname.setter def property\_name()

\_\_init\_\_

Object mutability: Changing the data inside an object is called modifying the internal state of an object

An object, whose internal state can be changed is called mutable. An object, whose internal state cannot be changed is called immutable.

Immutable objects: Numbers(int,float,Boolean), strings, tuples, user define classes, frozen sets

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

Lists uses asynchronous blocks of memory. It does object type, object size, reference count, size checking. It uses pointers to data blocks.

Variables:

Everything in Python are objects. Variable references. Mutables: Lists, Dict. Immutables: Strings, tuples

Returning a function

def cube(x):

return x \* x

def select\_func(y):

if y == 1:

return cube

print(select\_func(1)(5))

output: 25

Passing function as an argument

def cube(x):

return x \* x

def exec\_func(fn, n):

return fn(n)

print(exec\_func(cube, 3))

output: 9

Assigning a function:

def cube(x):

return x \* x

def exec\_func():

return cube

r = cube # giving a reference for the address

m = exec\_func() # creating an object of the function

print(r(9))

print(m(4))

output : 81, 16

Optimization:

CPython, JPython, IornPython (c#), PyPi – written in RPython a subset of Python written in C, for writing Python interpreters.

Interning – re-using objects on demand

Singletons are classes that can be instantiated only once.

A= sys.intern(‘string’)

Start = Time.perf\_counter()

For code optimization, pre calculation is for 20 characters long

My\_func.\_\_code\_\_.co\_consts

Lists and tuples are ordered sequences

Sequences in Python

A sequence is a succession of values bound together by a container that reflects their type. Almost every stream that you put in Python is a sequence.

Types of Sequences

Lists

Tuples

Xrange

String

--

Numbers: Boolan Truth values(0 false, 1 true), Integer Z, Rational Q, Real R, Complex C

C, Java numeric data types. Byte, short, int, long.

import sys

print(sys.getsizeof(0))

default base of an integer is 10. It should be between 02 and 36

import sys

a = sys.intern('Hello')

b = sys.intern('Hello')

c = 'Hello'

d = 'dear\_champ'

e = 'dear\_champ'

print(a is c)

print(a==c)

print(d is e)

print(id(a), id(b), id(c), id(d), id(e))

sign = -1 if number is < 0 else 1 (another python expression!)

Bankers round to closest even numbers

Complex number (real, imag) stored as floats. Math module will not work. Instead use cmath module

Pep 385 for Boolean data types, which is a subset of numbers

Bool class is a subclass of int // issubclass(bool, int) // True and false are Singleton objects, which retain their memory address throughout their life time. (None, Interned strings also)

Bool is false only when the value is 0

All objects in Python has an associated truth value. Truthiness

Boolean operators: not, and, or.

Operator precedence: < > <= >= == != in is not and or

Database null will be converted to None in Python

String class has many constants: whitespaces, ascii\_uppercase, ascii\_lowercase ….

Boolean operators are defined operate on and return Boolean values // bool(‘abc’)

**Comparison operators**: Binary operators (takes 2 candidates either side),

unary – only 1 operand (evaluates true/false bool)

Identity operations – is , is not (compares mem address)

value comparisons == , != compares values, supports different types, must be compatible

Ordering operators: < > <= >= Doesn’t work with all types.

Membership operators: in, not in – work with iterable types

‘is’ is identity operator.

Parameters of a function: def func(a, b) – a and b are the parameters of the function func

In a function definition, if a positional parameter is given a default value, every parameter following should also be given a default value.

Keyword arguments are given at the time of function call. Func(2, b=10)

Once a named argument is used in a function call, thereafter all the arguments must be named

Packed values refer to values that are bundled together in some way

Variable swapping – assignment takes place after evaluation of references at the right side of assignment. Assignment copies values in memory as tuples first, then assign

Sets are also like unordered type dictionary, having only the key’s not values

Dictionary has keys and values called items

dict\_1 = {'a':1, 'b':2, 'c':3, 'd':4}

print(dict\_1)

for k, y in dict\_1.items():

''' or for e in dict\_1.items():

k, y = e '''

print('Item {}, Value {}'.format(k, y))

Unpacking of a list:

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

a, b = l[0], l[3:]

print(a)

print(b)

a, \*b = l

print(a)

print(b)

sets and dicts cannot be sliced

use \*\* to unpack a dictionary. It cannot be used on the LHS

dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

dict2 = {'e':6, 'g':7, \*\*dict1}

print(\*dict1)

print(dict1)

print(dict2)

To unpack an ordered item, LHS should be a list or a tuple. Review(04/05/20)

s = {9,8,7,6,5,4,3,2,1}

\*a, = s

print(a)

Merging of sets: Also can use set.union (no duplicates)

s = {9,8,7,6,5,4,3,2,1}

s1 = {'a','b','c','d','e'}

merge = {\*s, \*s1}

print(merge)

l2 = [11,22,33,44,'Python']

a,b,c,d = l2[-1][3:-1], l2[0],l2[1], list('Programming')

print(a,b,c,d)

Use of boolean and in Python:

def func\_1(\*args):

count = len(args)

total = sum(args)

return count and total / count # returns count if and is false, else total/count

print(func\_1(10,10,10))

10

\*args – scoops up positional arguments and it’s a tuple. \* indicates the end of positional arguments

\*\*kwargs scoops up keyword arguments and it’s a dictionary

Ternary operator: lo = return-value-if-condition-is-true <condition> else return-value-if-condition-is-false

List comprehension

In general pass mutable / callable objects as default values in function parameters

A function definition object is created only once it is defined.

Recursive function:

def factorial(n):

if n < 1:

return 1

else:

print('Factorial of a number {}'.format(n))

return n \* factorial(n-1)

factorial(5)

Python's default recursion limit is 1000, meaning that Python won't let a function call on itself more than 1000 times, which for most people is probably enough. The limit exists because allowing recursion to occur more than 1000 times doesn't exactly make for lightweight code.

First class object: It can be passed as an argument to a function, can be returned from a function, can be assigned to a variable, can be stored in a data structure (list, tuple, dictionary etc)

Functions are also first class objects

Higher order functions are functions that take another function as an argument, or returns a function. Sorted, map, filter

Function introspection

<func\_name>.category = ‘mystring’

Sorted(<iterable>, key=<function>)

jan = 'January'

print(sorted(jan, key=lambda x: x.upper()))

Functools functions: reduce, any, all

Documentation strings (docstrings) are stored in function property \_\_doc\_\_

Annotations give extra information about a function. Help(function name). Stored in \_\_annotations\_\_

a, b = 10, 15

def sample(c: str) -> 'repeated: ' + str(max(a,b)) + ' times':

"This is a function docstring of sample"

return c \* b

print(sample(5))

print(sample.\_\_doc\_\_)

help(sample)

dir(object) gives properties of the object

inspect – ismethod(func), isfunction(func), isroutine(func), signature(func), getsource(func), getmodule(isfunc)

map and filter, zip (are generators) does not return a list. Manually has to convert it to a list

Reducing functions – all, any

Partial functions takes a few arguments on behalf of other function parameters

Operator.getitem, itemgetter, delitem

sphinx – document generator for python

type hints – enhanced version of annotations. Type checking

lambda <parameter>: <expression> // the expression is returned when called

lambda’s are closures

l = lambda x,y: x \* y

print(l(5,2))

l = lambda x,y: x \* y

print(l(5,2))

def fnc\_1(x,y, fn):

return fn(x,y)

a = fnc\_1(7,3,l)

print(a)

def fnc\_1(x, y, fn):

return fn(x, y)

a = fnc\_1(7, 3, lambda x, y: x \* y)

print(a)

--

def myfunc(n):

return lambda a : a \* n

mydoubler = myfunc(2)

print(mydoubler(11))

output: 22

Body of lambda limited to a single expression. No assignments, annotations. Line continuation is ok in a single expression.

Ord(‘A’) // ordinal or ascii number

Deep copy. Deep copy is a process in which the copying process occurs recursively. ... In case of deep copy, a copy of object is copied in other object. It means that any changes made to a copy of object do not reflect in the original object. In python, this is implemented using “deepcopy()” function.

A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original. The copying process does not recurs and therefore won't create copies of the child objects themselves. In case of shallow copy, a reference of object is copied in other object.

Permutations

First import itertools package to implement permutations method in python. This method takes a list as an input and return an object list of tuples that contain all permutation in a list form.

## Generators

[Generator](https://docs.python.org/3/glossary.html#term-generator)s are a simple and powerful tool for creating iterators. They are written like regular functions but use the [yield](https://docs.python.org/3/reference/simple_stmts.html#yield) statement whenever they want to return data. Each time [next()](https://docs.python.org/3/library/functions.html#next) is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

def reverse(data):

for index in range(len(data)-1, -1, -1):

yield data[index]

--

>>> for char in reverse('golf'):

... print(char)

...

f

l

o

g

--

>>> sum(i\*i for i in range(10)) # sum of squares

285

Function introspection: dir(object) gives the object properties

*Inspect* module: ismethod(obj), isfunction(obj), isroutine(obj), isfunction(func\_name)

A method is declared inside a class, which is bound to a class object. A function is declared in a module, not inside a class.

Inspect.getsource(func\_name) // inspect.getmodule(func\_name) // inspect.getcomments(func\_name) // inspect.signature(func\_name)

Python can use / in function definitions. User cannot. Func(x,y /)

Built in functions are the functions defined inside a module, not associated to a class.

Built in methods are inside a class.

Class() - \_\_new\_\_() –new object \_\_init\_\_() – instantiate new object, returns object

Class instances are callable, one which implement \_\_call\_\_ method in it

# map

def calc(x, y):

return x + y

l1 = [0, 1, 2, 3]

l2 = [10, 11, 12, 13]

l3 = list(map(calc, l1, l2))

print("Map: ", l3)

output: Map: [10, 12, 14, 16]

# Filter

def odd\_even(n):

return n \*\* 2

l4 = list(filter(odd\_even, l1))

print("Filter: ", l4)

# Zip

l1 = [1,22,3,4]

l2 = ['a','b',4,5,21]

l5 = list(zip(l2, l1))

print("Zip: {0}".format(l5))

Output: Zip: [('a', 1), ('b', 22), (4, 3), (5, 4)]

filter() function may choose an item even if the "decision function" doesn't explicitly return True but evaluates to True instead, such as 1

List comprehension: [<expression> for <iterator> in <list iterable>] // [x\*\*2 for x in l1 <if condition>]

print("List Comprehension: ", [x\*\*2 for x in l1])

Reducing functions (aggregates): Reduce works on any iterable. Functools Min, max, sum, any, all

Reduce(lambda function) – initializer

--

from functools import reduce

print(reduce(lambda x,y: x \* y, range(1, 5+1)))

Output: 120

--

Reducing function arguments:

def my\_func(a,b,c):

print(a,b,c)

def func(b,c):

return my\_func(10,b,c)

print(func(15,20))

--

Partial function:

from functools import partial

def my\_func(a, b, c):

print(a, b, c)

x = partial(my\_func, 20, 25)

x(5)

output: 20 25 5

--

*Operator module* is created to give functional equivalent to many of the operators. Arithmetic:

Add(a,b), sub(a,b), mul, div, floordiv, neg ..

Comparison and Boolean operators: lt(a,b), gt, le, ge, eq, ne, and\_, or\_ ..

Sequence / mapping functions: concat(a,b), contains, countOf, getitem/ itemgetter(i), setitem(s,I,val),delitem

A = Attrgetter(<property>) // a(obj)

--

from operator import attrgetter, methodcaller

#attrgetter callable function

s = 'Python'

obj = attrgetter('upper')

print(obj(s)())

#methodcaller function

print(methodcaller('upper')('python'))

Output:

PYTHON

PYTHON

--

Section 7: Scopes, closures, decorators

Variable scopes: local, nonlocal, global, nested

Nested scope: A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope. In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

Namespace is a table and it contains the labels and their references. Each scope has its name space

Global scope is module scope. It spans a single file only

Some builtin global are available across the modules. True, False, None, dict, print

Variables created inside a function is ‘local’ are really created when the function is ‘called’.

Use global and nonlocal statements inside a scope to refer them.

A double underscore \_\_ prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

Class variables are shared across all objects while instance variables are for data unique to each instance.

Instance variable overrides the Class variables having same name which can accidentally introduce bugs or surprising behaviour in our code.

Example: Private Attributes

class employee:

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

self.\_\_name=name # private attribute

self.\_\_salary=sal # private attribute

>>> e1=employee("Bill",10000)

>>> e1.\_\_salary

AttributeError: 'employee' object has no attribute '\_\_salary'

Python performs name mangling of private variables. Every member with double underscore will be changed to \_object.\_class\_\_variable. If so required, it can still be accessed from outside the class, but the practice should be refrained.

>>> e1=Employee("Bill",10000)

>>> e1.\_Employee\_\_salary

10000

>>> e1.\_Employee\_\_salary=20000

>>> e1.\_Employee\_\_salary

20000

--

**class** A:  
 \_\_x = 10

a = A()  
print(a.\_A\_\_x)

output: 10

A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class.

Decorators are powerful and useful, but you need to know how to use them properly to make the most out of this wonderful feature. In this article I’ll be discussing how we can use decorators with additional parameters (not the function alone). We will see 2 techniques of doing this, one with a function and one with a class.

Why do we even need this? lets start with a very short intro on decorators and then see one way in which we are limited. \*If you already understand decorators feel free to skip to the “after decorators” headline.

Decorators are a python feature that adds some functionality to some function. In practice it means I can write a decorator, which does something and by decorating another function, I add this functionality to a new function, without writing the code twice. An example would make it easy to understand:

def typing(num):

print('typing the number', num)

typing(4)

---> typing the number, 4

This is a rather simple function. Now lets say I have 4 functions and I want to add timing to all of them, I want to know how long did it take to run. One solution would be adding timing functionality to every function, but as you can imagine this isn’t very DRY (Don’t Repeat Yourself). Instead what we do is we write this timing function once, and add (decorate) this function to any other function that wants “its services”.

# Timing functionality from Python's built-in module

from time import perf\_counter

def decorating(fn):

def inner(\*args):

start = perf\_counter()

result = fn(\*args)

end = perf\_counter()

elapsed = end - start

print(result)

print('elapsed', elapsed)

return inner

So we have a function that starts calculating time, Runs the original function (the function using the timing services), calculates time again on a new variable and returns the difference (e.g time elapsed). Lets define a function that calculates a factorial, and add to it a timing functionality.

@decorating

def calc\_factorial(num):

if num < 0:

raise ValueError('Please use a number not smaller than 0')

product = 1

for i in range(num):

product = product \* (i+1)

return product

calc\_factorial(4)

--->

24

elapsed 5.719979526475072e-06

calc\_factorial(10)

--->

3628800

elapsed 6.7150103859603405e-06

And there it is! we get timing functionality for free, with only adding this @decorating symbol. You can imagine why this is much more scalable than writing everything again and again. Those are decorators, this is the main idea.

After Decorators

We’ve seen how decorators could be very useful, that being said, writing them this way has one important limitation we’ll discuss. What happens if I want to add some parameters to my decorating function that will be later on used by the decorated function?. meaning, what if I not only want to add functionality, but also transfer data. For example say I’m testing different ways to write the same function for optimization purposes. I want to run the same function many times and get an average of the time it took. Can I pass the number of loops as a parameter? not like this. To be able to also transfer data we need to create what is called a Decorator Factory

\* Like anything this could be done in many ways, we will use a way that demonstrates transferring the data all the way from the Decorator Factory.

