We donot use semicolon in Python

Numbers

12/3 🡪 4.0

If we use / symbol in any manner the result will be in floated decimal form

To avoid this decimal form we use 12//3 to get the output 4

If more than one operator is used then it will use maths rule

Modulus operator % to give remainder

Print

to print the context on the screen syntax: print()

print(“ ”) and print(‘ ’) both work in same manner another way to use print statement is print(“it\’s”)

print(first\_value,second\_value,sep=”separator”) if print has multiple arguments then all are printed out in a same line by a separator thing defined as last argument although the defaule separator is a space

Variables

Cann’t use numbers as variable name statups

Use alphabets numbers underscores

Comments

#Single line

“””

Multiline

“””

Input

Syntax: variable\_name=input(“Enter any line to show something: ”)

The input will be taken as string to get integer from it we use it like

Syntax: variable\_name=int(input(“Enter any line to show something: ”))

String formatting

1.Syntax: variable\_name=” Enter any string”+ another\_variable\_name\_which\_store\_string

2.Syntax: variable\_name=f”Enter the string { another\_variable\_name\_which\_store\_string }”

3.Syntax: variable\_name=”enter the string {}”

Variable\_name.format(another\_variable\_name\_which\_store\_string)

Boolean and Comparing

It can be either True or False…but not both at the same time but make sure the T and F are capital respectively.

not True means False

not False means True

not not True means True

and is used between two compare between two values such that if first one is False the result is False however if first one is True the second value is the result whatsoever the second value is.

Syntax: print (20 and 18)🡪18

or is used between two values such that if first one is true then the result is true and the if the first one is false the second one is the result

Syntax: print(20 or 18)🡪20

There is a great difference between their results…..

List ,Tuple ,Sets and Dictionary

Syntax of List: [],list are ordered if we print them they appear as they are typed, it supports .append() to add other elements, indexing is possible,it supports len(),it supports sum()

“the\_separator”.join(tuple\_var) will give the entire list elements as separated by each other by the separator

Syntax of Tuple:(),tuple are ordered if we print them they appear as they are typed, they do not support .append or .add it require the format variable\_name\_of\_tuple = variable\_name\_of\_tuple + (element, this comma is used to differentiate between tuple and other element ) , Indexing is possible, it supports len() and sum() function

Iterating over tuples is done by destructing syntax this is done like:

var1=ghg,jhj

var2,var3=var1 #now var2=ghg and var3=jhj

Syntax of Set:{} ,set are unordered they do not print as typed, it supports .add function to add other elements, indexing is not possible ,len() and sum() function are supported

Syntax of Dictionary{:},they are unordered they do not print as typed, indexing is not possible to do indexing try a dictionary into a list,len() function is supported, sum function is not supported

Iterating over dictionary is quite complex but is done using this syntax:

dict={key:value}

for ghg in dict:

print(ghg) #print out only key

for ghg in dict.values() :

print(ghg) #print out only values

for ghg,jhj in dict.items():

#now ghg=keys and jhj is values

If Loop and While Loop:

Syntax: if <condition> [<is True>,<in dictionary\_name>]:

<block of if statement>

elif <condition> [<is not True>]:

<block of elif statement>

Else:

<block of else statement>

In the if statement we just have to use condition equated to Boolean true or false there are various bool() function which are equal to False eg.

1. bool(0)
2. bool(0.0)
3. bool('') # empty string
4. bool(None)
5. bool([]) # empty list

Beside them other bool(“String”) or any other are considered True

Syntax: while <condition>[<is True>]:

<statement of while blcok>

It must be provided with a termination condition in the statement block otherwise an infinite loop will start

For loop:

This is little complicated we have to use a variable without any kind of declaration

Synatx: for <variable which takes value automatically> in <any tuple,list,sictionary>:

<statement of if block>

If there are 3 elements in any data element it will run 3 times

To run the loop as required we use a function called **range()**. Remember that it is a list of numbers

Eg: range(10)=[0,1,2,3,4,5,6,7,8,9] , range(0,101)=[0,1,2,3,4…….,99,100]

But range(10)🡪range(0,10)🡪range(0,10)

There is a little more complication while using dictionary items using for

Eg; for name in dictionary\_name:

print(name)🡪it will print key values

liawas asking for keys and values both but we need to use .items() function to access the items in a dictionary

Eg; for name,days in dictionary\_name.items():

print(name,days)🡪print out both keys and values

Note: the data structures used in loops are called iterables

break and continue keyword

break keyword is used to stop the flow further of a loop

continue keyword is used to move back to the loop and continue the loop further without proceeding the down statements of the keyword continue

For else loop:

This loop simply runs the for loop and if for loop is interrupted then else will run otherwise not

List slicing

List slicing is used to print only certain elements of a list in continuity by using [index\_start : index\_end (which is not included)] to move to the end just leave the index\_end or to begin for the first element just leave the index\_start like [index\_start:] and [:index\_end] respectively.

To indicate the last element -1 can be used but this will be illegal [-1:-3]as a list will not be procceded form right to left it is always preceeded from left to right.

