The 1Page Python Book

Beginners guide to Programming

Barani Kumar

The 1Page Python Book

Author - Barani Kumar https://linkedin.com/in/thirubaranikumar[LinkedIn]

Table of Contents

Presenting your application - GUI

Create a working UI and functions (Marketplace App)

Calculator application (UI & functional implementation)

Software Development Life Cycle

Performing User Research & Presentation

List of IDEs

Debugging & Performance Improvement

Learning Tools (Pycharm edu tools)

Copyright

Published, 2021

ISBN-13 9781005944803

Barani Kumar, Chennai, India

https://1pagepythonbook.club

About the Author

Barani Kumar, is an engineer, data analyst he has worked in engineering projects contributing in development and testing of Information and Technology solutions. He has experiences in wide businesses, Avionics, Semi-conductor, Software management and Financial Industry. Currently he is serving as the Product Developer and Co-Director for a Financial Services Company.

For suggestions or feedback you can write to my email – [hello@baranikumar.space]

Do subscribe to my newsletter @ [https://1pagepythonbook.club], Also find my 1 Page reading plan for this book in the website.

If you fail to plan, you are planning to fail, Benjamin Franklin.

Your positive reviews on the Smashwords book page is greatly appreciated.

Smashwords:

https://www.smashwords.com/books/view/1067036

Introduction

This Book provides an introduction into python programming, teaches python language basics and handles some of the intermediate concepts as well. you will be able to follow and do it yourself in the terminal provided at end of this file.

You can also set up a development environment in your computer and practice the concepts explained.

Hear from the founder {https://gvanrossum.github.io/}[Guido Van Rossum] of python language in his King's Day Speech {http://neopythonic.blogspot.com/2016/04/kings-day-speech.html}[blogpost] for some inspiration.

What is Programming ?

A program is a set of rules which performs calculations on a computer. The main purpose is to automate human efforts.

Read the section on history of automation in daily life {https://www.britannica.com/technology/automation/Computer-integrated-manufacturing#ref24857}[Automation in Telecom,Transportation, Consumer Products etc.]

Humans have always wanted to increase productivity. Some for their own good, some for the well being of mankind. They came up with mechanical, hydraulic and pneumatic systems. These systems offloaded some problems. But stuck with decision making abilities. With advent of electronics and semiconductors Computers began to emerge and thus certain ways for mathematical computation. All decision making tasks were seen as mathematical problems too and solved with help of computers. An abstract decision problem can be fit into something like this.

Input → Process → Output

An Input can be a data or a signal to perform or trigger some process.

A process will be the actions that will be performed either on the data or the with the signal that is received.

An output is the end goal or result that needs to be produced.

History of Programming

Initially programming was done only in Hex codes on a microprocessor. These were called low level programming language to talk to processing computer directly. These were too difficult to write complex programs or logics and hence high level languages were invented to interact with processing computer more naturally using english language words. Languages like ALGOL, Lisp & C came into existence (https://en.wikipedia.org/wiki/Programming_language).

Currently there are more than 700+ programming languages in existence. Many others have demised too due to its non-usage. Refer to the wiki link. https://en.wikipedia.org/wiki/List_of_programming_languages

How different is Python

Initially developed High level languages were called compiled language. The written programming code has to be processed before executing it on the computer processor. The written programming code can also be called as source code. This processing software is referred to as compiler. The compiler reads the complete source code, checks if there are no errors and then convert it to machine readable code (byte code). Only this byte code is executed on processors, they can only understand 0's and 1's. Before converting it to byte code the compiler first checks if the source code is devoid of errors. It checks for syntax error and semantic error. Syntax error are those that caused by inappropriate usage of words and grammar defined in the context of programming. Semantic errors are those that are logically incorrect and compiler will not be able to understand. Right now do not worry how would you write this program, syntax, what is the grammar or logical incorrectness for compiler to check. Many programming languages each with their own set of syntax, grammar were defined by individuals and organizations. It took hours for a compiler to convert to machine code before executing on computer in early days around 1980's. Few felt it is not wise to wait such long time to test small piece of code whether to update or check a system. Hence interpreted language was invented. In interpreted language

every single line or block of lines are checked for syntax and semantic errors, if there are no errors then only that segment is executed. After successful execution the next statement is picked for execution. This is repeated for the complete code that is provided to interpreter. Whenever a error is encountered, the complete execution gets halted and interpreter does no further processing until submitted again.