Decorator Factory

A decorator factory is a function that returns a decorator. Instead of returning a function (inner in our case), we will return a decorator (named “dec”). Lets see this:

from time import perf\_counter

def decorator\_factory(loops\_num):

def decorating(fn):

def inner(num):

total\_elapsed = 0

for i in range(loops\_num):

start = perf\_counter()

result = fn(num)

end = perf\_counter()

total\_elapsed += end - start

avg\_run\_time = total\_elapsed/loops\_num

print('result is', result)

print('num of loops is', loops\_num)

print('avg time elapsed', avg\_run\_time)

return inner

return decorating

See that here we return both the inner function and the decorating function. What we are returning, is a decorator. This enables the inner function to have access to some additional parameters (e.g a, b) and use them.

\* For this particular example we will be using num and not \*args to make it a bit more understandable. Usually \*args could be preferable as its more flexible.

Now we don’t only have the timing functionality, but we also have access to more parameters, something we could not have done earlier.

@decorator\_factory(500)

def calc\_factorial2(num):

if num < 0:

raise ValueError('Please use a number not smaller than 0')

product = 1

for i in range(num):

product = product \* (i+1)

return product

calc\_factorial2(4)

--->

result is 24

num of loops is 500

avg time elapsed 1.5613697469234467e-06

Now we can use the looping functionality with other functions as well.

@decorator\_factory(5)

def string(s):

print(s)

string('this is working'

--->

this is working

this is working

this is working

this is working

this is working

num of loops is 5

Write once, run everywhere.

Class Decorator Factory

Lets now look at an alternative way to generate the same behavior, but with a class. A class is sometimes easier to make more complicated operations with, and so its an important tool to have. Lets do exactly what we did with the decorator factory function, this time with a class.

class Decorator\_Factory\_Class:

def \_\_init\_\_(self, num\_loops):

self.num\_loops = num\_loops

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

def inner(num):

total\_elapsed = 0

for i in range(self.num\_loops):

start = perf\_counter()

result = fn(num)

end = perf\_counter()

total\_elapsed += end - start

avg\_run\_time = total\_elapsed/self.num\_loops

print('num of loops is', self.num\_loops)

return result

return inner

As a result, we can now use the class the same way we used the function

@Decorator\_Factory\_Class(5)

def calc\_factorial2(num):

if num < 0:

raise ValueError('Please use a number not smaller than 0')

product = 1

for i in range(num):

product = product \* (i+1)

return product

calc\_factorial2(4)

--->

num of loops is 5

avg\_run\_time is 2.301810309290886e-06

24

--

b = 'Hello'

def outer():

b = 20

def inner():

nonlocal b

b = 30

print(b)

def innest():

global b

b = 'Mr'

innest()

inner()

print(b)

outer()

print(b)

--output: 30

30

Mr

--

Closures enclose the referred free variables in the inner function. The inner function returns instead of called. Intermediate cell stores the value.

When a variable is referenced inside a nested function, but which is declared outside, it’s a closure.

Shared closures, nested closures

--

def incrementor(n):

# inner

def inner(start):

current = start

# inc

def inc():

nonlocal current

current += n

return current

return inc

return inner

incr = incrementor(5)

innr = incr(100)

print(innr())

print(innr.\_\_code\_\_.co\_freevars)

print(incr.\_\_code\_\_.co\_freevars)

Output: 105

('current', 'n')

('n',)

Decorators: A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class.

#decorator

def decor(func):

print('Insde decor func')

def wrapper(x):

print('Inside wrapper function')

print('calling'+ func.\_\_name\_\_)

func(x) # calling juno

func() # calling hello

return wrapper

@decor

def hello():

print('Inside hellow')

@decor

def juno(name):

print('Inside juno '+name)

juno('Jaison')

--

def counter(fn):

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

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

return inner

# appends to a list

def add(\*args):

lst = []

for i in args:

lst.append(i)

return lst

x = counter(add) # counter function decorates the function 'add'

* Output: [10, 20, 30, 40, 50]

@<decorator name>wraps // @<decorator name> (function name as argument)

Def <func name>():

Statements …

Decorator factory: Decorator function has to be defined first with nested functions..

dec = decorator\_function(10,20)

@dec

Def <func name>():

Statements..

--

@decorator\_func(10,20)

Def func\_name():

Stmnts..

---

Myfunc = dec\_factory(10,20)(myfunc)

Use underscore (\_) as variable names to indicate unimportant data.

Tuples are containers, order matters, heterogeneous, indexable, iterable, immutable, fixed length, fixed order

# extended unpacking of a tuple

t = 1, 2, 3, 4, 'a', 'b', 5

# want to unpack only the last two elements

\*\_, a, b = t

print(\_)

print(a, b)

output: [1, 2, 3, 4, 'a']

b 5

A function in Python can return more than a single value.

def func():

a,b,c = 10,20,30

return a,b,c

x,y,z = func()

print(f'The sum of 3 nums are {x + y + z}')

output: The sum of 3 nums are 60

from collections import namedtuple

namedtuple is a function, which is a classfactory. It creates new class which inherit from class tuple.

It provides named properties to access the elements of the tuple.

**namedtuple**() Factory Function for Tuples with Named Fields. Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

import collections

# Declaring namedtuple()

Student = collections.namedtuple('Student', ['name', 'age', 'DOB'])

# Adding values

S = Student('Nandini', '19', '2541997')

# initializing iterable

li = ['Manjeet', '19', '411997']

# initializing dict

di = {'name': "Nikhil", 'age': 19, 'DOB': '1391997'}

# using \_make() to return namedtuple()

print("The namedtuple instance using iterable is : ")

print(Student.\_make(li))

# using \_asdict() to return an OrderedDict()

print("The OrderedDict instance using namedtuple is : ")

print(S.\_asdict())

# using \*\* operator to return namedtuple from dictionary

print("The namedtuple instance from dict is : ")

print(Student(\*\*di))

Namedtuple return a class. ClassN = namedtuple(‘ClassN’,[‘x’,’y’]). Class.\_fields

Namedtuple -> to dictionary - instace.\_ascdict()

Output:

The namedtuple instance using iterable is :

Student(name='Manjeet', age='19', DOB='411997')

The OrderedDict instance using namedtuple is :

{'name': 'Nandini', 'age': '19', 'DOB': '2541997'}

The namedtuple instance from dict is :

Student(name='Nikhil', age=19, DOB='1391997')

--

*Changing the value of a tuple parameter:* pt = namedtuple(‘Point2D’,’x’,’y’)

pt = Point2D(10,20)

pt = Point2D(100, pt.y) // created a new reference for the ‘x’ parameter using the existing instance.

Extending a namedtuple // StockExt = namedtuple(‘Stock’,Stock.\_fields + (‘previous\_close’,))

-

from collections import namedtuple

# creating the tuple

Stock = namedtuple('Stock', 'one two three four five six')

stk = Stock(1, 2, 3, 4, 5, 6)

print(stk)

values = stk[:5]

stk = stk.\_make(values + (100,))

print(stk) review 7th apr

stk = stk.\_replace(three=.3, four=.4)

print(stk)

# extending the current tuple

ExtStock = namedtuple('ExtStock', stk.\_fields + ('Seven',))

stk\_ext = ExtStock(\*stk, 7)

print(stk\_ext)

print(stk\_ext.\_fields)

* Output: Stock(one=1, two=2, three=3, four=4, five=5, six=6)
* Stock(one=1, two=2, three=3, four=4, five=5, six=100)
* Stock(one=1, two=2, three=0.3, four=0.4, five=5, six=100)
* ExtStock(one=1, two=2, three=0.3, four=0.4, five=5, six=100, Seven=7)
* ('one', 'two', 'three', 'four', 'five', 'six', 'Seven')

--

Modules are loaded from built-in, standard, 3rd party libraries. Modules are of type ‘module’. Modules has namespace. Once a module is loaded at run time, it will be there in the ‘sys cache’.

Create A1.py as module1 (write code)

def hello():  
print('------------- hello from A1 module function--------------')  
  
  
print('--------Hi from A1 module.......')  
#hello()

Create Main.py write below..

import os.path

import types

import sys

module\_name = 'module1'

module\_file = 'A1.py'

module\_path = '.'

module\_rel\_file\_path = os.path.join(module\_path, module\_file)

module\_abs\_file\_path = os.path.abspath(module\_rel\_file\_path)

with open(module\_rel\_file\_path, 'r') as code\_file:

source\_code = code\_file.read()

mod = types.ModuleType(module\_name)

mod.\_\_file\_\_ = module\_abs\_file\_path

sys.modules[module\_name] = mod

code = compile(source\_code, filename=module\_abs\_file\_path, mode='exec')

exec(code, mod.\_\_dict\_\_)

import module1

module1.hello()

output: --------Hi from A1 module.......  
------------- hello from A1 module function--------------

--

Module.\_\_dict\_\_ // module.\_\_spec\_\_ // module.\_\_name\_\_ // module.\_\_file\_\_ // module.\_\_package\_\_

Pathfinder: sys.path or package.\_\_path\_\_ // built-in finder, frozen finder, pathfinder

Mod = importlib.import\_module(‘fractions’)

Finder + loader = import // sys.meta\_path // sys.path // importlib.utils.find\_spec(‘module2’) // os.enviorn[‘HOMEPATH’]

Sys.modules.keys() // importlib.reload(module\_name) // importlib.reload(sys.modules(module\_name)) // sys.meta\_path for module finder.

ModuleSpec is the module loader returned by the module finder.

A decorator is a function that receives another function as argument. The behaviour of the argument function is extended by the decorator without actually modifying it.

\_\_new\_\_() method returns a new object, which is then initialized by \_\_init\_\_().

Class Attributes

Class attributes are different from instance attributes. An attribute whose value is the same for all instances of a class is called a class attribute. The value of class attribute is shared by all objects. Class attributes are defined at class level rather than inside the constructor method \_\_init\_\_(). Unlike instance attributes, class attributes are accessed using the name of the class.

Argparser.ArgumentParser(descriptor=\_\_doc\_\_) // Parser.add\_argument(….) // parser.parse\_args

Zipfile. In command line, python –m zipfile <zipfile\_name> <zipfiles, ,..> // list zip file contents: python –m zipfile –l. The run file’s name should be \_\_main\_\_

Code for the ‘package’ will be inside \_\_init\_\_.py in the directory where the package resides. It also contains \_\_path\_\_, \_\_file\_\_.

Modules has \_\_file\_\_ and \_\_package\_\_ properties

Namespacce packages will not contain a \_\_init\_\_.py file in it.

Class inheritance:

class Rocket:

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

self.name = name

self.distance = distance

self.country = country

def reset(self, name, distance, country):

self.name = name

self.distance = distance

self.country = country

def r\_print(self):

print('Rocket Name is {0}'.format(self.name))

print('Rocket Distance covered is {0}'.format(self.distance))

print('Rocket Maker is {0}'.format(self.country))

class Planet(Rocket):

count = 0

def \_\_new\_\_(cls, \*args, \*\*kwargs):

cls.\_inst = super(Planet, cls).\_\_new\_\_(cls)

cls.count += 1

print('Instantated ', cls.count)

return cls.\_inst

def \_\_init\_\_(self, name, distance, country, destination):

super().\_\_init\_\_(name, distance, country)

self.destination = destination

def p\_print(self):

print('Rocket Name is %s' % self.name)

print('Rocket Distance covered is %s' % self.distance)

print('Rocket Maker is %s' % self.country)

print('Rocket aim for %s' % self.destination)

p1 = Planet('Fire', 1000, 'India', 'Sun')

p2 = Planet('Fire', 1000, 'India', 'Sun')

p3 = Planet('Fire', 1000, 'India', 'Sun')

p1.p\_print()

p1.r\_print()

---

output: Instantated 1

Instantated 2

Instantated 3

Rocket Name is Fire

Rocket Distance covered is 1000

Rocket Maker is India

Rocket aim for Sun

Rocket Name is Fire

Rocket Distance covered is 1000

Rocket Maker is India

--

*#super().\_\_init\_\_ always execute the init of the super class***class** Root:  
 **def** \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
**class** Sub(Root):  
  
 **def** \_\_init\_\_(self, a, b):  
 self.a = a  
 self.b = b  
 super().\_\_init\_\_(a,100)  
  
s = Sub(5,10)  
print(s.\_\_getattribute\_\_(**'y'**))  
  
print(s.\_\_dict\_\_)

What is a static method?

Static methods in Python are extremely similar to python class level methods, the difference being that a static method is bound to a class rather than the objects for that class.

This means that a static method can be called without an object for that class. This also means that static methods cannot modify the state of an object as they are not bound to it.

Advantages of Python static method

Static methods have a very clear use-case. When we need some functionality not w.r.t an Object but w.r.t the complete class, we make a method static. This is pretty much advantageous when we need to create Utility methods as they aren’t tied to an object lifecycle usually.

# Python program to demonstrate

# use of class method and static method.

from datetime import date

class Person:

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

self.name = name

self.age = age

# a class method to create a

# Person object by birth year.

@classmethod

def fromBirthYear(cls, name, year):

return cls(name, date.today().year - year)

# a static method to check if a

# Person is adult or not.

@staticmethod

def isAdult(age):

return age > 18

person1 = Person('mayank', 21)

person2 = Person.fromBirthYear('mayank', 1996)

print (person1.age)

print (person2.age)

# print the result

print (Person.isAdult(22))

Output:

21

22

True

Always use self for the first argument to instance methods.

Always use cls for the first argument to class methods.

Generate random numbers: import random

lst = [random.randint(10,20) for \_ in range(1\_000)]

random.choice(lst)

random.choices(lst, k=5)

--

import random

lst = [random.randint(10, 20) for \_ in range(15)]

print(lst)

def freq\_analysis(lst):

return {k: lst.count(k) for k in set(lst)}

x = freq\_analysis(lst)

print(lst.count(10))

print(x)

--- output: [15, 20, 19, 13, 12, 18, 16, 17, 12, 17, 12, 12, 10, 13, 19]

1

{10: 1, 12: 4, 13: 2, 15: 1, 16: 1, 17: 2, 18: 1, 19: 2, 20: 1}

--

from timeit import timeit

x = timeit(stmt='math.sqrt(2)', setup='import math', globals = globals(), number = number)

print(x)

output: 0.1616077

--

Sys.argv // sentinel values as parameter defaults

Decorator:

def decorator(func):

print('Decorator called')

def inner():

print('Inner called')

ups = func()

upr = ups.upper()

uprs = upr.split()

return uprs

return inner

@decorator

def say\_hai():

return 'Hello world'

print(say\_hai())

output: Decorator called

Inner called

['HELLO', 'WORLD']

Part 2

Builtin sequence types:

1. Mutabe: Lists, byte arrays
2. Immutabe: Strings, tuples, range, byte

Collection objects: namedtuples, deque, array etc

An iterable object is a container, iterable, indexable. Indexing is not available in sets

Upper bound is excluded in slicing – list[0:9] excluding 9.

When using concatenation with a sequence, use mutable elements. Otherwise will copy the same memory address. Modifying a mutable object, will modify the copied object also, because memory address is same for both.

List object inside a tuple is mutable. A = ([10,20],20,30) // a[0][0]=15 // a([15,20],20,30)

Enumerator return a container of tuple

Slicing always return a new object

Mutate lists using index assignment, append tool. Concatenation using + creates a new list object in memory.

Del lst[i:j] – deletes items from the list. Extending the list also is possible

Some methods supported by mutable sequences like list:

s.append(x), s.insert(i,x), s.clear(), s.extend(iterable), s.pop(i), s.remove(x), s.reverse(), s.copy()

slicing always return a new object

shallow copy copies the contends of the object and points to the same address. Any modification in either object will reflect in the other also.