Comprehension

It is used to reduce the stress of coding, it is used to easily define the modification of list, set and dictionary

Syntax: list\_variabe=[x\*2 for x in another\_variabe [<is x in another\_variable>]]

To modify the result in uppercase use x.capitalize() or x.upper() and to result the lowercase use x.lower()

For set comprehension just switch the [] into {}

For dictionary comprehension built two list with keys and values separately and use the built in function dict(zip(key \_list, value\_list))

Functions, Their 4 types and return values

Syntax: def function\_name([<parameters>]):

<block of function>

return None(default)

any variable defined inside a function does not have scope out of the function, the return value is by default None which can be changed to almost anything, function are called by simply writing their name() like this, the arguments and parameters are respectively the providing end and receiving end.

If we print a function by its name() like this it will display all the print, input and return statements

Anonymous function: lambda x,y: x+y (the x and y are arguments and x+y is a return statement), to use lambda function Syntax: (lambda x,y:x+y)(value\_of\_x,value\_of\_y)

First class function: which are used as parameters for other functions but when passing the functions as arguments do not use() because we don’t want to initiate that function we just want to use that as a reference

Higher order functions: which use other functions as parameters

While using other functions as parameters just simply type their name and do not use() brackets

Default parameters:

The default parameters once assigned value cannot be changed again except the case when calling the function. The default arguments should be called in a proper manner as if the first argument is called with the name of the parameter the second should also be called like that otherwise an error will initiate.

Object Oriented Programming with Python:

Why we use class because inside a dictionary we cann’t use a function as a value which can modify the values in the same dictionary.

\_\_init\_\_ is called dundet variable since it consists double underscore

Self keyword is used to indicate the object\_name as it looks professional we can use other names instead of self

Creating class:

Syntax; class <class\_name\_capital\_first\_word>:

def \_\_init\_\_<self,first\_argument,second\_argument>:

self.property\_1=first\_argument

self.property\_2=second\_argument

def [<other method(self)>]:

<block of other methods>

Creating object:

object\_name=Class\_name(<first\_argument,second\_argument>)

How to use these ojects?

With class name: print(Class\_name.method\_name(object\_name))

Without class name: print(object\_name.method\_name())

With only object : print(object\_name.property\_1)

Magic Functions

These are special functions or dunder methods use to access the methods of a class

To know the class of an iterable try this : print(variable\_name\_of\_iterable.\_\_class\_\_)🡪<class ‘type of that iterable’>

1.\_\_len\_\_(self):

return len(self.property\_name)

but how to call this function? By simply print(len(object\_name))

2.\_\_getitem\_\_(self,i):

return self.property\_name[i]

but how to call this function? By simply print(object\_name[value\_of\_i])

3.to use for in a class just need the above two property :\_\_len\_\_ and \_\_getitem\_\_

but how to call this function? By simply for any\_variable\_name in object\_name:

print(any\_variable\_name)

4.\_\_repr\_\_(self):such that with that thing you can recreate the object fully

but how to call this function? By simply print(repr(object\_name))

5.\_\_str\_\_(self):To print out the object to the user

but how to call this function? By simply print(object\_name) or print(str(object\_name))

Inheritance

This is used to inherit or access the parent class functions

class child\_class\_name(parent\_class\_name):

def \_\_init\_\_(self,then same parameters as in parent class, parameters in child class ):

super().\_\_init\_\_(parameters not defined in child class but defined in parent class)

self.property1=child\_class\_variable

but how to call this ? By simply object\_name=child\_class\_name(value\_of\_all\_parameters)

by using the super function the child class can access all the property and functions in the parent class but parent class cann,t do so. Both classes now use a common object as self.

@=decorator

@property :to create a method or so-called function into a property just simply remove the () thing from the method and type @property above the syntax or where the method is defines.

Limitation 1: this is used only for that function which have only self as argument and do not take any other argument than self

Limitation 2: we can use this @property method only when the method just returns a value and doesn’t do any action.

@classmethod: takes class as the first argument, Class\_name.function\_name(object\_name),

Syntax:

@classmethod

def function\_name(cls)

it is used to change the class name simply whenever needed since it uses class name as it’s first argument

@staticmethod: takes nothing as first argument ,object\_name.function\_name

Synatx:

@staticmethod

def function\_name()

It is used by removing the self from its parameters so there is no need to create an object to access any method just simply type Class\_name.method\_name(parametrs )

Abstraction is used for data hiding

from abc import ABCMeta, abstractmethod

The class which is to be abstracted should have syntax like class class\_name(metaclass=ABCMeta) and the function of that class which is to be abstract will have @abstractmethod on it.

Errors handling

Type of errors:

1.Index error: list index out of range

2.Key error: key is misused

3.Name error: variable not defined

4.Attribute error: list object has no attribute intersection

5.NotImplemented error: to raise an error

6.Runtime error: ?