Python is one of the most loved programming languages by developers across the world. Mainly because of the philosophy behind the language development free and open source software.

Features of the language

Open source – Python is made available as a open source programming language – actively maintained by Python Software Foundation.

Multipurpose – Python is used for a wide variety of purpose and in many business domains.

Cross-platform – The program is cross – platform, it can run across multiple operating systems.

Dynamically typed – A data type is inferred automatically during run time, there is no static type checking.

Indentation – Code blocks in python are defined using indentation in contrast to braces in other programming languages.

Garbage collection – Python automatically performs garbage collection, memory managed by itself and there is no need to explicitly allocate and free memory in code.

Installation

Installing python is a simple process. It is little different compared to windows and other unix based system.

Windows OS:

Go to python.org website and navigate to this path https://www.python.org/downloads/windows/

Search for version 3.8.2 and download the windows executable installer file.

Click the downloaded EXE file and follow through the instructions.

Ensure python is installed correctly by doing this test.

Open a new command prompt by typing cmd + R and then type ‘cmd’.

Type python and hit enter. If the terminal output shows python version that you just installed and switched to python terminal then installation is successful.

Mac OS:

Go to python.org website and navigate to this path https://www.python.org/downloads/mac-osx/

Search for version 3.8.2 (or any version in python 3.8.x) and download the macOS 64 bit installer file.

Click the downloaded PKG file and follow through the instructions.

Ensure python is installed correctly by doing the same test.

Open a new command terminal from launcher, type python3.8 and hit enter. If the terminal output shows python version that you just installed and switched to python terminal then installation is successful.

All interpreted languages follow what is called as REPL method,

Read – Evaluate – Print – Loop

This offers a interactive environment. You can type 2 + 2 at the terminal and see the result immediately. Or for the fact any valid python statement can be typed and obtain its result.

Language Basics - Input,Process & Output

Variables & Expressions

We are going to see the first building block in writing our program. Lets have some role play to help you visualize and understand the concept, Consider your grandfather has given some amount of money to buy seeds in the coming month. You need some placeholder to keep this money value, this we will call it as a variable or a more common terminology - identifier, and we need to give a name to it.

For example here the identifier that we will create is money.

And the value we store is 50000 as in below line.

All python code are represented inside a block like below. Actual python lines start from column 0 in screen. To represent Outputs below the code are

in line starting with '>>'

```
# Storing a numeric value in money 2)

money = 50000 # 1)
```

1) money is the Variable. It is also referred as identifier in technical terms. 2) In python, character # represents a ***comment*** character. Any text following # will not be processed by the interpreter. It is used to provide description and improve readability of the program.

To send this value to the terminal, we can use the print statement like in below line.

```
print(money)
```

Now, there are some rules for creating this identifier name,

- Should not start with numbers Eg: 1 2 3…etc.
- Should not start with special symbols Eg: - ! @ # $ % ^ & * ( ) …etc except underscore _ symbol
- Can start with alphabets in any case.
- Can have mix of alphabets and numbers and _
- Should not contain space in between the name
- Theoretically identifier name can be of any length. But always prefer giving meaningful names.

Data Types to store values

Basic Data Types

1) Numeric

2) String

3) Bool

A **Numeric** can be further categorized into 3 types.

Complex: A complex number is combination of 2 different numbers, it includes a real value and an imaginary value. 2+3i is an example of a complex number. It finds extensive usage in algebraic equations containing binomial expressions.