Constant folding is the process of recognizing and evaluating the constant expressions at compile time rather than run time.

Copying a list can be done using any of the following methods: a loop, a list comprehension, slice, list method call, copy method

If the sequence is immutable, its okay to return the sequence from the function.

#when a copied 'list' is modified, the original object also modified. Individual elements has no effect of change

a = [[10,20],11,22,['a','b']]

print(id(a))

b = a.copy()

b[1]= 25

b[0][0] = 'python'

b[3][0] = 'Java'

print('a',(a))

print('b',(b))

print(id(a))

print(id(b))

------- Output:

57756552

a [['python', 20], 11, 22, ['Java', 'b']]

b [['python', 20], 25, 22, ['Java', 'b']]

57756552

57755912

Copy module, copy, deepcopy, \_\_copy\_\_, \_\_deepcopy\_\_

Deepcopy works for objects in general, not for sequences

Set and dictionary cannot be indexed. Unordered collections

Slice is a type of object slice. A = slice(<index>)

#slice

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

s = slice(0,3)

b = a[s]

print('b', b)

output: b [1, 2, 3]

* S.start, s.stop, s.step

A **set** is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the **set** itself is mutable. ... **Sets** can be **used** to perform mathematical **set** operations like union, intersection, symmetric difference etc.

a = list(range(\*slice(10,-5,-1).indices(6)))

print(a)

output: [5, 4, 3, 2]

-- review 04/8

Sequence types are iterables, but not all iterables are sequences. Implements \_\_len\_\_, \_\_getitem\_\_

IndexError exception

Function caching allows us to cache the return values of a function depending on the arguments. ... In **Python** 3.2+ there is an **lru\_cache** decorator which allows us to quickly cache and uncache the return values of a function.

LRU cache stand for Least Recently Used Cache. which evict least recently used entry. As Cache purpose is to provide fast and efficient way of retrieving data.

A **staticmethod** is a method that knows nothing about the class or instance it was called on. It just gets the arguments that were passed, no implicit first argument. It is basically useless in **Python** -- you can just use a module function instead of a **staticmethod**

**classmethod**() in **Python**

The **classmethod**() is an inbuilt function in **Python** which returns a **class method** for given function. ... Return Type:This function returns converted **class method**. **classmethod**() methods are bound to class rather than an object. Class methods can be called by both class and object.

Inplace concatenation uses ‘+=’ ‘\*=’ and mutates the list and uses the same mem address

Using inplace concatenation, any iterable types can be mutated. i.e, list += tuple …

Deleting a slice is by using empty iterable

Insert using an empty slice [1:1]

List has an inplace sorting method. Lst.sort(key = lambda s: len(s)). It mutates the list.

List comprehensions are functions. Python creates a function at compile time and runs

An assert statement evaluate whether an expression is true. If it’s true, nothing happens; else an exception is raised.

Assert hasattr(Base, ‘foo), ‘fool, you broke the program’

--

a = int(input('Enter a'))

assert a > 10, 'A is > 10'

print(a)

class MyClass:

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

self.\_a = a

self.\_b = b

@property

def perimeter(self):

self.\_a = self.\_a \* self.\_b

return self.\_a

@perimeter.setter

def perimeter(self,a):

self.\_a = a

m1 = MyClass(3,5)

print(m1.perimeter)

m1.perimeter = 100

print(m1.perimeter)

--

If a collection object implements \_\_get\_next\_item, it can be iterated. If \_\_next\_\_ is implemented on the object, object can be iterated by builtin function next(object).

Iterator is one object which implements these two methods. \_\_iter\_\_ returns the object itself, \_\_next\_\_ returns the next element from the collection and raise StopIteraton exception.

Iterator protocol, iterable protocol

Iterator protocol says that the \_\_iter\_\_ should return an iterable object in an iterator

class Cities:

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

self.\_cities = ['Bengaluru', 'Chennai', 'Mumbai', 'Kolkota', 'Delhi']

self.\_index = 0

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

return len(self.\_cities)

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

print('Cities \_\_iter\_\_ called')

return CityIterator(self)

class CityIterator:

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

self.\_cities\_obj = cities\_obj

self.\_index = 0

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

print('City Iterator called \n -----')

return self

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

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

raise StopIteration

else:

item = self.\_cities\_obj.\_cities[self.\_index]

self.\_index += 1

return item

city = Cities()

x = CityIterator(city)

for i in x:

print(i)

output:

City Iterator called

-----

Bengaluru

Chennai

Mumbai

Kolkota

Delhi

Inner class:

class Human:

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

self.name = 'Guido'

self.head = self.Head()

class Head:

def talk(self):

return 'talking...'

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

guido = Human()

print guido.name

print guido.head.talk()

--

An iterable object returns an object everytime it is called. An iterator object returns objects only one time. An iterator object implements \_\_iter\_\_ and \_\_next\_\_.

Sentinel value is a value in a condition where the callable function returns and break the iteration

Reversed(list) – is an iterator function

List comprehension: x = [[i \* 3, i + i] for i in range(10)]

print(x) // output: [[0, 0], [3, 2], [6, 4], [9, 6], [12, 8], [15, 10], [18, 12], [21, 14], [24, 16], [27, 18]]

Importantly, it should be noted that iterators are stateful. Meaning once you’ve consumed an item from an iterator, it’s gone. So after you’ve looped over an iterator once, it’ll be empty if you try to loop over it again.

In Python 3, enumerate, zip, reversed, and a number of other built-in functions return iterators

Generators are essentially iterators. They are functions that returns generators.

genexpr = (expression for item in collection if condition)

for x in ('Bom dia' for i in range(3)):

print(x)

Generator Expressions in Python – Summary

Generator expressions are similar to list comprehensions. However, they don’t construct list objects. Instead, generator expressions generate values “just in time” like a class-based iterator or generator function would.

Once a generator expression has been consumed, it can’t be restarted or reused.

Generator expressions are best for implementing simple “ad hoc” iterators. For complex iterators, it’s better to write a generator function or a class-based iterator.

(expr for x in xs if cond1

for y in ys if cond2

...

for z in zs if condN)

Yield does 3 things: emit a value, use next to get that emitted value, now the function pauses. Use next again to resume the function.

A function that uses the ‘yield’ statement is a generator function.

A method inside a class can call a function outside. A function can call a method if that method does not use ‘self’ as parameter. A method which does not use ‘self’ as parameter, cannot be invoked from the class object.

*#Generator function***class** FibIterator(object):  
 **def** \_\_init\_\_(self, n):  
 self.n = n  
*#Iterator for the generator* **def** \_\_iter\_\_(self):  
 **return** FibIterator.gen(self.n)  
*#Generator function* @staticmethod  
 **def** gen(n):  
 **for** i **in** range(n):  
 **yield** i \* n  
*#class method can be called form class and object* @classmethod  
 **def** pprint(self):  
 print(**'Pretty print'**)  
  
  
sq = FibIterator(5)  
**for** i **in** sq:  
 print(i)  
sq.pprint()  
  
print(FibIterator.pprint())

--

Output:

0

5

10

15

20

Pretty print

Pretty print

None

--

Generator expressions or comprehensions use round parenthesis instead of square brackets

*#list comprehension*lst = [i\*\*i **for** i **in** range(5)]  
*#Generator comprehension or expression*tpl = ( i \*\* i **for** i **in** range(5))  
  
print(lst)  
  
print(tpl)  
  
**for** i **in** tpl:  
 print(i)

Output:

[1, 1, 4, 27, 256]

<generator object <genexpr> at 0x01139140>

1

1

4

27

256

--

In Generators, we do lazy evaluation.

[]: Used to define mutable data types - lists, list comprehensions and for indexing/lookup/slicing.

(): Define tuples, order of operations, generator expressions, function calls and other syntax.

{}: The two hash table types - dictionaries and sets.

An asterisk (\*) is placed before a parameter in function definition which can hold non-keyworded variable-length arguments and a double asterisk (\*\*) is placed before a parameter in function which can hold keyworded variable-length arguments.

To compile code, import dis

Compile(<expression code>, filename= ‘<String>’, mode =’eval’)

Disassemble: dis.dis(exp)

Import tracememalloc // stat = tracememalloc.take\_snapshot().statistics(‘linemo’)

Print(stat[0].bytes)

--

The eval() vs. exec() built-in functions (in Python 2, exec is a statement); the former is for expressions, the latter is for statements.

The curses package is part of the Python standard library and is useful for creating text-based user interfaces and generally controlling the screen and keyboard input. ... It uses the windows-curses package. The curses library goes back decades and is used to control terminal/shell output.

Curses.wrapper(main)

--

Partial function:

from functools import partial

def multiply(x,y):

return x \* y

# create a new function that multiplies by 2

dbl = partial(multiply,2) # 2 replaces param ‘x’

print(dbl(4))

output: 8

--

If you know you can loop over something, it’s an iterable.

If you know the thing you’re looping over happens to compute things as you loop over it, it’s a lazy iterable.

If you know you can pass something to the next function, it’s an iterator (which are the most common form of lazy iterables).

 Iterator is an object which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation. The iterator protocol consists of two methods. The \_\_iter\_\_() method, which must return the iterator object, and the next() method, which returns the next element from a sequence. Python has several built-in objects, which implement the iterator protocol.

**Iterable** is an object, which one can iterate over. It generates an Iterator when passed to iter() method. Iterator is an object, which is used to iterate over an **iterable** object using \_\_next\_\_() method.

#Iterable list made into an iterator

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

print(lst)

x = iter(lst)

print(x.\_\_next\_\_())

print(x.\_\_next\_\_())

print(x.\_\_next\_\_())

print(next(x))

output: [1, 2, 3, 4, 5]

1

2

3

4

Python | Difference between iterable and iterator

Iterable is an object, which one can iterate over. It generates an Iterator when passed to iter() method. Iterator is an object, which is used to iterate over an iterable object using \_\_next\_\_() method. Iterators have \_\_next\_\_() method, which returns the next item of the object.

Note that every iterator is also an iterable, but not every iterable is an iterator. For example, a list is iterable but a list is not an iterator. An iterator can be created from an iterable by using the function iter(). To make this possible, the class of an object needs either a method \_\_iter\_\_, which returns an iterator, or a \_\_getitem\_\_ method with sequential indexes starting with 0.

Python generators

Generator is a special routine that can be used to control the iteration behaviour of a loop. A generator is similar to a function returning an array. A generator has parameters, it can be called and it generates a sequence of numbers. But unlike functions, which return a whole array, a generator yields one value at a time. This requires less memory.

Generators in Python:

Are defined with the def keyword

Use the yield keyword

May use several yield keywords

Return an iterator

**From Itertools import \***

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, permutations, combinations, combinations\_with\_replacement.

If a class implements either a \_\_bool\_\_ or \_\_len\_\_ which returns False and 0 respectively, its truthiness is false.

Islice is a lazy iterator

reduce() in Python

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.This function is defined in “functools” module.

Working :

At first step, first two elements of sequence are picked and the result is obtained.

Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.

This process continues till no more elements are left in the container.

The final returned result is returned and printed on console.

Chain (\* args) // variable number of arguments and all arguments are iterables

For it in (iter1, iter2, iter3)

Print(it)

Teeing is copying of iterable ‘n’ times. All the copied iterables are of lazy ones, eventhough the copied iterable is note.

What are the sequences: lists, tuples, strings. All sequences can be sliced. Tee does not return a list. It returns a tuple of iterators.

Iterators

Iterator is an object that remembers iteration count via an internal state variable.

Nothing resets this variable back to zero when the iteration crosses the last item, instead StopIteration() is raised to indicate end of iteration.

this would mean they can only be iterated once.

This is also because the iter() method on the iterator returns itself (is essential to remember the state)

This is inturn caught by loop implementations like “for” to terminate the loop

Iterables

Iterables are objects that can be iterated any number of times.

since iteration is already done in iterators, the multiple iterations capability of an iterable is implemented with multiple iterators instantiation.

DRY principle simply reuses(by implementing an \_\_iter\_\_() method to return an iterator instance) the iterator object capabilities for an iterable object since an iterable should do all that an iterator does.

Iterables point to data structures residing in memory and therefore implement methods like a \_\_getitem\_\_() method that can take sequential indexes starting from zero

Iterators objects need not always hold pointers to data structures in memory, for e.g. in the case of generators that are lazily loaded with the next value. This results in some super kool perfomance boosts and is the main reason behind its great popularity.

Mapping is applying a callable (function, class ..) to each element of an iterable

Accumulation is reducing an iterable to a single value. Eg.sum(iterable)

Starmap(operator.mul(list). Use it when we need a function to unpack an iterable.

Reduce(fn, iterable) is function iterable, accumulate(iterable, fn) is iterable function

from itertools import starmap

lst = [[1,2],[2,3],[3,4]]

m = map(lambda x: x[0] \* x[1], lst)

print(m)

for i in m:

print(i)

output: 2

6

12

--

def add(x,y):

return x + y

s = starmap(add, lst)

print(type(s))

for i in s:

print(i)

--

Binary functions are functions that take 2 arguments.

from itertools import accumulate

import operator

# inner accumulate multiples the values as 10,200, 600 and outer accumulater add as 10, 210, 6210

x = list(accumulate(list(accumulate([10,20,30],operator.mul)), operator.add))

print(x)

output: [10, 210, 6210]

--

from itertools import chain

# Chaining 2 lists

x = list(chain([10,20,30],['Jaison ', 'jacob']))

print(x)

output: [10, 20, 30, 'Jaison ', 'jacob']

--

def funa(x):

print('Func A', x)

inp = str(input('Select char'))

def funb(x ):

print('Func B',x)

#a kind of switch statement (case)

b = {

'a': funa,

'b': funb

}

b.get(inp, funa)(20)

#input entered as 'c'

Output: Func A 20

--

#search for files in a directory

import os, glob

os.chdir("/Users/jaisojac/PycharmProjects/2020")

for file in glob.glob("\*.png"):

print ('{}'.format(file))

ddct = defaultdict(int) // s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

d = defaultdict(list)

for k, v in s:

d[k].append(v)

sorted(d.items())

[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

--

Groupby:

import itertools

data = (1,2,2,2,1,3,3,4)

grp = itertools.groupby(data)

for grp\_key, grp\_data in grp:

print(grp\_key, list(grp\_data))

output: 1 [1]

2 [2, 2, 2]

1 [1]

3 [3, 3]

4 [4]

--

from collections import defaultdict

makes = defaultdict(int)

with open("cars\_2014.csv") as f:

next(f)

for row in f:

make, \_ = row.strip('\n').split(',')

makes[make] +=1

for keys, value in makes.items():

print(keys, value)

--

Collection tools: This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, [dict](https://docs.python.org/3/library/stdtypes.html#dict), [list](https://docs.python.org/3/library/stdtypes.html#list), [set](https://docs.python.org/3/library/stdtypes.html#set), and [tuple](https://docs.python.org/3/library/stdtypes.html#tuple).

|  |  |
| --- | --- |
| [namedtuple()](https://docs.python.org/3/library/collections.html#collections.namedtuple) | factory function for creating tuple subclasses with named fields |
| [deque](https://docs.python.org/3/library/collections.html#collections.deque) | list-like container with fast appends and pops on either end (double ended queue) |
| [ChainMap](https://docs.python.org/3/library/collections.html#collections.ChainMap) | dict-like class for creating a single view of multiple mappings |
| [Counter](https://docs.python.org/3/library/collections.html#collections.Counter) | dict subclass for counting hashable objects (counts howmany times an item repeats in an object like list) |
| [OrderedDict](https://docs.python.org/3/library/collections.html#collections.OrderedDict) | dict subclass that remembers the order entries were added |
| [defaultdict](https://docs.python.org/3/library/collections.html#collections.defaultdict) | dict subclass that calls a factory function to supply missing values |
| [UserDict](https://docs.python.org/3/library/collections.html#collections.UserDict) | wrapper around dictionary objects for easier dict subclassing |
| [UserList](https://docs.python.org/3/library/collections.html#collections.UserList) | wrapper around list objects for easier list subclassing |
| [UserString](https://docs.python.org/3/library/collections.html#collections.UserString) | wrapper around string objects for easier string subclassing |

Itertools.Permutations, combinations, combinations\_with\_replace

Context Manager: The circumstances that form the setting for an event, statement, idea and in terms of which it can be fully understood.