7.Syntax error: typing python code error

8.Indentation error: when correct indentations are not provided

9.Tab error: either use tab or 4 spaces

10.Type error: adding two different types like ‘str’ and ‘int’

11.Value error :converting a wrong data type to another

12.Import error :circular imports

13.Depreciation Warning :when there is a better way of doing that thing

isinstance(object\_name,Class\_name) is used to check if the object is the object of the class mentioned ,the result is in Boolean form

raise NameOfTheErrorInCamelCasing(“message to display”),no need to add return in front of raise

how to create our own error? by subclassing the Exception class

Syntax: def Name\_of\_the\_error(Exception):

pass

then to use this error just raise Name\_of\_the\_error(“message to display”)

pass is a keyword which does nothing but it is used under the function so that the function does not have any indentation error and performs no action

docstring is the “”” here the docstring goes“””,it is just like comment

Exceptional handling:

Syntax:

try:

<block of try>

except <type\_of\_error>:

<block of except>

[<except < type\_of\_error > >]

<block of it if any>

else:

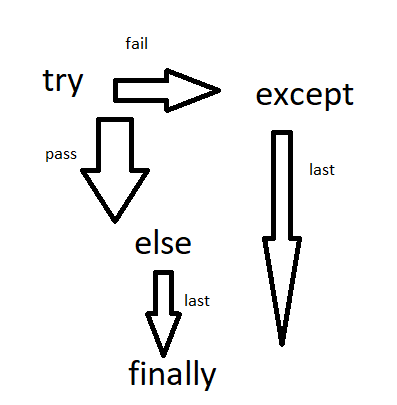
<block of else>

finally:

<block of final statement>

Working of the exceptional handling?

First of all the try goes if it gives any error then the type of error is searched in the exceptions if no exception found then finally block goes. If the try block works well then it goes directly to the else block and then finally



File Handling:

How to do file handling? First of all open the file then read the things you want from the file to a variable which exists even after the file is closed

Syntax to read a file: var\_any=open(‘file\_name.extension’,’r’)

another\_var\_which\_exists=var\_any.read()

var\_any.close()

Syntax to write a file: var\_any=open(‘file\_name.extension’,’w’)

Var\_any.write(the\_thing\_you\_want\_to\_write) #replace the content of file

var\_any.close()

read(): It will read the entire content of the file in strings

readline(): It will convert the entire file into single lines and store them in lists but one by one ,we need to call this function as many times and it changes its values from one line to another line

readlines(): It will convert the entire lines in the file to a single list with each line as a single element of that list

split(‘any\_symbol’): this works only on strings and convert the string into a list with each element as differentiated by other through the symbol between the ‘ ’. It do not work with lists

strip(): It eliminates the empty spaces from the each element in the list it works with list

title(): It is used to capitalize only the first letter

Why to close the file ? because we can open only limited number of files at a time in our system

Csv : comma separated values

Read csv values? By using split(‘,’) function in the entire list which contains the elements as single-single line and using a for loop for each element in that list

Json: Javascript object notation

It is a special kind of file used in javascript which uses dictionary as objects and it always uses “”(double quotes) with it.

Format:{ “name”=[{1 object},{ 2 object },,,,,,,,{ n object }]}

Syntax to read json file: import json

var\_any =open(‘file\_name\_which\_load.extension’,’r’)

other\_var=json.load(var\_any)

var\_any.close()

Syntax to write on a file: import json

var\_any =open(‘file\_name\_where\_save.extension’,’w’)

json.dump(data\_to\_pass , var\_any)

var\_any.close()

we can use dumps where ‘s’ stands for the string and we can pass string in place of data\_to\_pass

Csv to Json: we can do so by converting each item into a dictionary and append the dictionary into a list

Importing Files:

How to convert a directory into package? By creating a python file with name \_\_init\_\_.py

If we use print(\_\_name\_\_) in a file it print out the \_\_main\_\_ file when it is run otherwise the location of other files if they are imported in this file and those files contain print(\_\_name\_\_) too in them

When a file is imported all the functions defined in that file are now defined in this file and all other statements are also printed if we donnot want to print those statements then just put an if statement and check the file is not \_\_main\_\_

From parent: from ..location import file\_name

From child: from .location import file\_name

But it is better to locate the whole address like this from location import file\_name OR from location.file\_name import function\_name

Context manager: the context manager use with keyword and close the opened file automatically after the block ends but make sure to use semicolon(:)

Syntax: with open(‘file\_name.extension’) as var\_any:

Other\_var=json.load(var\_any)

Circular error i.e. import error which happen when we import the file from each other

Databases:

Connect database: import sqlite3

var=sqlite3.connect(‘name\_of\_database.db’)

var.close()

Syntax: #importing the sqlite3.py

import sqlite3

#connecting to the databse

var=sqlite3.connect(‘name\_of\_database.db’)

var2=var.cursor()

#this section executes a query in the database

var2.execute(‘YOUR SQL QUERY HERE’)

#this section save the changes and close the database

var.commit()

var.close()

commit(): is used before closing the database as it save the changes to the database

cursor(): is used to create an event for n events n cursor are required

Some sql queries:

CREATE TABLE table\_name (column\_name1 type1 [<primary\_key>],…column\_name2 type2)

‘INSERT INTO table\_name VALUES(?,?)’,(var\_name1,var\_name2)

SELECT \* FROM table\_name

SELECT column\_name FROM table\_name

UPDATE table\_name SET [<changes you need>]

DELETE FROM table\_name

UPDATE TABLE table\_name SET [<changes required>} WHERE [<any hint>]

SELECT \* FROM table\_name WHERE [<any hints>]

DELETE FROM table\_name WHERE [any hints]

SELECT \* FROM table\_name ORDER BY column\_name (default:ASC)

SELECT \* FROM table\_name ORDER BY column\_name DESC

SELECT \* FROM table\_name ORDER BY column\_name DESC LIMIT [integer no of rows required]

How to make own Context managers?