Complex numbers are applied in electronics and engineering fields. Read more about complex numbers and its application in [https://www.mathsisfun.com/numbers/complex-numbers.html] [Mandelbrot pattern] generation.

Integer: An Integer stores a whole number (It includes range from 0 to +∞ and 0 to -∞). It does not include decimal values.

When we want to store a whole number, we declare a statement as below.

```
money = 50000
```

This automatically takes data type as integer based on the supplied value. This makes programming easier. Suppose we want to store a number with decimal places, we write a statement as below,

Float: A Float includes decimal values and has range from 0 to +∞ and 0 to -∞.

```
seedprice = 12.5
```

This automatically takes data type as float based on the supplied value. While creating numeric values we can also declare in other number systems. But ultimately numbers will be store in decimal system only.

For example for numeric value 2 , in **binary** it is 0010. To create statement to store in a variable x,

```
x = 0b0010 #(There must be 0 followed by b)
```

For example for numeric value 12 , in **octal** it is 14. To create statement to store in a variable x,

```
$x = 0o14 #(There must be 0 followed by o)
```

For the same numeric value 12 , in **hex** it is C. To create statement to store in a variable x,

```
x = 0xC #(There must be 0 followed by x)
```

We just saw the following, How to,

- Create integers
- Create float
- Store a binary value
- Store a octal value

- Store a hexadecimal value

You may refer this [https://www.rapidtables.com/convert/number/base-converter.html] [online calculator] to verify number system conversions. Learning the conversions by hand is advised from an aptitude point of view.

String: In this lesson we will see how to ***CREATE, ACCESS, EDIT, MODIFY & DELETE*** string values.

CREATE: A string value can be stored in either inside single quotes, double quotes or triple quotes. Lets see how to create them and purpose of each of them. Lets store a string value “tomato” in variable seedname1

```
Seedname1 = "tomato" #(this is within double quotes and perfectly valid)
```

Now store a string value ‘brinjal’ in variable seedname2

```
seedname2 = 'brinjal' #(this is within single quotes and perfectly valid)
```

Every opening quote has to be terminated by another closing quote. Otherwise it will cause errors. We should not mix between the quotes as well.

Some of the valid declarations:

```
Target_statement = "To get yield of 100 'kgs' "
```

Some of the invalid declarations:

```
Target_statement = 'To get yield of 100 'kgs' '

Target_statement = "To get yield of 100 'kgs' '
```

And finally we can store strings in triple quotes as well like following,

```
Target_statement = """ To start sowing tomato

seeds and achieve

20 kgs yield"""
```

(Declaring a string in triple quotes can span across multiple lines) Except that it is like the same string as in before case. The places where we will use this, we will discuss at appropriate times. This will be used in concept of functions and classes most often.

Life cycle of Data type & Identifier:

Now this is a right time to introduce life cycle of a identifier and the value we store. With any identifier,

- We can create and store value,
- Edit value
- Modify/ Transform stored value
- Delete the value or identifier itself.

EDIT: Lets see an illustration for this using the string data type:

Let me create a variable Seedname1 with value tomato.

```
Seedname1 = 'tomato'
```

Now when I want to change its value, I can reassign it with a new value like below

```
Seedname1 = 'brinjal'
```

To change its case – from lower case to upper case, I can use what is called inbuilt functions in python.

```
Seedname1.upper() # (this will output a new string and we have to overwrite it to the same variable to change its value)

Seedname1 = Seedname1.upper()
```

ACCESS:

You can print this output also, using the print statement to make sure the value has got updated.

```
print(Seedname1)
```

To delete the value only, we can simply set it to empty quotes.

```
Seedname1 = ""
```

To delete the identifier itself, we can use

```
del Seedname1
```

Now when we try to print Seedname1, we will get a error since the identifier is deleted already.

```
print(Seedname1)
```

Access – Slicing - Substring:

Strings in python are ***immutable***. We cannot edit the string present already in a memory location. We can only create a new string and assign it to the

existing identifier. Substrings are portions of existing string. We can access a portion of an existing string by using the slicing operation.