In python: the state surrounding the section of code. PEP343

1. Create an object. 2) do some work on that object. 3) clean up that object after we’re done using it.

Classes implement context management protocol by implementing the \_\_enter\_\_ , \_\_exit\_\_ methods.

Common patterns: open – close, change – reset, lock – release, start – stop, enter - exit

The contextManager() creates an instance which is an object returned by the \_\_enter\_\_ method. It is a handle. On \_\_exit\_\_, the handle is released.

If an exception occurs in a with block, \_\_exit\_\_ will have 3 arguments passed. A) type, b) value, c) traceback object. Return True / False to silence an Exception or to propagate it.

Generator function – context manager interaction: the \_\_enter\_\_ method of the contxt manager class implements return next(gen\_func) to yield the value from generator function.

Decorator functions always takes a function as its argument.

--

from contextlib import contextmanager

@contextmanager

def open\_file(fname, mode):

try:

f = open(fname, mode)

print('Opening file ' + f.name)

yield f

finally:

print(f.name + ' closing')

f.close()

with open\_file('words.txt', 'r') as f:

print(f.readlines())

--

Decorator ContextManager is applied to the generator function. Then the generator function is applied in with statement.

Tasking: Concurrent(single processor), parallel(multi processor), cooperative (yields), preemptive(forcefully, by scheduler).

Coroutines are a form of cooperative processing. Generator based coroutines which uses extension of yield. Asyncio.

Coroutines are the generalization of subroutines, which is used in cooperative multitasking environment. The process voluntarily gives up control periodically or when they are idle for other applications to run simultaneously.

Abstract[can be implemented in many ways] data structures: Queue’s [fifo] stacks [lifo].

In a stack list, append element to the end, pop from the end. In a queue list, insert element in front, pop element from end.

Deque, insert and remove elements from both ends.

When calling the generator function inside the calling function like, row = gen(), no code is executed, but its only ‘created’. To run the generator function, call next(row).

Inspect.getgeneratorstate

Generator states: created, running, suspended, closed

Calling function sends echo.send(‘Hello’) // received = yield // ‘yield’ gets ‘hello’, give it back to ‘received’. To send data, yield should be in suspended state.

‘Priming’ generators is calling next(gen) to get the generator to a suspended state and to send a value to it.

Generators has a ‘close()’ method also. GeneratorExit exception

If the exception is thrown in the gen method, None is yielded. Gen.throw()

‘Yield from’ creates a 2 way communication channel. Caller-> delegator -> subgen. Subgen yield to delegator, delegator to caller.

#flatten a list

def flatten(cur\_item):

if isinstance(cur\_item, list):

for i in cur\_item:

#flatten(i)

yield from flatten(i)

else:

#print(cur\_item)

yield cur\_item

#flatten(lst)

for item in flatten(lst):

print(item)

---

Data consume is sometime called ‘sink’

#Pipeline pulling data through generators

import csv

def parse\_data(file):

with open(file) as f:

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

f.seek(0)

next(f)

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

import itertools

'''for rows in itertools.islice(parse\_data('cars.csv'),100):

print(rows)'''

def filter\_data(rows, contains):

for row in rows:

if contains in row[0]:

yield row

'''data = parse\_data('cars.csv')

filtered\_data1 = filter\_data(data,'Chevrolet')

filtered\_data2 = filter\_data(filtered\_data1,'Malibu')'''

'''for row in itertools.islice(filtered\_data1,25): #1st filter

print(row)'''

'''for item in filtered\_data2: # 2nd filter

print(item)'''

def output(f\_name, \*filter\_words):

data = parse\_data(f\_name)

for filter\_word in filter\_words:

data = filter\_data(data, filter\_word) # data is passed with new data

yield from data

results = output('cars.csv','Chevrolet', 'Carlo', 'Landau')

'''for row in itertools.islice(results,5):

print(row)'''

for row in results:

print(row)

--

Python Dictionaries: Associative arrays/hash table/hash maps.

Functional programming using: zip, map, filter, sorted, any, all, chain

Standard library decorators: @lru\_cache, @singledispatch(for generic funcions), @wraps

Dictionary: index, key, value. /0, ‘jaison’, ‘Jaison’/ probing use sequencing of indexes.

Key sharing: Shannon PEP 412. Since Python 3.5, order of keys is guaranteed in dictionaries.

Mutable types(list, set) , set, dictionary (mutable collections) are NOT hashable. Immutable (frozen sets, tuple) are hashable. Functions are immutable and hashable.

Hash keys are to be of immutable type. Mutable means, ‘can be changed’

Dictionary comprehension. Dict.fromkeys(key, value)

Import Pdb is python debugger. Pdb.set\_trace stops at the error

Pip install virtualenv. Virtualenv vnv // source vnv/bin/activate

List methods are mapping, sorting, filtering, zipping

*#implementing function as key in a dictionary***def** add(a,b):  
 **return** a+b  
  
**def** pwr(a):  
 **return** a\*\*2  
  
**def** dvd(a,b):  
 **return** a /b  
  
d = {add: (10,20), pwr: (5,), dvd: (50,5)}  
  
**for** i, args **in** d.items():  
 print(i(\*args))

Output: 30

25

10.0

--

*#dictionary comprehension*key = [**'a'**,**'b'**,**'c'**]  
val = 1,2,3  
  
d = {k:v **for** k, v **in** zip(key,val)}  
  
print(d)

--

List Comprehensions

Dictionary Comprehensions

Set Comprehensions

Generator Comprehensions

frm = dict.fromkeys(**'abc'**,0) – creates a new dictionary  
print(frm)

--

KeyError exception. Dictionary.get(key) – default is None.

Dic.pop(key) – removes and returns the value // dic.popitem() removes the last inserted item LIFO

Val = dict.setdefault(key,value)

--

*# for loop runs till end. If it doesn't break, returns else:***def** func():  
 **for** i **in** range(1,50):  
 **if** i == 100:  
 **return** i  
 **else**:  
 **return 'out of range'**print(func())

Output: out of range

--

Dictionary views: keys, values and items.

Sets has unique items. Sets don’t have order. Dictionaries has insertion order.

Set operations are: union, intersection, difference.

Union or (^ caret) is symmetric difference to apply in the dictionaries.

d. update(d2), d.update(iterable), d.update(kwargs)

To unpack an iterable, use single star \*. To unpack a dictionary, use double star \*\*.

In shallow copy, the container object is new. But the key and value are references to the old one (shared reference).

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

From copy, import deepcopy(d).

Timeit from timeit. Timeit(‘func(arg)’, golbals=globals(),10)

For custom classes, objects id’s are taken as its hash value. If we implement \_\_eq\_\_ method in class, python stops automatic hashing of the object. Use \_\_hash\_\_() return <attribute>

**class** Person():  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 **def** \_\_eq\_\_(self, other):  
 **if** isinstance(other, Person):  
 **return** self.name == other.name  
 **else**:  
 **return False  
  
 def** \_\_hash\_\_(self):  
 **return** hash(self.name)  
  
p1 = Person(**'John'**)  
p2 = Person(**'John'**)  
  
print(p1 **is** p2)  
print(p1 == p2)  
  
d1 = {p1:78}  
*#search dictionary using other object*print(d1[p2])

output: False

True

78

--

Sets are a collection of distinctive objects. It do not have order.

Tuples are immutable equivalent of lists. Frozen sets are immutable equivalents of sets.

Iterable objects will have \_\_iter\_\_ method defined, and \_\_next\_\_ for the iterable object to be iterated. Iterators will have \_\_next\_\_ and \_\_iter\_\_ methods.

Iterators has a sate.

The ‘for’ loop calls the \_\_iter\_\_ on the object, returns self; object inturn call its \_\_next\_\_ to return the value.

Generator function objects automatically implements \_\_iter\_\_ and \_\_next\_\_ methods.

**def** func(start, end):  
 current = start  
 **while** current < end:  
 **yield** current  
 current +=1  
  
num = func(1,10)  
print(**"func "**, dir(func))  
print(**"num "**, dir(func))

--

Set common operations: len, membership – ‘1 in s’, s.add(val), s.remove, s.discard, s.pop, s.clear

Set operations: union, intersection (method – s2 can be a hashable iterable. In & both must be a set), difference, symmetric difference(right associative), containment (strict & non strict). Operators can use only sets. Methods can use iterables.

Two sets are disjoint, if their intersection is empty. Empty sets are falsy.

s.update(), s.intersection\_update(), s.difference\_update(), s.symmetric\_difference\_update()

copy of immutable objects (tuple, frozenset) can point to same object id.

For memorization, functools.lru\_cache.

*#function parameter order. 1)positional, 2) variable arguments, 3) default arguments, 4) keyword arguments***def** func(a,\*args,b=15, \*\*kwargs): *# default should come only after \*variable args*  
 print(**f'positional {**a**}, v\_args {**args**}, defaault {**b**}, kwargs {**kwargs**}'**)  
  
*#variable positional args*p = (23,33,44)  
*#variable kwargs*r = {**'l'**:100, **'o'**:200}  
  
func(10, 17,30,50, x = 10, y = 25,\*p, \*\*r)

Output:

positional 10, v\_args (17, 30, 50, 23, 33, 44), defaault 15, kwargs {'x': 10, 'y': 25, 'l': 100, 'o': 200}

--

Dictionary views: d.keys(), d.values(), d.items(). Views implement \_\_iter\_\_and other protocols.

Changing values during iteration is possible.

As we iterate through the dictionary, as long as not altering the size of the dictionary, we can delete and add items in same number.

*#modifying dictonary on the run (without changing size)*d = {**'a'**:1, **'b'**:2,**'c'**:3}  
  
**for** k, v **in** d.items():  
 **del** d[k]  
 d[k\*2] = **'c'**\*2  
  
print(d)

Serialization: creating a persistant representation of the object. Converting an object to a different representation is often called serializing.

Deserilization: reconstructing the object from the serialized data

MongoDB is a document database stores data in JSON format. NoSQL stores graphs, documents.

Persistance and transformation. // marshmallow – third party library

Pickle module – binary serialization by default. Pickling(serialize) unpickling(deserialize)

‘is’ is identity. == is equality.

Dump (to a file), dumps(to some variable) // load(loads from file), loads(loads from variable)

Regular expressions: regex

Impor re // raw string is ‘r’. pattern.finditer(text)

Metacharacters need to be escaped.

Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns

Except for control characters, (+ ? . \* ^ $ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python −

Sr.No. Pattern & Description

1

^

Matches beginning of line.

2

$

Matches end of line.

3

.

Matches any single character except newline. Using m option allows it to match newline as well.

4

[...]

Matches any single character in brackets.

5

[^...]

Matches any single character not in brackets

6

re\*

Matches 0 or more occurrences of preceding expression.

7

re+

Matches 1 or more occurrence of preceding expression.

8

re?

Matches 0 or 1 occurrence of preceding expression.

9

re{ n}

Matches exactly n number of occurrences of preceding expression.

10

re{ n,}

Matches n or more occurrences of preceding expression.

11

re{ n, m}

Matches at least n and at most m occurrences of preceding expression.

12

a| b

Matches either a or b.

13

(re)

Groups regular expressions and remembers matched text.

14

(?imx)

Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.

15

(?-imx)

Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.

16

(?: re)

Groups regular expressions without remembering matched text.

17

(?imx: re)

Temporarily toggles on i, m, or x options within parentheses.

18

(?-imx: re)

Temporarily toggles off i, m, or x options within parentheses.

19

(?#...)

Comment.

20

(?= re)

Specifies position using a pattern. Doesn't have a range.

21

(?! re)

Specifies position using pattern negation. Doesn't have a range.

22

(?> re)

Matches independent pattern without backtracking.

23

\w

Matches word characters.

24

\W

Matches nonword characters.

25

\s

Matches whitespace. Equivalent to [\t\n\r\f].

26

\S

Matches nonwhitespace.

27

\d

Matches digits. Equivalent to [0-9].

28

\D

Matches nondigits.

29

\A

Matches beginning of string.

30

\Z

Matches end of string. If a newline exists, it matches just before newline.

31

\z

Matches end of string.

32

\G

Matches point where last match finished.

33

\b

Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.

34

\B

Matches nonword boundaries.

35

\n, \t, etc.

Matches newlines, carriage returns, tabs, etc.

36

\1...\9

Matches nth grouped subexpression.

37

\10

Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

--

Quantifiers: \* zero or more, + 1 or more, ? zero or 1, {3} exact 3 numbers, {3,4} min max range of numbers. Groups are placed inside ‘()’. Use ‘|’ to distinct inside group.

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

Sr.No. Modifier & Description

1

re.I

Performs case-insensitive matching.

2

re.L

Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).

3

re.M

Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).

4

re.S

Makes a period (dot) match any character, including a newline.

5

re.U

Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.

6

re.X

Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

**import** string, re  
  
txt = string.ascii\_lowercase + string.ascii\_uppercase + string.ascii\_letters  
print(txt)  
  
pattern = re.compile(**'lmn'**)  
matches = pattern.finditer(txt)  
  
**for** match **in** matches:  
 print(match)  
print(txt[11:14])

--

Server:Client socket program:

*#server socket programmin!***import** socket  
  
srvr\_skt = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
srvr\_skt.bind((socket.gethostname(), 1235))  
srvr\_skt.listen(5)  
  
**while True**:  
 clnt\_skt, clnt\_addrss = srvr\_skt.accept()  
 print(**f'connection from {**clnt\_addrss**} accepted'**)  
 clnt\_skt.send(bytes(**f'Hi from {**srvr\_skt**} ..'**, **'utf-8'**))

*#Socket client!***import** socket  
clnt\_skt = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
clnt\_skt.connect((socket.gethostname(), 1235))  
  
msg = clnt\_skt.recv(1234)  
clnt\_skt.send(bytes(**'Hi from Client'**, **'utf-8'**))  
print(msg.decode(**'utf-8'**))

--

Import os // os.chdir, os.listdir, os.mkdir, os.makedirs, os.rename, os.getcwd, os.rmdir, os.removedirs, os.stat(<file/dir name>), os.walk(<dir path>) – to traverse through the dirs., os.environ // os.environ.get(‘HOME’), os.path // os.path.join(os.environ.get(‘HOME’), ‘test.txt), (os.path.basename (<path>), os.path.dirname(<path>), os.path.splittext(<path>), os.path.isfile(<path>))

JSON datatypes: string, floats(integers), Boolean, arrays, dictionaries, empty(null), special(finite, infinite)

Json keys must be strings in a dictionary. Json objects using double quotes

Json keys do not guarantee order

Vars(object) – makes a dictionary in Json out of the objects attributes

*#vars***class** A:  
 **def** \_\_init\_\_(self,name, age):  
 self.name = name  
 self.age = age  
  
p = A(**'Jaison'**, 44)  
  
print(vars(p))

Output: {'name': 'Jaison', 'age': 44}

--

Custom class objects can be converted by using a serialize method calling into Json.dumps.