Syntax: import sqlite3

Class class\_name:

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

self.connection=None

self.host=host

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

self.connection=sqlite3.connection(self.host)

retrun self.connection

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

self.connection.commit()

self.connection.commit()

if there is any error while exit the context manager then any one of the value of exc in parameter is more than 0 so in such case we can use if and donot include commit function in but only close and in else part both so no error will be evolve

Type hinting in python?

It is ignored when program is run, it is used for developing only, is used to tell the program about the various variable types or parameter types in a function or tells the return value of a function

from typing import List,Dict,Union (new in python 3.7)

It can be done so by def func\_name(parameter\_name1: type1,parameter\_name2:type2) 🡪 type of return

Data types in return and in parameters are float,int,str,blob,none

fetch(): is used with response to var2 as var2.fetchall() or fetchone() or fetchmany(no\_of\_rows\_in\_database)

Generator: which remember it’s state in between execution, it uses a keyword called yield which is like a return keyword ,it stops execution after the yield statement and continues it from there next time.

By using the:

yield var\_name

To print out: print(next(var\_name)) , print(list(var\_name))

To create our own generator:

class FirstHundredGenerator(object):

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

self.number = 0

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

if self.number < 100:

current = self.number

self.number += 1

return current

else:

raise StopIteration()

gen = FirstHundredGenerator()

next(gen) # 0

next(gen) # 1

Some more generator stuff:

filter(func,iterable): it simply move from one value to another does not perform any operation

friends = ['Rolf', 'Jose', 'Randy', 'Anna', 'Mary']

start\_with\_r = filter(lambda x: x.startswith('R'), friends)

print(start\_with\_r) # generator!

print(list(start\_with\_r))

map(func,iterable):it performs the function and bring back the values

friends = ['Rolf', 'Charlie', 'Anna']

friends\_lower = map(lambda x: x.lower(), friends)

print(list(friends\_lower))

startswith(‘alphabet’):it is used to check the elements first alphabet

any(var):it returns True or False based upon any value in the variable

all(var):it also returns the True or False based upon all the variables should be in the variable

enumerate(iterable):it simply assign them value from 0 to n based upon their position,it forms a tuple generator as (0,’value’) so to separate them we use the way we do it with keys and values in the dictionary

a tuple comprehension is a generator

“””

how to define iterator?----

How to define iterable in a class?

Iterator vs iterables: an iterable has an \_\_iter\_\_

How to convert iterator into iterables?

Generator comprehension

“””

Mutability:id()

A mutable object which can be changed when it is created

Integer,float,string,tuples are immtable

When we use = sign then the id gets changed but when we use += the id donot get changed

Default Parameters: the parameters can be made default by simply assigning them the values they need but make sure we use the default arguments after the arguments which don’t use default values

If we want to use the mutability and default arguments then the objects of function run by itself so in order to avoid either don’t use default argument or donot use a mutable object as a type hint in the parameter

Argument Unpacking: \*t it simply moves to the first value of the t variable and unpacks the following values as two arguments of a function

Dictionary Unpacking:\*\*t so that the first two values of the first two keys will be passed as two arguments

Why named arguments when dictionary unpacking? Because the dictionary is not in order

Collections: these are predefined things to save our time

\* counter: The counter is used to count an iterable and it will turn into a counter of elements.

from collections import Counter

device\_temperatures = [13.5, 14.0, 14.0, 14.5, 14.5, 14.5, 15.0, 16.0]

temperature\_counter = Counter(device\_temperatures)

print(temperature\_counter[14.0]) # 2

\* defaultdict

from collections import defaultdict

coworkers = [('Rolf', 'MIT'), ('Jen', 'Oxford'), ('Rolf', 'Cambridge'), ('Charlie', 'Manchester')] # Rolf got a master's

coworker\_alma\_maters = defaultdict(list) # remember list is a function, returns []

for coworker, place in coworkers:

coworker\_alma\_maters[coworker].append(place)

print(coworker\_alma\_maters['Rolf'])

print(coworker\_alma\_maters['Anne']) # []

\* ordereddict: The elements will be in the same order as they arrive

from collections import OrderedDict

o = OrderedDict()

o['Rolf'] = 6

o['Jose'] = 10

o['Jen'] = 3

print(o) # keys are always in the order in which they were inserted

o.move\_to\_end('Rolf')

o.move\_to\_end('Jose', last=False)

print(o)

o.popitem()

print(o)

\* namedtuple: A named tuple is another object that we can use like a tuple, where each of the elements of the tuple has a name. In addition, the tuple itself also has a name.

from collections import namedtuple

Account = namedtuple('Account', ['name', 'balance'])

account = Account('checking', 1850.90)

print(account.name)

print(account.balance)

The account keyword in the parameter of namedtuple is mandatory

\* deque: from collections import deque

friends = deque(('Rolf', 'Charlie', 'Jen', 'Anna'))

friends.append('Jose')

friends.appendleft('Anthony')

print(friends)

friends.pop()

print(friends)

friends.popleft()

print(friends)

Timezones

UTC is the reference time zone

the date and time object which do not know about the time zone is called Naïve otherwise it is called aware

Syntax:from datetime import datetime:

print(datetime.now())

Syntax: from datetime import datetime,timezone

print(datetime.now(timezone.utc))

To check the time to run a function

import time

start=time.time()

<block of function>

end=time.time()

print(end-start)

this time.time() simply checks the seconds from 1970 till that time

In a simple way

import timeit

print(timeit.timeit(list\_comprehension))🡪slower way

or

print(timeit.timeit(map(func,iteratble)))🡪faster way than above

REGEX:

regular expressions are a language and not a programming language

The `.` means “anything”; such as a letter, number, symbol, space, etc… \*but not newline characters\*.