Eg: To access world from below string.

```
s = "hello world"
s[6:11]
>>>world
```

MODIFY:

Lets see the different transformation functions that are available inbuilt in the string data type:

capitalize: Converts to capital the first character of the string.

```
s = 'brinjal'
>>’Brinjal’
```

casefold: This converts to lowercase also includes all unicode character equivalents. Follows casefold algorithm.

```
s = 'BRINJAL'
>>’brinjal’
```

center: Returns a new string centered by adding white spaces before and after according the length input provided. (white spaces are the padding)

```
s.center(20)
>>' brinjal '
```

count: Counts number of occurrences of a specific character.

```
s.count('j')
>>1
```

encode: Encode method returns an binary encoded version based on the encoding method we mention. Default is utf-8 for all text input.

```
s.encode()
>>b'brinjal'
sym = ‘☆’
sym.encode()
>>b'\xe2\x9c\xaf'
```

To see all [https://unicode-table.com/en/] [unicode character]. To read more on [https://en.wikipedia.org/wiki/UTF-8] [utf-8 encoding].

endswith: Useful to check If a string ends with a specific character.

```
s='brinjal'

s.endswith('l')

>>True

s.endswith('o')

>>False
```

expandtabs: Replace tab character in a string with spaces. Default input is 8 and multiples of 8. The statement while executing starts counting at 1 and increments for every character. whenever tab \t is encountered fills with spaces until the counter equals multiple of 8. And starts counter from 1 again for the next character. All the characters except tab are copied same. If any other numeric input is given then the expands tab counter said above uses that as base.

```
'01\t012\t012301234'.expandtabs()

>>'01 012 0123 01234'

'01\t012\t012301234'.expandtabs(4)

>>'01 012 0123 01234'
```

find: Is it used to get the starting index position of a substring.

```
s = ‘brinjal’

s.find(‘jal’)

>>4
```

Remember index position of string in python starts at 0

format: The format operation replaces placeholders in curly braces with the supplied tuple values. Within curly braces are mentioned the index of values provided to format. Also to note the format operation is carried on a valid string.

```
"The sum of {0} + {1} is {2}".format(1,2,1+)

'The sum of 1 + 2 is 3'
```

format will expect same number of values equal to the number of placeholders created in string.)

If a particular placeholder index is not found Index error will be raised.

format_map: The format_map operation replaces placeholders in curly braces with the supplied class dictionary values. Within curly braces are mentioned the key to be replaced with its value. There can be missing key by using this way of format_map.

```
class Default(dict):

def __missing__(self, key):

return key

'{name} was born in {country}'.format_map(Default(name='Guido'))

>>'Guido was born in country'
```

index: Similar to find method. Returns index position for a given substring. But throws error when substring is not found.

```
s

>>'brinjal'

s.index('jal')

>>4

s.index('dal')

>>Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: substring not found
```

isalnum: Returns true if the string is alphabet or numeric or alphanumeric else returns False

```
s.isalnum()

>>True
```

isalpha: Returns true if the string is alphabet else returns False

```
s = '100'

s.isalnum()

>>True
```

```
s.isalpha()
>>False
```

isascii: Returns true if the string contains ascii characters else returns False

```
s = '100'
s.isascii()
>>True
```

isdecimal: Returns true if the string contains numeric or decimal characters

```
s = '100'
s.isdecimal()
>>True
```

isdigit: Returns true if the string contains digits

```
s = '100'
s.isdigit()
>>True
```

isidentifier: Returns True if the string matches any identifier in memory

```
s.isidentifier()
>>False
```

islower: Returns true if all characters are lowercase

```
s = 'tomato'
s.islower()
>>True
```