@singledispatch - Python fairly recently added partial support for function overloading in Python 3.4. They did this by adding a neat little decorator to the functools module called singledispatch. This decorator will transform your regular function into a single dispatch generic function. Note however that singledispatch only happens based on the first argument’s type.

--

Json don’t know how to serialise ‘date’ format. The date format is ‘ISO8601’.

YYYY-MM-DDTHH:MM:SS+\_05:00 (year-month-dateTimeHourMinuteSecondsOffset+or-fromUTCtime.

t.strftime("%A, %d. %B %Y %I:%M%p") 'Tuesday, 21. November 2006 04:30PM'

t.strftime("%H:%M:%S %Z") '12:10:30 Europe/Prague'

t.strptime() – coverts a string back to a format, is opposite of t.strftime.

datetime.isoformat()

**from** datetime **import** datetime **as** dt  
  
current = dt.utcnow()  
  
**def** to\_date(date):  
 **return** date.strftime(**'%Y-%m-%dT%H:%M:%S%z'**)  
  
t = to\_date(current)  
print(t)  
print(dt.isoformat(current))

--

Log\_rcd = {‘time’:datetime.utcnow(),’message’:’testing’}

json.dumps(log\_rcd, default=<function that covert date value to string format in log\_rcd>)

JSONEncoder class

Float values can be ‘infinity’, ‘minusinfinity’, ‘nan’

--

**from** datetime **import** datetime  
**import** json  
  
deflt = json.JSONEncoder()  
print(deflt.encode(**'abc'**))  
*#class***class** CustomEncoder(json.JSONEncoder):  
 **def** default(self, o):  
 **if** isinstance(o, datetime):  
 obj = dict(  
 datatype =**'datetime'**,  
 date = o.date().isoformat(),  
 iso = o.isoformat(),  
 time = o.time().isoformat(),  
 year=o.year,  
 month =o.month,  
 day=o.day  
 )  
 **return** obj  
 **else**:  
 **return** super().default(o)  
d = {**'Time'**:datetime.utcnow()}  
  
print(json.dumps(d,cls=CustomEncoder, indent=10, separators=(**';'**, **'\*'**)))

--

Output:

"abc"

{

"Time"\*{

"datatype"\*"datetime";

"date"\*"2020-02-14";

"iso"\*"2020-02-14T06:33:56.980680";

"time"\*"06:33:56.980680";

"year"\*2020;

"month"\*2;

"day"\*14

}

}

--

Object-hook for deserializing a dictionary. Parse\_int, parse\_float, parse\_contstant. Object\_pairs\_hook received a list of tuples.

JSONDecoder class. Decode method to override.

--

**import** json  
**from** pprint **import** pprint  
j = **'''  
 {  
 'a': 100  
 'b': [1,2,3]  
 'c': 'Python'  
 'd':  
 {  
 'a': 4  
 'b': 5.4  
   
 }  
 }  
 '''  
  
class** CustomDecoder(json.JSONDecoder):  
 **def** decode(self, arg):  
 print(**'decode'**, type(arg),arg)  
 **return 'a simple string object'**json.loads(j, cls=CustomDecoder)

--

Output:

decode <class 'str'>

{

'a': 100

'b': [1,2,3]

'c': 'Python'

'd':

{

'a': 4

'b': 5.4

}

}

---

JSON Schema is a grammar language for defining the structure, content, and (to some extent) semantics of JSON objects. It lets you specify metadata (data about data) about what an object's properties mean and what values are valid for those properties.

--

Pip install jsconschema

#JSON schema:

**from** jsonschema **import** validate, Draft4Validator  
**from** jsonschema.exceptions **import** ValidationError  
**from json import JSONDecodeError, loads**  
  
my\_schema = {  
 **"$schema"**: **"http://json-schema.org/draft-04/schema#"**,  
 **"title"**: **"Person"**,  
 **"description"**: **"A person"**,  
 **"type"**: **"object"**,  
 **"properties"**:  
 {  
 **"name"**:  
 {  
 **"description"**: **"A person's name"**,  
 **"type"**: **"string"** },  
 **"age"**:  
 {  
 **"description"**: **"A person's age"**,  
 **"type"**: **"number"**,  
 **"minimum"**: 18,  
 **"maximum"**: 64  
 }  
 },  
 **"required"**: [**"name"**, **"age"**]  
}  
  
p1= **'''  
 {  
 "name": "Jaison", "age":63  
 }  
 '''  
  
‘’’ try**:  
 validate(loads(j\_docs),my\_schema)  
**except** JSONDecodeError **as** ex:  
 print(**f'Decode error: {**ex**}'**)  
**except** ValidationError **as** ex:  
 print(**f'Validation error {**ex**}'**)  
**else**:  
 print(j\_docs)’’’

evaluate = Draft4Validator(my\_schema)  
**for** error **in** evaluate.iter\_errors(loads(j\_docs)):  
 print(error, end=**'\n\_\_\_\_\_\_\n'**)  
  
print(j\_docs)

--

Output: {

"name": "Jaison", "age":63

}

--

pyYAML yaml.safeload(). Yamls can used along with marshmallow.

Serpy is ridiculously fast for object serialization(not for deserialization).

Specialised Dictionaries: defaultdict, ordereddict, counters, chainmap, userdict

Factory functions takes a callable as its argument.

Defaultdict is a subclass of dictionary; means an instance of class ‘dict’

Ordereddict is reversible. D = ordereddict(a=10,b=20) for key in reversed(d) print(key)

ordereddict

multisets are sets with repeated elements. Counter is pythons implementation of multiset

counter has elements.

**from** collections **import** Counter  
  
a= **'The quick brown fox jumped over the lazy dog'**x = Counter(a)  
print(x)  
**for** c **in** x.elements():  
 print(c)

--

These tools and their built-in counterparts also work well with the high-speed functions in the [**operator**](https://docs.python.org/3.1/library/operator.html#module-operator) module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: sum(map(operator.mul, vector1, vector2)).

**Infinite Iterators:**

| **Iterator** | **Arguments** | **Results** | **Example** |
| --- | --- | --- | --- |
| [**count()**](https://docs.python.org/3.1/library/itertools.html#itertools.count) | start, [step] | start, start+step, start+2\*step, ... | count(10) --> 10 11 12 13 14 ... |
| [**cycle()**](https://docs.python.org/3.1/library/itertools.html#itertools.cycle) | p | p0, p1, ... plast, p0, p1, ... | cycle('ABCD') --> A B C D A B C D ... |
| [**repeat()**](https://docs.python.org/3.1/library/itertools.html#itertools.repeat) | elem [,n] | elem, elem, elem, ... endlessly or up to n times | repeat(10, 3) --> 10 10 10 |

**Iterators terminating on the shortest input sequence:**

| **Iterator** | **Arguments** | **Results** | **Example** |
| --- | --- | --- | --- |
| [**chain()**](https://docs.python.org/3.1/library/itertools.html#itertools.chain) | p, q, ... | p0, p1, ... plast, q0, q1, ... | chain('ABC', 'DEF') --> A B C D E F |
| [**compress()**](https://docs.python.org/3.1/library/itertools.html#itertools.compress) | data, selectors | (d[0] if s[0]), (d[1] if s[1]), ... | compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F |
| [**dropwhile()**](https://docs.python.org/3.1/library/itertools.html#itertools.dropwhile) | pred, seq | seq[n], seq[n+1], starting when pred fails | dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1 |
| [**filterfalse()**](https://docs.python.org/3.1/library/itertools.html#itertools.filterfalse) | pred, seq | elements of seq where pred(elem) is False | filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8 |
| [**groupby()**](https://docs.python.org/3.1/library/itertools.html#itertools.groupby) | iterable[, keyfunc] | sub-iterators grouped by value of keyfunc(v) |  |
| [**islice()**](https://docs.python.org/3.1/library/itertools.html#itertools.islice) | seq, [start,] stop [, step] | elements from seq[start:stop:step] | islice('ABCDEFG', 2, None) --> C D E F G |
| [**starmap()**](https://docs.python.org/3.1/library/itertools.html#itertools.starmap) | func, seq | func(\*seq[0]), func(\*seq[1]), ... | starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000 |
| [**takewhile()**](https://docs.python.org/3.1/library/itertools.html#itertools.takewhile) | pred, seq | seq[0], seq[1], until pred fails | takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4 |
| [**tee()**](https://docs.python.org/3.1/library/itertools.html#itertools.tee) | it, n | it1, it2 , ... itn splits one iterator into n |  |
| [**zip\_longest()**](https://docs.python.org/3.1/library/itertools.html#itertools.zip_longest) | p, q, ... | (p[0], q[0]), (p[1], q[1]), ... | zip\_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D- |

**Combinatoric generators:**

| **Iterator** | **Arguments** | **Results** |
| --- | --- | --- |
| [**product()**](https://docs.python.org/3.1/library/itertools.html#itertools.product) | p, q, ... [repeat=1] | cartesian product, equivalent to a nested for-loop |
| [**permutations()**](https://docs.python.org/3.1/library/itertools.html#itertools.permutations) | p[, r] | r-length tuples, all possible orderings, no repeated elements |
| [**combinations()**](https://docs.python.org/3.1/library/itertools.html#itertools.combinations) | p, r | r-length tuples, in sorted order, no repeated elements |
| [**combinations\_with\_replacement()**](https://docs.python.org/3.1/library/itertools.html#itertools.combinations_with_replacement) | p, r | r-length tuples, in sorted order, with repeated elements |
| product('ABCD', repeat=2) |  | AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD |
| permutations('ABCD', 2) |  | AB AC AD BA BC BD CA CB CD DA DB DC |
| combinations('ABCD', 2) |  | AB AC AD BC BD CD |
| combinations\_with\_replacement('ABCD', 2) |  | AA AB AC AD BB BC BD CC CD DD |

Itertools chain iterates over iterables in forloop. Where as, chainmap iterates over a dictionary.

Collections ChainMap is updatable. Chainmap.maps returns a list of maps. Mutation affects only th first child. Chainmaps.parents, new\_child

UserDict is not a subclass of dictionary. New class should inherit UserDict.

For regular expressions: https://regexr.com

Object serialization and deserialization is also called ‘marshalling’

OOPS

Object instantiation and initialization. Data attributes and function attributes. Properties are a bybrid between data and functions. Properties use functions to get access to data.

Objects are containers. Data can be accessed through properties. ‘state’ and ‘behavior’ of an object.

Metaclass creates ’class’ objects. The type of a class is ‘type’

Classes and objects has built-in properties given by python. ‘type’ is a metaclass

The ‘state’ of an object is stored in a dictionary, the \_\_dict\_\_ property of the object, as mappingproxy (is a namespace).

Getattr(object, ‘attribute’ , default), setattr(<>), delattr(<>) // del class.attr

Mutating a class will affect have an effect on its objects

Python is dynamic, can add or remove attributes after the objects creation

Instance attributes are stored in object.\_\_dict\_\_

Method is the object that combine the class instance and function.

**class** A:  
 **pass**a = A()  
*#monkey patching on an instance*a.do = **lambda** self: **f'Hello from do {**self**}'**print(a.do(a))

output: Hello from do <\_\_main\_\_.A object at 0x02C4D028>

--

Add function to an object by using bind method. From types import Methodtype.

Obj.say\_hello = MethodType(<function>, obj)

Bare class attributes accessed through methods are called properties. The ‘property’ class defines properties in python.

Name = property(fget=get\_name, fset = set\_name, fdel= del\_name, doc= ‘Name property’)

**class** A:  
 *""" A class with property """* **def** \_\_init\_\_(self, name):  
 self.\_name = name  
  
 **def** set\_name(self, val):  
 self.\_name = val.strip()  
 print(**'name set'**)  
  
 **def** get\_name(self):  
 **return** self.\_name  
  
 **def** del\_name(self):  
 **del** self.\_name  
 print(**'name removed'**)  
*#property definition* unda = property(fget=get\_name, fset=set\_name, fdel=del\_name, doc=**'name property'**)  
  
a = A(**'Jaiosn'**)  
print(a.\_name)  
a.unda = **'Thumba'**print(a.\_name)  
print(a.unda)

*#help(A)*--

Output: Jaiosn

name set

Thumba

Thumba

--

Property getter, setter, deletter.

**class** A:  
  
 **def** \_\_init\_\_(self, name):  
 self.\_name = name  
  
 @property  
 **def** name(self):  
 print(**'getter called'**)  
 **return** self.\_name  
  
 @name.setter  
 **def** name(self, val):  
 self.\_name = val  
 print(**'setter called'**)  
  
a = A(**'John'**)  
print(a.name)  
a.name = **'Hello'**print(a.name)

output:

getter called

John

setter called

getter called

Hello

--

@property can be applied to computed functions also. Those methods can be accessed like a readonly variable.

Interface of the class ..

Urllib.request.openurl()

**class** A:  
 **def** \_\_init\_\_(self, size):  
 self.size = size  
  
 @property  
 **def** size(self):  
 **return** self.\_size  
 @size.setter  
 **def** size(self, val):  
 self.\_size = val  
 print(**'Attribute set'**)  
  
 @size.deleter  
 **def** size(self):  
 **del** self.\_size  
 print(**'Attribute deleted'**)  
  
a = A(10)  
print(a.size)  
a.size = 20  
print(a.size)  
print(a.\_\_dict\_\_)  
**del** a.\_size  
print(a.\_\_dict\_\_)

--

Output: 10

Attribute set

20

{'\_size': 20}

{}

--

A classmethod is a function, which if it is either called from a class or an instance, is bound to the class.

A staticmethod is a function, which is not bound to any object in particular(class or instance)

Instead of declaring staticmethods inside a class, define function in a module, which can be accessible inside the class.

Pytz is a 3rd party library for managing time in python.

‘class variables’ are declared at the beginning of the class.

Abstract classes are one which can’t be instantiated. Should be inherited.

Functions inside a class body has a module scope.

**def** func():  
 MAJOR = 3  
 MINOR = 4  
 VERSION = **'{}- func {}'**.format(MAJOR, MINOR)  
  
 **class** A:  
 MAJOR = 5  
 MINOR = 6  
 VERSION = **'{}-{}'**.format(MAJOR, MINOR)  
  
 @classmethod  
 **def** abc(cls):  
 print(VERSION)  
 **return** A  
  
cls = func()  
cls.abc()

output: 3- func 4

If a class attribute is not found in the instance, it will be searched in the class.

**import** datetime, pytz  
  
tm = datetime.datetime.now(tz=pytz.timezone(**'Asia/Kolkata'**))  
print(tm)  
**for** zon **in** pytz.all\_timezones:  
 print(zon)

--

Special dunder methods:

\_\_init\_\_ : during instance instantiation

\_\_enter\_\_, \_\_exit\_\_ : context managers ‘with ctx() as context’

\_\_getitem\_\_, \_\_setitem\_\_, \_\_delitem\_\_ : sequence types a[i], a[i:j], del a[i]

\_\_iter\_\_, \_\_next\_\_: iterables and iterators iter() and next()

\_\_len\_\_ for len() // \_\_contains\_\_ : implements ‘in’

\_\_str\_\_, \_\_repr\_\_ : for object description repr(), str()

--

NotImplemented Type a immutable. A placeholder that can be returned from overloaded operators to indicate unsupported operand types.

Return NotImplemented is used to indicate that the method is not implemented.