The `+` means “one or more of”. The `\*` means “zero or more of”. The `?` means “zero or one of”.

The `^` means “beginning of string” and The `$` means “end of string”

So `.+` means “one or more of anything”. `.\*` means “zero or more of anything”. `.?` means “zero or one of anything”.

For `jose`, `[a-z]` would match every letter individually.

For `jose`, `[a-z]+` would match as many consecutive set of letters in that range as possible; that’s the entire word.

For `jo.se`, `[a-z]+` would match twice; `jo`, and `se`.

Let’s look at the e-mails.

[A-z]+@[a-z]+\.[a-z]+

Of course this one won’t match the periods or the underscores on the e-mail. Let’s fix it:

[A-z\.\_]+@[a-z]+\.[a-z]+

If instead of matching all TLDs (that’s `net`, `com`, me`) we wanted to match only the ones we’ve seen, we could do:

[A-z\.\_]+@[a-z]+\.(com|me|net)

There’s a lot more to regular expressions, but this will certainly get you started. http://regexr.com has more information, cheatsheets, and also the online editor for you to try things out.

How to apply in python?

import re

var1=’expression\_you\_want\_to\_match’

var2=’regex’

var3=re.findall(var2,var1)

var3 will be a list of all the items matched

replace(‘alphabet1’,’alphabet2’):it is used to replace an alphabet1 in a string by an alphabet2

if the var2 consists of () it means the output will be in groups i.e. gropu0 is simply everything and group1 is the regex in () these brackets

var3=re.search(var2,var1)

var3.group(0)🡪 string in that

HTML parsing:

To access the string inside an element

from bs4 import BeautifulSoup

var=’’’the html code’’’

var2=BeautifulSoup(var,’html.parser’)

print(var2.find(‘elements of html document’).string) or print(var2.find\_all(‘elements of html document’).string)

To find elements with defined class

find(‘element’,{‘class’:’class\_name’}) or

element.attrs[‘class’]

To access elements inside other element

locator=’<element.class\_name\_if\_any><space><element\_inside\_this>’

var3=var2.select\_one(locator)

print(var3.attrs[‘attribute\_inside\_the\_last\_element\_in\_locator’])

To fetch the html directly from a website

import requests

from bs4 import BeautifulSoup

var1=requests.get(‘http://www.example.com’)

print(var1.content)

it will printout the entire html code as a string

to further use it as a parser

var2=BeautifulSoup(var1.content,’html.parser’)

sorted():

Caching Problem

Can use something like functools.lru\_cache for caching function calls. That is, if you apply this decorator to a function and then you call the function with the same arguments 10 times, 9 of them will be really quick and the function won't evaluate.

Can use cachetools.TTLCache to cache a function call for up to a certain amount of time. When interacting with APIs it can be useful as sometimes we won't be interested in repeating the same call over and over.

Syntax: import functools

import time

@functools.lru\_cache(2)

def cached\_function(value):

for i in range(value):

i \*\* value

def timed():

start = time.time()

cached\_function(4647)

print(time.time() - start)

timed()

timed()

Synatx: import requests

from cachetools import cached, TTLCache

class OpenExchangeClient:

BASE\_URL = "https://openexchangerates.org/api/"

def \_\_init\_\_(self, app\_id):

self.app\_id = app\_id

@property

@cached(cache=TTLCache(maxsize=2, ttl=900))

def latest(self):

return requests.get(f"{self.BASE\_URL}/latest.json?app\_id={self.app\_id}").json()

def convert(self, from\_amount, from\_currency, to\_currency):

rates = self.latest['rates']

to\_rate = rates[to\_currency]

if from\_currency == 'USD':

return from\_amount \* to\_rate

else:

from\_in\_usd = from\_amount / rates[from\_currency]

return from\_in\_usd \* to\_rate

MULTITHREADING:

Every process when initiated it is initiated with a GIL that acts as a key and only one process can correlate with it until that is released. Multithreading helps us to perform two functions at one time and saves a lot of time.

Syntax: from threading import Thread

Var1=Thread(target=name\_of\_function1)

Var2=Thread(target=name\_of\_function2)

Var1.start()

Var2.start()

Var1.join()

Var2.join()

We can do multithreading with the help of context manager:

Syntax: from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max\_workers=no\_of\_processes) as var1:

var1.submit(name\_of\_function1)

varN.submit(name\_of\_functionN)

If threads are killed! Deadlock arises

MULTIPROCESSING: The syntax is quite different from multithreading

Syntax: from multiprocessing import Process

Var1=Process (target=name\_of\_function1)

Var1.start()

name\_of\_function2()

Var1.join()

We can do multiprocessing with the help of context manager by replacing ThreadPoolExecutor with ProcessPoolExecutor

Beside this async and await keyword are used which are beyond the scope of this!