isnumeric: Returns true if all characters are numeric

```
s = '100'
s.isnumeric()
>>True
```

isprintable: Certain unicode characters cannot be printed, if any contains them then this will return false.

```
s.isprintable()
>>True
```

isspace: Returns True if there is one or more white space only.

```
s = ' '
s.isspace()
>>True
```

istitle: Returns true if the first character is in upper case and others in lower.

```
s = 'Banana'
s.istitle()
>>True
s = 'banana'
s.istitle()
>>False
```

isupper: Returns true is all the letters in upper case

```
s = 'BANANA'
s.isupper()
>>True
```

join: Returns a string by joining all elements of a collection iterable and using the provided separator.

```
l = ['apple','banana','grape']
' '.join(l)
>>'apple banana grape'
```

ljust: Returns a string in left justified manner with the provided string length.

```
s = 'apple'
s.ljust(20)
>>'apple '
```

lower: Converts all characters to lowercase form in the string

```
s = "Apples are Healthy"
s.lower()
>>'apples are healthy'
```

lstrip: Removes white spaces from left of the string.

```
s = " Apples are Healthy "
s.lstrip()
>>'Apples are Healthy '
```

maketrans: Build a translation table to be used in translate method.

```
txt = "Hello Programmer!"
mytable = txt.maketrans("P", "D")
print(txt.translate(mytable))
>>Hello Drogrammer!
```

partition: Splits a string using the separator and returns 3 portions as a tuple. 1 – string before separator, 2 – separator, 3 – string after separator. This splits at first occurrence.

```
name = 'Ram,Gopal'
name.partition(',')
>>('Ram', ',', 'Gopal')
```

replace: Replace a portion of a string with the input characters.

```
s = "Apples are Healthy"
s.replace('Apples','Oranges')
>>'Oranges are Healthy'
```

rfind: Returns the highest index of the occurring substring. In the below statement using rfind returns 5, whereas find would have returned the first value as 1.

```
c = 'potato'
c.rfind('o')
>>5
```

rindex: Similar to rfind() but raises ValueError when the substring is not found.

```
c = 'potato'
c.rindex('o')
>>5
```

```
c.rindex('x')

>>Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: substring not found
```

rjust: Returns a string in right justified manner within the provided string length.

```
s = 'apple'

s.rjust(20)

>>' apple'
```

rpartition: Splits a string using the separator and returns 3 portions as a tuple. 1 – string before separator, 2 – separator, 3 – string after separator. This splits at last occurrence.

```
name = 'Ram,Gopal'

name.rpartition(',')

>>('Ram', ',', 'Gopal')
```

rsplit: Returns a list of elements by splitting using the separator character. This spilts starting from right internally.

```
'1,2,3'.rsplit(',')

>>['1', '2', '3']
```

rstrip: Removes white spaces from right of the string. s = " Apples are Healthy " s.rstrip() >>' Apples are Healthy'

split: Returns a list of elements by splitting using the separator character. This spilts starting from left internally. Produces similar result to rsplit.

```
'1,2,3'.rsplit(',')

>>['1', '2', '3']
```

splitlines: Splits a string at newline boundaries and returns a list of elements. New line boundary elements can be any of below (Refer to python help for more info)

\n	Line Feed
\r	Carriage Return
\r\n	Carriage Return + Line Feed
\v or \x0b	Line Tabulation
\f or \x0c	Form Feed
\x1c	File Separator
\x1d	Group Separator
\x1e	Record Separator
\x85	Next Line (C1 Control Code)
\u2028	Line Separator
\u2029	Paragraph Separator

```
'ab c\n\nde fg\rkl\r\n'.splitlines()
>>['ab c', '', 'de fg', 'kl']
```

startswith: Returns true if a string startswith the provided character.

```
s = 'Bananas'
s.startswith('B')
>>True
```