Numpy uses fixed type, continuous memory, SIMD vector processing. Its is useful in mathematics, (MATLAB), plotting (MATPOLOTLIB), backend(Pandas, connect4, digital photography), machine learning – tensers. Tensor Flow

Np = numpy.array([1,2,3]) // np.ndim, np.dtype, np.itemsize, dtype = int16, np.nbytes, np.shape. Indexing of numpy and list differs.

\_\_matmul\_\_ @ new arith2metic operator for matrix calculation in numpy.

\_\_truediv\_\_, \_\_floordiv\_\_ // \_\_iadd\_\_, \_\_isub\_\_ for +=, -= .. (in-place operators)

\_\_radd\_\_,\_\_rsub\_\_ .. for right handside operand binary operations.

Unary operators: \_\_neg\_\_ -a, \_\_pos\_\_ +a, \_\_abs\_\_ abs(a)

Interpolated f’string’ // in-place operators mutate the list object

Rich comparisons are \_\_lt\_\_, \_\_gt\_\_ …

Functools total\_ordering @total\_ordering

@functools.total\_ordering

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

The class must define one of \_\_lt\_\_(), \_\_le\_\_(), \_\_gt\_\_(), or \_\_ge\_\_(). In addition, the class should supply an \_\_eq\_\_() method.

To hash an object it should be hashable. Should implement \_\_hash\_\_ (usually takes object id) and \_\_eq\_\_ (uses ‘is’ for equality).

*#custom class instance as dictionary key***class** Person:  
 **def** \_\_init\_\_(self, name):  
 self.\_name = name  
  
 @property  
 **def** name(self):  
 **return** self.\_name  
  
 **def** \_\_eq\_\_(self, other):  
 **return** isinstance(other, Person) **and** other.name == self.name  
  
 **def** \_\_hash\_\_(self):  
 **return** hash(self.name)  
  
 **def** \_\_repr\_\_(self):  
 **return f'Person (name = {**self.name**}'**p = Person(**'Jan'**)  
p1 = Person(**'Jan'**)  
p2 = Person(**'Feb'**)  
  
print(**f' if the names are equal:'**, p == p1)  
print(**f' if the memory address are same :'**, p **is** p1)  
print(**'Mem address :'**, id(p), id(p1), id(p2))  
  
k = {p:**'April'**, p2:**'May'**}  
print(k)

Functools.wraps @wraps decorator applies correct name and docstrings to the decorated functions. Function.\_\_name\_\_, function.\_\_docs\_\_

\_\_call\_\_ is implemented to make the object callable.

Python dir() Function

The dir() function returns all properties and methods of the specified object, without the values. This function will return all the properties and methods, even built-in properties which are default for all object.

\_\_del\_\_ is called by python just before the object is destroyed by the GC

Prefer contextmanager to clean up resources

unhandled exception in \_\_del\_\_ is ignored by python. It is send to stderr. Sys.stderr

issubclass(object, parent) checks through the parent hierarchy to obtain the result, like isinstance(obj, class).

Object is the top level class from which all other classes inherit.

Hasattr(object, <attribute>) – to check if an instance has that attribute.

DRY: Don’t repeat yourself – aimed at reducing repetition of information.

Super(). Delegation works its way up until it finds what it needs

\_\_slots\_\_ are used to use special class data structures to store predefined instance attributes. Here, the instance will not have the attributes stored in \_\_dict\_\_. Monkey patching not possible in slots. Slots item has to be an iterable

If delattr decorator is not set on a property attribute, it cannot be deleted from the instance.

Create \_\_dict\_\_ as a member of \_\_slots\_\_ so that object.\_\_dict\_\_ will be created and monkey patching can be done.

**class** Person:  
 *"\_\_slots\_\_"* \_\_slots\_\_ = **'skin\_color'**,**'name'**, **'age'**, **'gender'**, **'\_\_dict\_\_'**,*#dict is added to create object for class instance* **def** \_\_init\_\_(self, name, age, gender, skin\_color=**'brown'**):  
 self.name = name  
 self.age = age  
 self.gender = gender  
 self.skin\_color = skin\_color  
  
 @property  
 **def** color(self):  
 **return** self.skin\_color  
  
 @color.setter  
 **def** color(self, clr):  
 self.skin\_color = clr  
  
 **def** \_\_str\_\_(self):  
 **return f'name = {**self.name**} age = {**self.age**} sex= {**self.gender**} character is: {**self.character**} physiq= {**self.physiq**} skin\_color = {**self.skin\_color**}'  
  
 def** ball(self, name, size, \*, company=**'adidas'** ):  
 print(**'ball name: {}, ball size {}, company: {},'**.format(name, size, company))  
  
  
p = Person(**'Jaison'**,12,**'M'**)  
*#monkey patching on class*Person.character = **'bold'***#monkey patching on object*p.physiq = **'stout'**print(p)  
p.ball(size=10, name=**'cricket'**)  
  
print(Person.\_\_dict\_\_)  
print(p.\_\_dict\_\_)  
p.skin\_color = **'Fair'**print(p.color)  
p.color = **'black'**print(p.color)  
Person.skin\_color = **'red'**print(p.color)  
*#del Person.skin\_color  
#del p.skin\_color*print(Person.color)

output:

name = Jaison age = 12 sex= M character is: bold physiq= stout skin\_color = brown

ball name: cricket, ball size 10, company: adidas,

{'\_\_module\_\_': '\_\_main\_\_', '\_\_slots\_\_': ('skin\_color', 'name', 'age', 'gender', '\_\_dict\_\_'), '\_\_init\_\_': <function Person.\_\_init\_\_ at 0x01079580>, 'color': <property object at 0x010806B8>, '\_\_str\_\_': <function Person.\_\_str\_\_ at 0x010794A8>, 'ball': <function Person.ball at 0x01079460>, 'age': <member 'age' of 'Person' objects>, 'gender': <member 'gender' of 'Person' objects>, 'name': <member 'name' of 'Person' objects>, 'skin\_color': <member 'skin\_color' of 'Person' objects>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Person' objects>, '\_\_doc\_\_': None, 'character': 'bold'}

{'physiq': 'stout'}

Fair

black

red

<property object at 0x010806B8>

--

Data descriptors are simply classes that implements certain methods. \_\_get\_\_, \_\_set\_\_, \_\_delete\_\_, \_\_set\_name\_\_ // (for non-data descriptors, only \_\_\_get\_\_)

Descriptors are the underpinning mechanism for properties, methods, functions and even slots.

Data & non-data descriptors. If it’s a data descriptor, descriptor dictionary overrides instance dictionary to store and retrieve data. Non-data descriptor does the opposite.

Property objects are data descriptors.

If an assignment is done on a non data descriptor, it get stored on the instance dictionary.

from numbers import Integral

class IntegerValue:

def get\_age(self):

return self.\_age

def set\_age(self, value):

self.\_age = value

age = property(get\_age,set\_age)

print(hasattr(age, '\_\_doc\_\_'))

class ValidString:

num = IntegerValue()

num.age = 10

print(num.get\_age())

--

Properties are data descriptors

Sequence types are indexable. To check if an object is indexable, check;

isinstance([‘a’,’b’,’c’,’d’,’e’], collections.abc.Sequence // MutableSequence)

Functions are objects that implements non-data descriptors as protocol. \_\_get\_\_

Class.function.\_\_func\_\_ - returns the object to which the function is bound.

Types.MethodType(<func>, <instance>)

**import** keyword  
print(keyword.kwlist)

['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

**import** keyword, sys  
print(sys.version\_info)

--

Python Collections (Arrays)

There are four collection data types in the Python programming language:

List is a collection which is ordered and changeable. Allows duplicate members.

Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

Set is a collection which is unordered and unindexed. No duplicate members. Once a set is created, you cannot change its items, but you can add new items.

Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

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

\_\_get\_\_ can be called from class, instance. Class returns ‘descriptor instance’, whereas instance returns ‘attribute value’.

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

The class that implements \_\_get\_\_ and \_\_set\_\_ are called descriptor

The globals() function returns a dictionary containing the variables defined in the global namespace. When globals() is called from a function or method, it returns the dictionary representing the global namespace of the module where the function or method is defined, not from where it is called.

*#data descriptor***class** IntegerValue:  
  
 **def** \_\_init\_\_(self):  
 self.values = {}  
  
 **def** \_\_set\_\_(self, instance, value):  
 self.values[instance] = int(value)  
   
 **def** \_\_get\_\_(self, instance, owner):  
 **if** instance **is None**:  
 **return** self  
 **return** self.values.get(instance)  
  
**class** Point2D:  
 x = IntegerValue()  
 y = IntegerValue()  
  
  
p1 = Point2D()  
p2 = Point2D()  
  
p1.x = 10  
p2.x = 20.002  
print(p1.x)  
print(Point2D.x.values)

Output: 10

{<\_\_main\_\_.Point2D object at 0x0173D0B8>: 10, <\_\_main\_\_.Point2D object at 0x0173D1D8>: 20}

Data descriptors always override instance dictionaries.

Non-data descriptors may be overridden by instance dictionaries.

Weak references is a reference to an object, but it will not be counted as a reference by the gc

Import weakref

P1 = Person()

P2 = weakref.ref(P1) – creating a weak reference for p2 to Person class

P3 = p2() – here p3 becomes a strong reference to Person class.

Weakref.getweakrefcount(p1)

List(d.weakrefs())

WeakKeyDictionary(). P1 = Person() // d = WeakKeyDictionary() // d[p1] = p2

**import** ctypes  
ref\_count = ctypes.c\_long.from\_address(id(p1)).value

only custom classes allow weak references.

\_\_set\_name\_\_ is called as soon as the descriptor class is instantiated. Here uses the instances dictionary to store data.

Instance attribute shadows class attribute

For loop implements \_\_iter\_\_ protocol.

Transforming Code into Beautiful, Idiomatic Python // https://www.youtube.com/watch?v=OSGv2VnC0go

names = [**'jaison'**, **'jacob'**, **'sonia'**, **'emil'**, **'esther'**]

print(sorted(color, key=len))

--

META PROGRAMMING:

Meta programming is a programming technique in which computer programs have the ability to treat other programs as their data. Code modify code.

Decorators modify other functions. Descriptors set and get decides how ‘.’ Operator bounds a function.

When writing a library or framework, metaclasses are useful. Application coding not require metaclasses. Don’t abuse metaclass. A new hammer shouldn’t see everything a nail.

Function.\_\_qualname\_\_ returns the function name

#descriptor

**class** Integer\_Value(object):#descriptor class  
 k = -1 *# Integer\_value class variable* **def** \_\_set\_name\_\_(self, owner\_class,name):  
 self.name = name  
   
 **def** \_\_set\_\_(self, instance,prop\_name):  
 **if not** isinstance(prop\_name, int):  
 **raise** TypeError(**"property must be an int"**)  
 instance.\_\_dict\_\_[self.name] = prop\_name  
  
 **def** \_\_get\_\_(self, instance, owner\_class):  
 **return** instance.\_\_dict\_\_.get(self.name, **None**)  
  
**class** Point2D(object):  
 x = Integer\_Value()  
 y = Integer\_Value()  
 m = 0 *#Point2D class variable* **def** \_\_init\_\_(self, x, y, z):*#z is not an object of Integer\_Value class. Its local instance variable* self.x = x  
 self.y = y  
 self.z = z  
  
p = Point2D(10,20,23)  
*#p1 = Point2D(5, 10.5)*print(Integer\_Value.\_\_dict\_\_)  
print(Point2D.\_\_dict\_\_)  
print(p.\_\_dict\_\_)  
print(type(p))  
  
print(p.x, **'... '**, p.y, **'..'**, p.z)

--

*#Object call (instance)***class** A:  
 **def** func(self):  
 print(**"Hello"**)  
 **def** \_\_call\_\_(self, \*args, \*\*kwargs):  
 print(args, kwargs)  
  
a = A()  
a(10,20, j=**'hi'**, k=**' bhavani'**)

--

\_\_new\_\_ is a static method

If we are overriding \_\_new\_\_ in the custom class, the parameter in the new and in the \_\_init\_\_ should match.

--

Def \_\_new\_\_(cls, \*args, \*\*kwargs)

Code..

Instance = super().\_\_new\_\_(cls)

Code..

Return instance

--

#create an instance by calling the object.\_\_new\_\_

P = object.\_\_new\_\_(Person)

Built-in abstract base classes. When inheriting from built-in classes, use \_\_new\_\_ to pass in the init params. Don’t use \_\_init\_\_ for that.

In \_\_new\_\_ can do all the initialization done by \_\_init\_\_.

*#class using \_\_new\_\_ and \_\_init\_\_***class** Parent():  
  
 **def** \_\_new\_\_(cls, name, \*args):  
 print(**'creating new object'**)  
 instance = super().\_\_new\_\_(Parent)  
 instance.name = name *# setting name in new* **return** instance  
  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age *# setting age in init*p = Parent(**'jaison'**, 43)  
  
print(p.name, p.age)  
print(p.\_\_dict\_\_)

--

Classes are instances of type. Type is a class, callable too. Type also inherits from object. Type is used to create other classes.

*#exec*namespace = {}  
exec(**'''  
a = 10  
b = 20  
print(a + b)  
'''**, globals(), namespace)  
  
print(namespace)  
print(**'a' in** namespace)

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

*#construction of a class  
#step1: execute the structure  
#a: declare class name*class\_name = **'Circle'***#b: declare the tuple to occupy the class bases*class\_bases = ()  
*#c: declare a dictionary for class body namespace*class\_dict = {}  
*#d: class body*class\_body = **'''  
def \_\_init\_\_(self, a, b):  
 self.a = a  
 self.b = b  
  
def pprint():  
 print("Hello world")  
'''***#step2:  
#exec*exec(class\_body, globals(), class\_dict)  
*#step:  
#create the class*Circle = type(class\_name,class\_bases, class\_dict)  
*#class instance*c = Circle(10,20)  
print(**'a = '**, c.a,**', b = '**, c.b)  
print(c.\_\_dict\_\_)

Output: a = 10 , b = 20

{'a': 10, 'b': 20}

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

**#meta class creation**

**import** math  
  
**class** CustomClass(type):  
 **def** \_\_new\_\_(cls, name, class\_bases, class\_dict):  
 cls\_obj = super().\_\_new\_\_(cls, name, class\_bases, class\_dict)  
 *#creating a function at class creation* cls\_obj.circ = **lambda** self: 2 \* self.r \* 2  
 print(**f'Metaclass {**cls.\_\_name\_\_**} created'**)  
 **return** cls\_obj  
  
class\_base = ()  
class\_dic = {}  
class\_body = **'''  
def \_\_init\_\_(self, x,y,r):  
 self.x = x  
 self.y = y  
 self.r = r  
   
def area(self):  
 return math.pi \* (self.r \*\*2)  
'''**exec(class\_body, globals(),class\_dic)  
*#class creation. CustomClass is a metaclass*c\_class = CustomClass(**'c\_class'**, class\_base,class\_dic)  
*#instance creation*c = c\_class(2,4,6)  
  
print(c.circ())  
print(c.area())

output: Metaclass CustomClass created

24

113.09733552923255

**import** math  
*#meta class creation***class** CustomClass(type):  
 **def** \_\_new\_\_(cls, name, class\_bases, class\_dict):  
 cls\_obj = super().\_\_new\_\_(cls, name, class\_bases, class\_dict)  
 *#creating a function at class creation* cls\_obj.circ = **lambda** self: 2 \* self.r \* 2  
 print(**f'Class {**cls.\_\_name\_\_**} created'**)  
 **return** cls\_obj

*# creating class using metaclass***class** MyClass(metaclass=CustomClass):  
 **def** \_\_init\_\_(self, x, y, r):  
 self.x = x  
 self.y = y  
 self.r = r  
  
 **def** area(self):  
 **return** math.pi \* (self.r \*\* 2)  
  
m\_class = MyClass(4,5,6)  
print(m\_class.circ())  
print(m\_class.area())

Output: Class CustomClass created

24

113.09733552923255

--

*#instead of meta class, use class decorators to create class attributes  
  
#decorator function taking an argument***def** accounttype(type\_):  
 **def** decor(cls):  
 cls.account\_type = type\_  
 **return** cls  
 **return** decor  
  
*# BankAccount = accounttype(BankAccount)*@accounttype(**'Savings'**)  
**class** BankAccount:  
 **pass**@accounttype(**'Fixed'**)  
**class** BankAccount1:  
 **pass**print(BankAccount.account\_type)  
print(BankAccount1.account\_type)  
  
b1 = BankAccount()  
print(b1.account\_type)  
b1.amount = 10\_000  
  
*#print(BankAccount.\_\_dict\_\_)*print(b1.\_\_dict\_\_)

Output: Savings

Fixed

Savings

{'amount': 10000}

--

*#class decorator injecting a method into a class  
#decorator func***def** hello(cls):  
 *#cls.say = lambda self: f'{self} says hello'  
 #to call from class* cls.say = **lambda** cls: **f'{**cls.\_\_name\_\_**} says hello'  
 return** cls  
  
@hello  
**class** Greet:  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 **def** \_\_str\_\_(self):  
 **return f'My name is: {**self.name**}'**obj = Greet(**'John'**)  
  
print(obj)  
  
*#print(Greet.\_\_dict\_\_)  
#when the class function called from an instance, it turnsout to be a method  
#print(obj.say())  
  
#calling from class*print(Greet.say(Greet))

Output: My name is: John

Greet says hello

--

Classmethod and staticmethod are not functions. They are descriptors. They are not callables.