Web Develpoment:

This is used to build html pages with the help of python with the help of Flask

We can create a base jinja2 file with the content like this:

|  |
| --- |
| <!DOCTYPEhtml> |
|  | <head></head> |
|  | <body> |
|  | {% block content %} |
|  | #the content keyword is just a variable |
|  | {% endblock %} |
|  | </body> |
|  | </html> |

And we can create other jinja2 files which contain the content of our html like this:

|  |
| --- |
| {%extends'base.jinja2'%} |
|  |  |
|  | {% block content %} |
|  | <h1>content </h1> |
|  | <p>content</p> |
|  | {% endblock %} |

But to use variables in these jinja files we use {{variable}} in this the way we do in print{} statements

Now in the main file we import Flask,render\_template from flask

var=Flask(\_\_name\_\_)

And use @var.route(your\_url,methods=[‘GET’,’POST’]) as a decorator on the functions

return render\_template(‘name\_of\_jinja2\_file.jinja2’)

Now to interact with the webpage through this:

if request.method == 'POST':

title = request.form.get('title')

content = request.form.get('content')

post\_id = len(blog['posts'])

blog['posts'][post\_id] = {'id': post\_id, 'title': title, 'content': content}

return redirect(url\_for('post', post\_id=post\_id))

return render\_template('create.jinja2')

and at the end

if \_\_name\_\_=’\_\_main\_\_’:

var.run()

GUI DEVELOPMENT

To check working

import tkinter

tkinter.\_test()

To print message:

**import** tkinter **as** tk  
  
**from** tkinter **import** ttk  
  
**def** greet():  
  
 print(**"Hello, World!"**)  
  
root = tk.Tk()  
  
root.title(**"Hello"**)  
  
greet\_button = ttk.Button(root, text=**"Greet"**, command=greet)  
  
greet\_button.pack(side=**"left"**, fill=**"x"**, expand=**True**)  
  
quit\_button = ttk.Button(root, text=**"Quit"**, command=root.destroy)  
  
quit\_button.pack(side=**"left"**, fill=**"x"**, expand=**True**) *# could use side="right"*root.mainloop()

To use text field

**import** tkinter **as** tk  
  
**from** tkinter **import** ttk  
  
**def** greet():  
  
 *# The get() method is used to fetch the value of a StringVar() instance.  
  
 # If user\_name is empty, print Hello, World!* print(**f"Hello, {**user\_name.get() **or 'World'}!"**)  
  
root = tk.Tk()  
  
root.title(**"Greeter"**)  
  
*# Here we create an instances of the StringVar() class, which is to track the content of widgets*user\_name = tk.StringVar()  
  
name\_label = ttk.Label(root, text=**"Name: "**)  
  
name\_label.pack(side=**"left"**, padx=(0, 10))  
  
name\_entry = ttk.Entry(root, width=15, textvariable=user\_name)  
  
name\_entry.pack(side=**"left"**)  
  
name\_entry.focus()  
  
greet\_button = ttk.Button(root, text=**"Greet"**, command=greet)

greet\_button.pack(side=**"left"**, fill=**"x"**, expand=**True**)  
  
root.mainloop()

To understand the packing

**import** tkinter **as** tk  
  
*# -- Aligning with `side` ---*root = tk.Tk()  
  
tk.Label(root, text=**"Label 1"**, bg=**"green"**).pack(side=**"left"**)  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**)  
  
root.mainloop()  
  
  
  
*# -- Filling in one direction --*root = tk.Tk()  
  
tk.Label(root, text=**"Label 1"**, bg=**"green"**).pack(side=**"left"**, fill=**"y"**)  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**, fill=**"x"**)  
  
root.mainloop()  
  
  
  
*# -- Filling in both directions --*root = tk.Tk()  
  
tk.Label(root, text=**"Label 1"**, bg=**"green"**).pack(side=**"left"**, fill=**"both"**)  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**, fill=**"both"**)  
  
root.mainloop()  
  
  
  
*# -- Even if either label doesn't fill --*root = tk.Tk()  
  
tk.Label(root, text=**"Label 1"**, bg=**"green"**).pack(side=**"left"**)  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**, fill=**"both"**)  
  
root.mainloop()

root = tk.Tk()  
  
tk.Label(root, text=**"Label 1"**, bg=**"green"**).pack(side=**"left"**, fill=**"both"**)  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**)  
  
root.mainloop()  
  
*# -- expand can make it grow as much as possible. It won't hide other widgets, but other widgets will be compressed --*root = tk.Tk()  
  
tk.Label(root, text=**"Label 1"**, bg=**"green"**).pack(side=**"left"**, expand=**True**, fill=**"both"**)  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**)  
  
root.mainloop()  
  
  
  
*# -- expanding two widgets means they share the available space evenly --*root = tk.Tk()  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
tk.Label(root, text=**"Label 2"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
root.mainloop()

*# -- whichever side comes first gets expansion priority --*root = tk.Tk()

tk.Label(root, text=**"Label left"**, bg=**"green"**).pack(  
  
 side=**"left"**, expand=**True**, fill=**"both"**)  
  
tk.Label(root, text=**"Label top"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
tk.Label(root, text=**"Label top"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
root.mainloop()

root = tk.Tk()  
  
tk.Label(root, text=**"Label top"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
tk.Label(root, text=**"Label top"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
tk.Label(root, text=**"Label left"**, bg=**"green"**).pack(  
  
 side=**"left"**, expand=**True**, fill=**"both"**)  
root.mainloop()  
  
*# As you can see, even though we specificied `side="left"`, the last label was still underneath.  
  