strip: Returns a new string removing trailing white space both in the beginning and end of the string.

```
s = " Apples are Healthy "
s.strip()
>>'Apples are Healthy'
```

swapcase: Returns a new string interchanging character casing, upper case in place of lower and lower case in place of upper.

```
s = " Apples are Healthy "

s.swapcase()

>>' aPPLES ARE hEALTHY '
```

title: Returns a new string where words start with an uppercase character and the remaining characters are lowercase.

```
'Hello world'.title()

>>'Hello World'
```

translate: Translates a string using a translation table prebuilt.

```
txt = "Hello Programmer!"

mytable = txt.maketrans("P", "D")

print(txt.translate(mytable))

>>Hello Drogrammer!
```

upper: Returns a new string converting all lower case to upper case.

```
s = 'bananas'

s.upper()

>>'BANANAS'
```

zfill: Returns a new string adding padded zeros before the input string. Number of zeros are decided by the zfill count argument which the total length of the new output string.

```
"42".zfill(5)

>>'00042'

"-42".zfill(5)

>>'-0042'
```

DELETE:

del : This command deletes the identifier from memory. Attempt to access after deletion throws NameError.

```
del s
```

```
print(s)
>>Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 's' is not defined
```

It is not required to remember all of this, you can quickly refer to the python help and only understand what is possible when using strings in python. You can also read more about this from the python user manual page. https://docs.python.org/3.9/library/stdtypes.html#text-sequence-type-str

Boolean: This is a data type to store binary data 1 and 0, It is created using keywords True and False respectively. The numeric equivalent for True is 1 The numeric equivalent for False is 0

```
flag = True #An example boolean identifier
token = False
```

Derived Data Types

1) List

2) Tuple

3) Set

4) Dictionary

LIST:

CREATE:

A list is created using square brackets []

```
flowers= []
```

We can modify elements in a list in any order and allows to delete also. We can store any data type.

For simplicity we will handle string objects alone for now.

append: Add a single element to end of list.

```
flowers = [“rose”,”jasmine”,”lavender”,”lily”]
flowers.append(“daffodil”)
flowers.append(“sunflower”)
```

access: Access elements of a list using its index and slicing operation. Fetches from lower index to upper index – 1. Remember list index starts from 0.

```
Print(flowers)

flowers[0:2]
```

EDIT:

To change any value is a list simple assign the new values to the required index position within square brackets.

```
flowers[0] = "roses"

flowers
```

MODIFY:

Following are other functions available in list.

extend: Add multiple elements to end of list.

```
flowers.extend(["roses","roses","jasmine"])

print(flowers)

>>['roses', 'jasmine', 'lavender', 'lily', 'daffodil', 'roses', 'roses', 'jasmine']
```

count: Count the number of occurrences of a specific element.

```
flowers.count('roses')

>>3

flowers.count('jasmine')

>>2
```

insert: Insert a new element in a specific index position. Insert ‘tulip’ in index 2.

```
flowers.insert(2,'tulip')

flowers

>>['roses', 'jasmine', 'tulip', 'lavender', 'lily', 'daffodil', 'roses', 'roses', 'jasmine']
```

index: Find the index position of a list element. It returns only the first found position.

```
flowers.index('tulip')
```

```
>>2
```

copy: Create a new copy of the list. Any changes made in the new copy will not affect the original list and likewise changes in original list will not affect the new copy created.

```
bokeh = flowers.copy()

bokeh

>>['roses', 'jasmine', 'tulip', 'lavender', 'lily', 'daffodil', 'roses', 'roses', 'jasmine']

flowers.append('tulip')

# This new addition will not be available in bokeh list.
```

reverse: Reverse the elements in list and stores the reversed elements also.

```
flowers

>>['roses', 'jasmine', 'tulip', 'lavender', 'lily', 'daffodil', 'roses', 'roses', 'jasmine', 'tulip']

flowers.reverse()

flowers

>>['tulip', 'jasmine', 'roses', 'roses', 'daffodil', 'lily', 'lavender', 'tulip', 'jasmine', 'roses']
```

sort: Sort the elements of list in ascending order.

```
flowers.sort()

flowers

>>['daffodil', 'jasmine', 'jasmine', 'lavender', 'lily', 'roses', 'roses', 'roses', 'tulip', 'tulip']
```