A callable object is an object that can accept some arguments (also called parameters) and possibly return an object (often a tuple containing multiple objects). A function is the simplest callable object in Python, but there are others, such as classes or certain class instances.

Closures

A closure is a nested function with an after-return access to the data of the outer function, where the nested function is returned by the outer function as a function object. Thus, even when the outer function has finished its execution after being called, the closure function returned by it can refer to the values of the variables that the outer function had when it defined the closure function.

An example:

def adder(outer\_argument): # outer function

def adder\_inner(inner\_argument): # inner function, nested function

return outer\_argument + inner\_argument # Notice outer\_argument

return adder\_inner

add5 = adder(5) # a function that adds 5 to its argument

add7 = adder(7) # a function that adds 7 to its argument

print add5(3) # prints 8

print add7(3) # prints 10

Closures are possible in Python because functions are first-class objects. A function is merely an object of type function. Being an object means it is possible to pass a function object (an uncalled function) around as argument or as return value or to assign another name to the function object. A unique feature that makes closure useful is that the enclosed function may use the names defined in the parent function's scope.

When a class decorator decorates a function, the function become an instance of that class.

*#decorator class***class** Logger:  
 **def** \_\_init\_\_(self,fn):  
 print(**'Logger init called'**)  
 self.fn = fn  
  
 *#making the instance callable* **def** \_\_call\_\_(self, \*args, \*\*kwargs):  
 print(**'Get called'**)  
 **return** self.fn(\*args, \*\*kwargs)  
  
 *#making the class a data descriptor* **def** \_\_get\_\_(self, instance, owner):  
 **if** instance **is None**:  
 **return** self  
 **else**:  
 *#making the function bound to the instance* **return** MethodType(self, instance)  
  
**from** types **import** MethodType  
  
**class** Person:  
 **def** \_\_init\_\_(self, name):  
 print(**'Person init called'**)  
 self.name = name  
  
 @Logger  
 **def** say\_hello(self):  
 **return f'{**self.name**} says hello'**p = Person(**'Jaiosn'**)  
print(p.say\_hello())

Output: Logger init called

Person init called

Get called

Jaiosn says hello

--

We cannot inherit mixed metaclasses in the subclass. Metaclass parameters should be named arguments.

\_\_prepare\_\_ creates the class\_dict for the metaclass. It is also a static method like \_\_new\_\_.

\_\_prepare\_\_ returns a mapping type – dictionary.

When creating a new class, 1st python calls the \_\_prepare\_\_ to create the class dictionary. Then calls the \_\_new\_\_ to create the class instance, populating the dictionary.

*# \_\_prepare\_\_  
#simply use an OrderedDict. Any dict can have used here***from** collections **import** OrderedDict  
  
**class** Meta(type):  
 **def** \_\_prepare\_\_(name, bases):  
 print(**'Meta called'**)  
 d = OrderedDict()  
 d[**'language'**] = **'Python'** d[**'version'**] = 3.2  
 **return** d  
  
**class** MyClass(metaclass=Meta):  
 **pass**print(MyClass.\_\_dict\_\_)  
print(vars(MyClass))  
*#output: same for both..*

--

To find the sub classes of a super class, print(<Base\_Class>.\_\_subclasses\_\_())

For a class creation, - python creates ‘name’, ‘module name’, create ‘bases’ call \_\_prepare\_\_ to create dictionary, executes class ‘body’ within class dictionary namespace, assigns the created class to the symbol ‘class name’.

A ‘class’ is callable – p = Person() by the \_\_call\_\_ in the ‘type’ class.

Python process calls \_\_prepare\_\_ of subclass. Then it calls \_\_call\_\_ of Meta class. 🡪 calls \_\_new\_\_ in sub class, which returns to \_\_call\_\_. 🡪 then call ‘\_\_init\_\_ in subclass. Then return the ‘class'.

To make a customer class object ‘hashable’, if the class implements \_\_eq\_\_, it has to use \_\_hash\_\_ also.

Attribute accessors call \_\_getattribute\_\_. It it fails to find an attribute, python calls \_\_getattr\_\_.

Raises AttributeError.

For an attribute, 1st check class data descriptor(\_\_get\_\_), then class instance, then non-data descriptor, then in instance dictionary. If not found, it uses \_\_getattr\_\_. Can return None.

Overriding protocol methods in (dumder) classes are to be used for accessing with instances. It can be overridden by using super().\_\_getattribute(). In order to access ‘class’ attributes, python uses meta class \_\_getattribute\_\_ and \_getattr\_\_ like methods. To avoid infinite recursion, use super().\_\_getattribute\_\_

If a class attribute has to be looked for, create \_\_getattribute\_\_ and \_\_getattr\_\_ inside **meta** class and call inside subclass. If it is to be searched in instance dictionary, define in super class and call in sub class.

Write accessor : \_\_setattr\_\_.

Enum: when a class inherits from enum, the type of class attributes are enum.

Enumerators are always hashable unless override the ‘eq’ method.

If a class implements \_\_getitem\_\_ method, [] can be used to access the item. Enumerations are callable. Enumerations are iterables. Collections which implements iterable protocol can be ‘listed’

Enumerations are immutable, cannot be subclassed unless it is empty.

Rich comparison operations and mathematical ops cannot be done on enums

200k sloc(source line code) in python. 200 top modules. 165 exceptions. 10 files per package.

Heapq is a module without a class. It’s an array.

Enum.\_\_members\_\_. Enum.\_\_members\_\_.key

**import** enum  
  
**class** Color(enum.Enum):  
 RED = **'red'** BLUE = **'blue'** CYAN = 1  
 ORANGE = 2  
  
print(Color.\_\_members\_\_)  
print(Color.BLUE.value)  
print(type(Color.RED))  
  
print(getattr(Color, **'BLUE'**))  
print(Color[**'CYAN'**] **in** Color)  
  
data = {**'redding'**: **'RED'**, **'yEllow'**: **'ylo'** }  
**import** json  
  
payload = **"""  
{   
 "redding" : "RED",  
 "Status" : "status"  
}  
"""**d1 = json.loads(payload)  
  
print(type(d1))  
  
print(Color[d1[**'redding'**]], **'\n'**, Color[data[**'redding'**]].value)

output: {'RED': <Color.RED: 'red'>, 'BLUE': <Color.BLUE: 'blue'>, 'CYAN': <Color.CYAN: 1>, 'ORANGE': <Color.ORANGE: 2>}

blue

<enum 'Color'>

Color.BLUE

True

<class 'dict'>

Color.RED

Red

A lookup for aliases in enumerations will return the master member. @enum.unique – decorator to raise ValueError if the enumerator contain aliases.

By default, the truthiness of enum members are True. We can override this behavior by implementing \_\_bool\_\_

Def \_\_bool\_\_(self):

Return bool(self.value)

--

Enums are not extensibile inorder to keep its immutability.

MRO(Method resolution order) in multiple inherit including super().\_\_init\_\_()

Enums can also be used in functions

Enum.auto() : calls \_generate\_next\_value\_ from enum class.

**import** enum  
  
**class** MyEnum(enum.Enum):  
 **def** \_generate\_next\_value\_(name, start, count, last\_values):  
 print(**f'name {**name**}, start {**start**}, count{**count**}, last\_value {**last\_values**}'**)  
 **return** 50  
  
 a = enum.auto()  
 b = enum.auto()  
 c = enum.auto()

output: name a, start 1, count0, last\_value []

name b, start 1, count1, last\_value [50]

name c, start 1, count2, last\_value [50, 50]

--

Exception: Exception propagation, handling exception, raising exception, Exception class hierarchy, Custom Exception classes, custom hierarchy, extending functionality.

Try: except: else: finally

Two categories: 1) Compilation (syntax), 2) Execution (value error, key error, stopInteration ….)

BaseException is the exception base class. Refer Python bltin-exception for full list.

SystemExit, GeneratorExit, KeyboardInterrupt, Exception

Exception🡪 ArithmeticError, AttributeError, LookupError, RuntimeError, SyntaxError, TypeError, ValueError.

ArithmeticError 🡪 FloatingPointError, ZeroDivisionError.

LookupError 🡪 KeyError, IndexError.

Broad catch: catching an ‘exception class’ will catch all its ‘sub exceptions’

Exception workflow is exception propagation. ‘Bare’ exception is when the except: clause is left with no exceptions.

Try: ‘guarded code’, except Exception as ex: code to run if that exception specified occurs, finally: code runs irrespective of the exception happened or not. Else: code runs if exception didn’t occur and also only if the ‘except’ clause is specified.

Bare exception handlers are one which do not specify any exception type.

Sys.exc\_info() gives info about current exception. Returns the exception object. Exc\_type, exc\_value, exc\_traceback

Nested try: clause is possible inside any excep, finally or else. New exception can be raised inside excep, finally or else clause.

To re-raise the current exception from the except clause, just use ‘raise’ only.

Raise KeyError() from None // from ex – of the previous exception object.

*#Exception handling a few ways***try**:  
 **try**:  
 a = 10  
 **except** KeyError(**'Custom error'**) **as** ex:  
 print(**'Excption handling'**)  
 **except** ValueError **as** val:  
 print(**'...stmts...'**)  
**except** Exception **as** ex1:  
 **raise** *# re-raising the exception  
  
#custom exception class definition***class** CustomException(Exception):  
 **pass  
  
def** func():  
 x = 10  
 j = x / 0  
 **if** j == 0:  
 *#raising custom exception* **raise** CustomException(**'wrong division'**)  
 **else**:  
 **return** j  
  
**try**:  
 func()  
 *#handling exception***except** (CustomException, ZeroDivisionError):  
 print(**'....'**)

A docstring is a statement if its inside a function or a class.

Exception hierarchy (like Python exception hierarchy) is important while creating customer exceptions.

Exceptionobject.\_\_traceback\_\_

Logging levels: notset, debug, info, warning, error, critical

---

Nested classes:

Python does have support for nested or inner classes. These classes can be used when you do not want to expose the class, and it is closely related to another class.

class Cats:

class \_cat\_iter:

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

self.cats = cats

self.cur = 0

def next(self):

i = self.cur

if i >= len(self.cats):

raise StopIteration

self.cur += 1

return self.cats[i]

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

self.cats = []

def add(self, name):

self.cats.append(name)

return self

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

return Cats.\_cat\_iter(self.cats)

Since the interface to the Cats class did not change, there is no change in the client code.

a = Cats()

a.add('joe').add('jack').add('fink').add('dink')

for c in a:

print c

PyQT:

Qt is set of cross-platform C++ libraries that implement high-level APIs for accessing many aspects of modern desktop and mobile systems. These include location and positioning services, multimedia, NFC and Bluetooth connectivity, a Chromium based web browser, as well as traditional UI development.

PyQt5 is a comprehensive set of Python bindings for Qt v5. It is implemented as more than 35 extension modules and enables Python to be used as an alternative application development language to C++ on all supported platforms including iOS and Android.

PyQt5 may also be embedded in C++ based applications to allow users of those applications to configure or enhance the functionality of those applications.

PyPI · The Python Package Indexpypi.org

The Python Package Index (PyPI) is a repository of software for the Python programming language. PyPI helps you find and install software developed and shared by the Python community.

Regular Expressions

Regular expression engine. Re.match(pattern, text), re.search, re.findall, re.finditer

Always use raw ‘r’ string when defining patterns

Regular expressions are case sensitive

(?i)A – for caseinsensitive search of characher ‘A’ or ‘a’ in the text

A|b|c – for searching any of the character in the text. [abc] also can be used

[^aeiou] – to search any character excluding that in the list. [a-ek-p3-7] – search from ‘a’ to ‘e’, k-p, 3-7

‘.’ Searches for everything except ‘newline \n’

‘inline’ operators are used with ‘group’ expressions. (?# inline comment) (?x) end of line comment

Conditional expressions \b(?P<test>\d)?(?(test)\d{2}|[a-z]{4})\b

Regular expression engines belongs to a class of ‘nondeterministic finite automation’.

5 key points with which the Reg expr engine works: 1) one character at a time. 2) left to right evaluation. 3) greedy, lazy and backtracking. 4) groups, 5) look ahed and look behind

In group index, the whole pattern string is the index[0]. 1st group in the string is index[1].

Non capturing group: (?:) // positive look ahead gives ‘=’ and negative look ahead gives ‘!’ inside the parahesis.

Look behind uses ‘<’.

Robocopy log parser.

Jose

Type hinting. From typing import List, Dict, Union

By default, a function returns ‘None’.

Repl.it

String template:

name = "My name is {}. I am studying Python"

f\_name = name.format('Jaison')

print(f\_name)

output: My name is Jaison. I am studying Python

--

Always runs the nested function first.

In a tuple, can’t append an item. But, se ‘+’ to add item to a tuple.

A = ‘jaison’, ‘jacob’

B = a + ‘sonia’

--

List.join // friends =[‘jaison’, ‘jacob’,’sarah’] print (‘,’.join(friends))

‘set’ uses ‘add’ to add an item to a set. No duplicates, no order.

Set functions: difference, symmetric\_difference, intersection, union.

Dictionaries: create dictionary entries inside a tuple. // create tuple inside list and create dictionary out of it. Dic = dict(list)

Loops in python is to do repeated things a defined number of times.

Destructuring a list:

list\_tpl = (['Jaiosn',45],['Jacob',65],['Sonia',39],['Esther',11],['Emil',8])

for firend in list\_tpl:

print(firend[0] , firend[1], end= ' ')

output: Jaiosn 45 Jacob 65 Sonia 39 Esther 11 Emil 8

--

Dictionary:

dic = {'a':1, 'b':2,'c':3,'d':4}

for i in dic:

print(dic[i], end= ' ')

print('\n')

for i in dic.keys():

print(i, end= ' ')

print('\n')

for i in dic.values():

print(i, end= ' ')

print('\n')

for key, val in dic.items():

print(key, val, end= ' ')

output: 1 2 3 4

a b c d

1 2 3 4

Else runs in forloop only if no break happens.