# We can't just use packing on the root to take care of all our layout needs.  
  
# Hence, Tkinter has `Frame`, which is a container for other widgets.*

Packing with frames

**import** tkinter **as** tk  
**from** tkinter **import** ttk

root = tk.Tk()  
  
main = ttk.Frame(root)  
  
main.pack(side=**"left"**, fill=**"both"**, expand=**True**)  
  
tk.Label(main, text=**"Label top"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
tk.Label(main, text=**"Label top"**, bg=**"red"**).pack(side=**"top"**, expand=**True**, fill=**"both"**)  
  
tk.Label(root, text=**"Label left"**, bg=**"green"**).pack(  
  
 side=**"left"**, expand=**True**, fill=**"both"**)  
  
  
root.mainloop()

Adding a notebook

**import** tkinter **as** tk  
**from** tkinter **import** ttk

**def** create\_file():

text\_area = tk.Text(notebook)  
  
 text\_area.pack(fill=**"both"**, expand=**True**)  
  
 notebook.add(text\_area, text=**"Untitled"**)

notebook.select(text\_area)  
  
  
root = tk.Tk()  
  
root.title(**"Teclado Text Editor"**)  
  
main = ttk.Frame(root)  
  
main.pack(fill=**"both"**, expand=**True**, padx=(1), pady=(4, 0))  
  
notebook = ttk.Notebook(main)  
  
notebook.pack(fill=**"both"**, expand=**True**)  
  
create\_file()  
  
root.mainloop()

Adding a menu

**import** tkinter **as** tk  
**from** tkinter **import** ttk

**def** create\_file():  
  
 text\_area = tk.Text(notebook)  
  
 text\_area.pack(fill=**"both"**, expand=**True**)  
  
 notebook.add(text\_area, text=**"Untitled"**)  
  
 notebook.select(text\_area)  
  
root = tk.Tk()  
  
root.title(**"Teclado Text Editor"**)  
  
root.option\_add(**"\*tearOff"**, **False**)  
  
main = ttk.Frame(root)  
  
main.pack(fill=**"both"**, expand=**True**, padx=(1), pady=(4, 0))  
  
menubar = tk.Menu(root)  
  
root.config(menu=menubar)  
  
file\_menu = tk.Menu(menubar)  
  
menubar.add\_cascade(menu=file\_menu, label=**"File"**)  
  
file\_menu.add\_command(label=**"New"**, command=create\_file)  
  
notebook = ttk.Notebook(main)  
  
notebook.pack(fill=**"both"**, expand=**True**)  
  
create\_file()  
root.mainloop()

Saving the file

import tkinter as tk

from tkinter import ttk, filedialog

def create\_file():

text\_area = tk.Text(notebook)

text\_area.pack(fill="both", expand=True)

notebook.add(text\_area, text="Untitled")

notebook.select(text\_area)

def save\_file():

file\_path = filedialog.asksaveasfilename()

try:

filename = file\_path.split("/")[-1]

text\_widget = root.nametowidget(notebook.select())

content = text\_widget.get("1.0", "end-1c")

with open(file\_path, "w") as file:

file.write(content)

except (AttributeError, FileNotFoundError):

print("Save operation cancelled")

return

notebook.tab("current", text=filename)

root = tk.Tk()

root.title("Teclado Text Editor")

root.option\_add("\*tearOff", False)

main = ttk.Frame(root)

main.pack(fill="both", expand=True, padx=(1), pady=(4, 0))

menubar = tk.Menu(root)

root.config(menu=menubar)

file\_menu = tk.Menu(menubar)

menubar.add\_cascade(menu=file\_menu, label="File")

file\_menu.add\_command(label="New", command=create\_file)

file\_menu.add\_command(label="Save", command=save\_file)

notebook = ttk.Notebook(main)

notebook.pack(fill="both", expand=True)

create\_file()

root.mainloop()

Opening a file

def open\_file():

file\_path = filedialog.askopenfilename()

try:

filename = file\_path.split("/")[-1]

with open(file\_path, "r") as file:

content = file.read()

except (AttributeError, FileNotFoundError):

print("Open operation cancelled")

return

create\_file(content, filename)

THE rest project with shortcut binding,closing individual tabs while checking for unsaved state and adding a permanent scroll bar along with an help menu

import tkinter as tk

from tkinter import ttk, filedialog, messagebox

text\_contents = dict()

def check\_for\_changes():

current = get\_text\_widget()

content = current.get("1.0", "end-1c")

name = notebook.tab("current")["text"]

if hash(content) != text\_contents[str(current)]:

if name[-1] != "\*":

notebook.tab("current", text=name + "\*")

elif name[-1] == "\*":

notebook.tab("current", text=name[:-1])

def get\_text\_widget():

current\_tab = notebook.nametowidget(notebook.select())

text\_widget = current\_tab.winfo\_children()[0]

return text\_widget

def close\_current\_tab():

current = get\_text\_widget()

if current\_tab\_unsaved() and not confirm\_close():

return

if len(notebook.tabs()) == 1:

create\_file()

notebook.forget(current)

def current\_tab\_unsaved():

text\_widget = get\_text\_widget()

content = text\_widget.get("1.0", "end-1c")

return hash(content) != text\_contents[str(text\_widget)]

def confirm\_close():