DELETE operations:

remove: Remove a element by its value. It removes the first occurring element.

```
flowers.remove('tulip')

flowers

>>['daffodil', 'jasmine', 'jasmine', 'lavender', 'lily', 'roses', 'roses', 'roses', 'tulip']
```

pop: Remove a element from a specific index position. To remove lavender from index 3.

```
flowers.pop(3)

>>'lavender'
```

```
flowers
>>['daffodil', 'jasmine', 'jasmine', 'lily', 'roses', 'roses', 'roses', 'tulip']
```

clear: Empties the complete list.

```
flowers.clear()
flowers
>>[]
```

Finally to delete the identifier:

```
del flowers
flowers
>>Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'flowers' is not defined
```

TUPLE – DATA TYPE:

Tuple is a *immutable* data type. Once a tuple is created neither its elements can be modified nor new elements be added. To create a tuple use () brackets.

create: create a tuple within curved brackets

```
crops = ("rice","corn","pulses")
crops
>>('rice', 'corn', 'pulses')
```

access: access a element in tuple using square brackets and index

```
crops[0]
>>'rice'
crops[1]
>>'corn'
```

modify: Tuple element cannot be modified. Attempt to modify will throw error.

```
crops[1] = 'sugarcane'
>>Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

count: count the number of occurrences of a specific element in tuple.

```
crops.count('rice')

>>1
```

index : returns the index position of a specific element

```
crops.index('pulses')

>>2
```

DELETE:

Tuple element cannot be deleted as well. There is no function available to remove or delete. But the complete tuple can be deleted.

```
del crops

crops

>>Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'crops' is not defined
```

SET - DATA TYPE

Set is mostly used for mathematical functions.

CREATE:

A set is created using set() command. It can also be created by assigning sequence of values with in curly braces{}.

```
Vegetables1 = {'carrot','radish','cabbage','spinach"beans'}

Vegetables2 = {'carrot','radish','carrot','eggs"milk'}

Vegetables1

>>{'cabbage', 'spinach', 'beans', 'radish', 'carrot'}

Vegetables2

>>{'eggs', 'radish', 'carrot', 'milk'}
```

A set does not store duplicates. It does not have index also.

ACCESS:

Cannot be accessed with index position.

```
Vegetables1[1]

>>Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'set' object is not subscript-able
```

MODIFY:

add: To add element to a existing set

```
Vegetables1.add('potato')

Vegetables1

>>{'beans', 'cabbage', 'carrot', 'spinach', 'radish', 'potato'}
```

difference : To find the difference between 2 sets. Here difference between Vegetables1 and Vegetables2. Difference only gives the result as a new set and does not cause changes to existing sets.

```
Vegetables1.difference(Vegetables2)

>>{'cabbage', 'beans', 'spinach', 'potato'}
```

difference_update: It is similar to difference. Whereas it updates the result to the first set.

```
Vegetables1.difference_update(Vegetables2)

Vegetables1

>>{'beans', 'cabbage', 'spinach', 'potato'}
```

Lets re-initialize Set1 to carry out other methods.

```
Vegetables1 = {'cabbage', 'spinach', 'beans', 'radish', 'carrot'}
```

intersection: this identifies the common elements between 2 Sets.

```
Vegetables1.intersection(Vegetables2)

>>set()
```

There is no common element and returns a empty set. Lets add some elements from set 1 to set 2

```
Vegetables2.add('cabbage')

Vegetables2.add('spinach')

Vegetables1.intersection(Vegetables2)
```

```
>>{'cabbage', 'spinach'}
```

This difference is a new set result and the original set is not modified.

intersection_update: This is similar to intersection but copies the result to first set.