Prime number: a number that is divisible only by itself and 1 (e.g. 2, 3, 5, 7, 11).

A square root of a number is a value that, when multiplied by itself, gives the number. Example: 4 × 4 = 16, so a square root of 16 is 4. Note that (−4) × (−4) = 16 too, so −4 is also a square root of 16. The symbol is √ which always means the positive square root. Example: √36 = 6 (because 6 x 6 = 36)

import math

def prime\_(number):

''' Finding the square root of a number '''

if number == 1:

return False

if number == 2:

return True

if number > 2 and number % 2 == 0:

return False

num\_sqr = math.floor(math.sqrt(number))

for d in range(3, 1 + num\_sqr, 2):

if number % d == 0:

return False

return True # see how (where) its returned

for n in range(1, 21):

print(n, prime\_(n))

In using comprehensions, for loop is used to create a new list based on the conditions.

1. Loops thru the collection, 2) check the condition of the loop variable, 3) if true, value loop variable is stored in the list.
2. lst = [20,30,21,33,40,50,55,57,59,61]
3. ls = [age for age in lst if age % 2 == 0]
4. print(ls)

output: [20, 30, 40, 50]

--

Nested for also possible inside ‘if’ in comprehension.

A parameter is a variable in a method definition. When a method is called, the arguments are the data you pass into the method's parameters. Parameter is variable in the declaration of function. Argument is the actual value of this variable that gets passed to function.

A higher order function is a fuction that operates on other functions, either by taking a function as its argument, or by returning a function.

A first-class function is one, which can be passed as an argument to another function.

Strip() will remove all the white spaces from the input.

Json allows lists and dictionaries in its object. Not tuples.

A folder is made into a package with modules in it, by creating a ‘\_\_init\_\_.py file inside in it, also to make it compatible with older python versions.

'\_\_main\_\_' is the name of the scope in which top-level code executes. Basically you have two ways of using a Python module: Run it directly as a script, or import it. When a module is run as a script, its \_\_name\_\_ is set to \_\_main\_\_ .

Any class which defines a \_\_next\_\_ method, is an iterator. Its not a generator class.

Timezone aware datetime is called ‘aware’. ‘Naïve’ doesn’t know about timezone.

Downloading a web page and extracting data from it is called web scraping

Global interpreter lock [GIL]

Asynchronous processing is threads. Threadpoolexecutor

Atomic operations cannot be interrupted in the middle.

Priming the generator. Send()

Generators that receive data are called coroutines

‘Await’ is replacing ‘yield from’. Async def greet(g) await g

Pip freeze – lists all the installed apps.

Virtualenv <env\_name> --python=<python3.8.0> // to create a virtual environment for the app to run

Pipenv // pip3.7 install pipenv // pip3.7 –upgrade pip // pipfile.lcok

Pipfile specifies the ‘libraries’ required, piplock specifies the ‘versions’ required.

Pipenv folder is created somewhere else. ‘Pipenv run python’

Using Pipenv https://www.udemy.com/course/the-complete-python-course/learn/lecture/9552016?start=0#overview

Virtual Environments

When you start a new project, you’ll want it to be completely separate from other projects in

terms of the libraries it has installed.

If you start working today on a project using version x of a library, and in two years time

you work on another project using version y of a library, you may not want to update. If

version y includes some breaking changes from version x , your old project would stop

working.

This is why it’s interesting have separate environments for each project, so that the changes

and updates to libraries one depends on does no affect other projects.

Installing new libraries

Python installs new packages using pip . You can install packages in your system Python

installation easily using pip :

The Complete Python Course — Udemy

Page 2

Using Pipenv

pip install requests

This will install the requests library in the Python that is linked to pip . By default on Mac

OS X, this is a Python version that comes installed in the computer, Python 2.7.

If you’ve installed another Python version, it’ll be that version.

If you have multiple Python versions installed, you’ll have to be more specific:

• python will refer to the first Python version you installed.

• python3 will refer to the version of Python 3.x that you have installed, if you have

only one version of Python 3 and one of Python 2.

• python3.7 will refer to Python3.7 if you have two or more versions of Python 3 installed.

Virtual environments

In order to separate projects into different environments with different libraries, we use an

extremely popular library called virtualenv .

You can create a new virtual environment—which is just a copy of the Python version you

use to create it—using one of the following commands (depending on the Python program as

above):

virtualenv venv

virtualenv venv --python=python3

virtualenv venv --python=python3.7

This creates a new folder called venv inside your current folder, which is a copy of Python

without any libraries installed.

Now we can activate this virtual environment, which then means any libraries we install using

pip will go in the virtual environment; and when we run the python program that’ll be the

version installed in the virtual environment:

source venv/bin/activate

Page 3

https://tecladocode.com/blog

The Complete Python Course — Udemy

Or on Windows:

.\venv\Scripts\activate.bat

Pipenv

Pipenv brings together pip , virtualenv , and a dependency management system.

Instead of creating a virtualenv, activating it, and installing libraries, we can just install a library using Pipenv. Make sure you don’t have a virtual environment created locally by deleting it if you have created it.

rm -rf venv

Then, let’s install Pipenv. You’ll have to use:

• pip if you only have a single version of Python installed;

• pip3 if you have a version of Python3 and one version of Python2 installed; or

• pip3.7 if you have two or more versions of Python3 (and one of them is Python3.7).

pip install pipenv

Then, we can install the first library using Pipenv:

pipenv install requests

This creates a new virtual environment (in a different folder in your computer) and installs the

library in it.

In addition, it also creates two new files in your project folder:

• Pipfile ; and

• Pipfile.lock .

Page 4

Using Pipenv

The Pipfile file

After installing the requests library as above, the Pipenv file will look like this:

[[source]]

url = “https://pypi.python.org/simple”

verify\_ssl = true

name = “pypi”

[packages]

requests = “\*”

[dev-packages]

This file is divided into sections: [[source]] , [packages] , and [dev-packages] .

The most interest one here is the [packages] section, which has all the dependencies of our

project—everything our project depends on. These libraries here are installed from the url

defined in the [[source]] section.

Here we specify the requests library, and the version we want is \* ; any version is acceptable. By default, we will always install the latest version of any package.

However, when you share the project with someone else, you’ll share both the Pipfile

(which tells them what to install), and the Pipfile.lock (which tells them which versions of

things to install.

Let’s talk about Pipfile.lock !

The Pipfile.lock file

This file looks more complicated. Something like this:

{

“\_meta”: {

“hash”: {

“sha256”:

“a0e63f8a0d1e3df046dc19b3ffbaaedfa151afc12af5a5b960ae7393952f8679”

},

Page 5

https://tecladocode.com/blog

The Complete Python Course — Udemy

“host-environment-markers”: {

“implementation\_name”: “cpython”,

“implementation\_version”: “3.7.0b1”,

“os\_name”: “posix”,

“platform\_machine”: “x86\_64”,

“platform\_python\_implementation”: “CPython”,

“platform\_release”: “17.4.0”,

“platform\_system”: “Darwin”,

“platform\_version”: “Darwin Kernel Version 17.4.0: Sun Dec 17 09:19:54

PST 2017; root:xnu-4570.41.2~1/RELEASE\_X86\_64”,

“python\_full\_version”: “3.7.0b1”,

“python\_version”: “3.7”,

“sys\_platform”: “darwin”

},

“pipfile-spec”: 6,

“requires”: {},

“sources”: [

{

“name”: “pypi”,

“url”: “https://pypi.python.org/simple”,

“verify\_ssl”: true

}

]

},

“default”: {

“certifi”: {

“hashes”: [

“sha256:14131608ad2fd56836d33a71ee60fa1c82bc9d2c8d98b7bdbc631fe1b3cd1296”,

“sha256:edbc3f203427eef571f79a7692bb160a2b0f7ccaa31953e99bd17e307cf63f7d”

],

“version”: “==2018.1.18”

},

“chardet”: {

“hashes”: [

“sha256:fc323ffcaeaed0e0a02bf4d117757b98aed530d9ed4531e3e15460124c106691”,

“sha256:84ab92ed1c4d4f16916e05906b6b75a6c0fb5db821cc65e70cbd64a3e2a5eaae”

],

“version”: “==3.0.4”

},

“idna”: {

“hashes”: [

“sha256:8c7309c718f94b3a625cb648ace320157ad16ff131ae0af362c9f21b80ef6ec4”,

“sha256:2c6a5de3089009e3da7c5dde64a141dbc8551d5b7f6cf4ed7c2568d0cc520a8f”

],

“version”: “==2.6”

},

“requests”: {

“hashes”: [

“sha256:6a1b267aa90cac58ac3a765d067950e7dbbf75b1da07e895d1f594193a40a38b”,

“sha256:9c443e7324ba5b85070c4a818ade28bfabedf16ea10206da1132edaa6dda237e”

],

“version”: “==2.18.4”

},

“urllib3”: {

“hashes”: [

“sha256:06330f386d6e4b195fbfc736b297f58c5a892e4440e54d294d7004e3a9bbea1b”,

Page 6

Using Pipenv

“sha256:cc44da8e1145637334317feebd728bd869a35285b93cbb4cca2577da7e62db4f”

],

“version”: “==1.22”

}

},

“develop”: {}

}

This defines a lot more information, and this file is used to “lock” the versions of the libraries

you have installed to a specific point in time.

If you give the Pipfile and the Pipfile.lock to someone else, they’ll end up with exactly

the same versions of libraries as you.

Each library in the Pipfile.lock comes with two things:

• version ; and

• hashes .

The version is the version of the package you want. Normally this is the only thing you

need in order to retrieve the correct package from the source url (defined in the Pipfile ).

However, it is possible that someone might swap out a version of a library for something

else—a security issue to say the least!

That is where the hashes come into play. The hashes are numbers generated from the library

itself. If the library changes at all, the numbers generated will change. So if you try to install

the same version as earlier on but the hashes are different, you’ll get an error. Pipenv will

make sure it’s all safe!

Security is a difficult topic, but the entire Python community is agreeing that having these

lock files is a great idea. Other programming communities, like JavaScript, is also using similar

techniques for securing their dependency management. It’s the way to go!

— Jose and the Teclado team

API’s are programs that receives data from client and returns data.

Openexchangerates.org for api id.

TTL (time to live cache). From cachetools import cached, TTLCache

@cached(cache=TTLCache(maxsize=2, ttl=900))

Functools.wraps on the decorator wrapper function keep the name and docstrings of the original function intact.

Decorators use positional \*args and keyword args \*\*kwargs to pass arguments to parameters.

Abstract methods has to be implemented by the inheriting class.

No object can be created for the class defining abstract method.

From abc import ABCMeta, abstractmethod. @abstractmethod is a decorator

Binary search to find a number in the ordered list by searching the halfth value. Biary tree

Flask. Mkdir first\_flask // cd first\_flask // Pip install pipenv // pipenv install Flask //

Render\_template jinja2

From screen reader // query string parameter

Psycopg2 library used to interact with postgresql database. Aiopg SQLAlchemy

Flask is popular because of its extensions enhance power.

Communications library: smtplib, mailgun, twilio( for chat apps)

GUi: Kivy,Tkinter

Data science: Numpy, SciPy, Pandas, Matplotlib

Comuter vision: OpenCV, SimpleCV

Devlopment tool: pylint(check python file for errors), flake8, yapf, black

IDE’s: Pycharm, Visual Studio Code, Atom, Sublime.

Gmail servers are smtp server.

Slint for javascript. When working on javascript, prefereably use Visual studio code as IDE.

Binary tree traversal: inorder -> go left first (useful for sorting the tree) , preorder(useful for saving and reconstructing later), postorder (useful for deleting items)

Xcross

import unittest – python unittest module. Unittest.TestCase // https://www.youtube.com/watch?v=1Lfv5tUGsn8

**import** unittest  
**from** math **import** pi  
*#function to be tested***from** circle **import** circle\_area  
  
**class** Test\_Circle(unittest.TestCase):  
 **def** test\_equal(self):  
 self.assertAlmostEqual(circle\_area(0),0)  
 self.assertAlmostEqual(circle\_area(1), pi)  
 self.assertAlmostEqual(circle\_area(2.1), pi \* 2.1 \*\*2)  
  
 **def** neg\_value(self):  
 self.assertRaises(ValueError, circle\_area, -2)

run as python –m unittest test\_circle.py // setUp, tearDown

setUpClass // tearDownClass @classmethod

CURSES

1. Installation

For Unix Environments (linux distros and macOS), curses is already available in the Python Standard Library. No need to install anything.

The Windows version of Python doesn’t include the curses module. You can install windows-curses package to to use curses on Windows. Install it using this simple pip command:

pip install windows-curser

In [1]:

import curses

2. Initializing a curses application

curses.initscr()

Initialize the library. Return a window object which represents the whole screen.

curses.noecho()

Leave echo mode. Echoing of input characters is turned off.

curses.cbreak()

Input buffering is turned off. Characters are available to be read one by one. User does not require to press Enter key every time to send an input. This helps application to react to keys instantly

stdscr.keypad(True)

Make curses interpret special keys like Page Up, Home, Right, Left, Up, Down, etc as special values instead of multibyte escape sequences. For example, Left button can be interpreted as curses.KEY\_LEFT value.

3. Closing a curses application

Before closing the application, we need to reverse the curses-friendly terminal settings.

curses.nocbreak()

stdscr.keypad(False)

curses.echo()

And finally, we can close the application.

curses.endwin()

4. Some other utility functions

stdscr.addstr(y,x,str)

Write str from position (x,y) on the screen.

stdscr.clear()

Clears the screen.

stdscr.refresh()

When you call a method to display or erase text, the effect doesn’t immediately show up on the display. Calling refresh() method of window object will update the screen.

Example 1

import time

import curses

# initialize application

stdscr = curses.initscr()

# tweak terminal settings

curses.noecho()

curses.cbreak()

stdscr.keypad(True)

curses.curs\_set(0)

# write something on the screen

stdscr.addstr(5, 10, "Hello, world!")

# update the screen

stdscr.refresh()

# wait for 3 seconds

time.sleep(3)

# clear the screen

stdscr.clear()

# reverse terminal settings

curses.nocbreak()

stdscr.keypad(False)

curses.echo()

# close the application

curses.endwin()

5. Wrapping up the application

A common problem when debugging a curses application is to get your terminal messed up when the application dies without restoring the terminal to its previous state. In Python this commonly happens when your code is buggy and raises an uncaught exception. Keys are no longer echoed to the screen when you type them, for example, which makes using the shell difficult.

We can avoid these complications and make debugging much easier by using curses.wrapper() function.

curses.wrapper(func)

Initialize curses and call another callable object, func, which should be the rest of your curses-using application.

If the application raises an exception, wrapper will restore the terminal to a sane state before re-raising the exception and generating a traceback.

Before calling func, wrapper()

turns on cbreak mode,

turns off echo,

enables the terminal keypad, and

initializes colors if the terminal has color support.

On exit (whether normally or by exception), it will

restore cooked mode,

turn on echo, and

disable the terminal keypad.

Example 2

import time

import curses

def main(stdscr):

# disable cursor blinking

curses.curs\_set(0)

# write something on the screen

stdscr.addstr(5, 10, "Hello, world!")

# update the screen

stdscr.refresh()

# wait for 3 seconds

time.sleep(3)

# clear the screen

stdscr.clear()

curses.wrapper(main)

--