return messagebox.askyesno(

message="You have unsaved changes. Are you sure you want to close?",

icon="question",

title="Unsaved changes",

)

def confirm\_quit():

unsaved = False

for tab in notebook.tabs():

tab\_widget = root.nametowidget(tab)

text\_widget = tab\_widget.winfo\_children()[0]

content = text\_widget.get("1.0", "end-1c")

if hash(content) != text\_contents[str(text\_widget)]:

unsaved = True

break

if unsaved and not confirm\_close():

return

root.destroy()

def create\_file(content="", title="Untitled"):

container = ttk.Frame(notebook)

container.pack()

text\_area = tk.Text(container)

text\_area.insert("end", content)

text\_area.pack(side="left", fill="both", expand=True)

notebook.add(container, text=title)

notebook.select(container)

text\_contents[str(text\_area)] = hash(content)

text\_scroll = ttk.Scrollbar(container, orient="vertical", command=text\_area.yview)

text\_scroll.pack(side="right", fill="y")

text\_area["yscrollcommand"] = text\_scroll.set

def open\_file():

file\_path = filedialog.askopenfilename()

try:

filename = file\_path.split("/")[-1]

with open(file\_path, "r") as file:

content = file.read()

except (AttributeError, FileNotFoundError):

print("Open operation cancelled")

return

create\_file(content, filename)

def save\_file():

file\_path = filedialog.asksaveasfilename()

try:

filename = file\_path.split("/")[-1]

text\_widget = get\_text\_widget()

content = text\_widget.get("1.0", "end-1c")

with open(file\_path, "w") as file:

file.write(content)

except (AttributeError, FileNotFoundError):

print("Save operation cancelled")

return

notebook.tab("current", text=filename)

text\_contents[str(text\_widget)] = hash(content)

def show\_about\_info():

messagebox.showinfo(

title="About",

message="The Teclado Text Editor is a simple tabbed text editor designed to help you learn Tkinter!",

)

root = tk.Tk()

root.title("Teclado Text Editor")

root.option\_add("\*tearOff", False)

main = ttk.Frame(root)

main.pack(fill="both", expand=True, padx=(1), pady=(4, 0))

menubar = tk.Menu(root)

root.config(menu=menubar)

file\_menu = tk.Menu(menubar)

help\_menu = tk.Menu(menubar)

menubar.add\_cascade(menu=file\_menu, label="File")

menubar.add\_cascade(menu=help\_menu, label="Help")

file\_menu.add\_command(label="New", command=create\_file, accelerator="Ctrl+N")

file\_menu.add\_command(label="Open...", command=open\_file, accelerator="Ctrl+O")

file\_menu.add\_command(label="Save", command=save\_file, accelerator="Ctrl+S")

file\_menu.add\_command(

label="Close Tab", command=close\_current\_tab, accelerator="Ctrl+Q"

)

file\_menu.add\_command(label="Exit", command=confirm\_quit)

help\_menu.add\_command(label="About", command=show\_about\_info)

notebook = ttk.Notebook(main)

notebook.pack(fill="both", expand=True)

create\_file()

root.bind("<KeyPress>", lambda event: check\_for\_changes())

root.bind("<Control-n>", lambda event: create\_file())

root.bind("<Control-o>", lambda event: open\_file())

root.bind("<Control-q>", lambda event: close\_current\_tab())

root.bind("<Control-s>", lambda event: save\_file())

root.mainloop()

Decorator are the functions which return a function and accept a function as a parameter

def user\_has\_permission(func):

def secure\_func():

if user.get('access\_level') == 'admin':

return func()

return secure\_func

way 1:

def my\_function():

return 'Password for admin panel is 1234.'

my\_secure\_function = user\_has\_permission(my\_function)

print(my\_secure\_function())

way2:

@user\_has\_permission

def my\_function():

return 'Password for admin panel is 1234.'

print(my\_function.\_\_name\_\_)

Now since function do not return its name we use functools for this by import functools

And use @functools.wraps(func) on def secure\_func

Generic Decorator is the one which accept any parameter we can use argument unpacking for this purpose

def user\_has\_permission(func):

@functools.wraps(func)

def secure\_func(\*args, \*\*kwargs):

if user.get('access\_level') == 'admin':

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

return secure\_func

The \* is for list,tuple and \*\* is for dictionary when my\_function has a parameter In it

Smtplib

import smtplib

from email.message import EmailMessage

var1=’’’content’’’

var2=EmailMessage()

var2[‘Subject’]=’Subject’

var2[‘From’]=’From’

var2[‘To’]=’To’

var2.setcontent(var1)

smtplib.SMTP(host=’smtp.gmail.com’,port=587).starttls()

smtplib.SMTP(host=’smtp.gmail.com’,port=587).login(‘email’,’password’)

smtplib.SMTP(host=’smtp.gmail.com’,port=587).send\_message(var2)

smtplib.SMTP(host=’smtp.gmail.com’,port=587).quit()

Unit Testing

from unittest import TestCase

class class\_name(TestCase):

def func\_name(self):

way1:self.assertAlmostEqual(func(par1,par2), result\_var, delta=0.0001)

way2:self.assertEqual(func(par1,par2),result\_var)

Test for errors:

with self.assertRaises(TypeOfError):

func(value1,value2) #the result should cause an error

|  |
| --- |
|  |
|  |
|  |
|  |