```
Vegetables1.intersection_update(Vegetables2)

Vegetables1

>>{'cabbage', 'spinach'}
```

union: This produces a new set including elements from both set 1 and 2.

```
Vegetables1

>>{'cabbage', 'spinach'}

Vegetables2

>>{'carrot', 'cabbage', 'spinach', 'milk', 'radish', 'eggs'}

Vegetables1.union(Vegetables2)

>>{'cabbage', 'carrot', 'spinach', 'milk', 'radish', 'eggs'}
```

Lets re-initialize set 1 and set 2

```
Vegetables1 = {'cabbage', 'spinach', 'beans', 'radish', 'carrot'}

Vegetables2 = {'carrot','radish','carrot','eggs"milk'}
```

symmetric_difference: This produces a new set that contains all elements from both sets except the intersecting elements. Does not modify the original sets.

```
Vegetables1.symmetric_difference(Vegetables2)

>>{'beans', 'cabbage', 'spinach', 'milk', 'eggs'}
```

symmetric_difference_update: This creates a set that contains all elements from both sets except the intersecting elements and it assigned to the first set.

```
Vegetables1.symmetric_difference_update(Vegetables2)

Vegetables1

>>{'beans', 'cabbage', 'spinach', 'milk', 'eggs'}
```

Lets re-initialize set 1

```
Vegetables1 = {'cabbage', 'spinach', 'beans', 'radish', 'carrot'}
```

update: This is used to add multiple elements to a existing set

```
Vegetables1.update({'lemon','chilli'})

Vegetables1

>>{'beans', 'cabbage', 'carrot', 'lemon', 'spinach', 'chilli', 'radish'}
```

isdisjoint: If 2 sets are different, there are no single common element then the sets are called disjoint and this functions returns true, else returns false.

```
Vegetables1.isdisjoint(Vegetables2)

>>False
```

issuperset: If all the elements of Set 2 are present in Set 1 in addition to its own elements then Set 1 is called superset of Set 2 Lets re-initialize values of the set to illustrate working of this method.

```
Vegetables1 = {'carrot','brinjal','lemon','tomato"chilli'}

Vegetables2 = {'brinjal','chilli'}

Vegetables1.issuperset(Vegetables2)

>>True

Vegetables2.issuperset(Vegetables1)

>>False
```

issubset: If all the elements of Set 2 are present in Set 1 in addition to its own elements then Set 2 is called subset of Set 1 Lets re-initialize values of the set to illustrate working of this method. The same above example holds good here too, only that we test from Set 2 perspective.

```
Vegetables1 = {'carrot','brinjal','lemon','tomato"chilli'}

Vegetables2 = {'brinjal','chilli'}

Vegetables2.issubset(Vegetables1)

>>True

Vegetables1.issubset(Vegetables2)

>>False
```

copy: This method is similar to the one we saw in list. This creates a new copy of the Set we create from. Any changes made to the original set or the new copy remains independent and does not affect each other.

```
Eggatarian = Vegetables2.copy()

Eggatarian.add('egg')

Eggatarian

>>{'chilli', 'egg', 'brinjal'}

Vegetables2

>>{'chilli', 'brinjal'}
```

DELETE:

pop: This removes first element (from left) from the Set. When there is no more element to be removed throws KeyError.

```
Eggatarian

>>{'chilli', 'egg', 'brinjal'}

Eggatarian.pop()

>>'chilli'

Eggatarian.pop()

>>'egg'

Eggatarian.pop()

>>'brinjal'

Eggatarian.pop()

>>Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'pop from an empty set'
```

remove: This removes a element from a Set by its value. If the value is not found in Set then throws KeyError.

```
Eggatarian

>>{'chilli', 'egg', 'brinjal'}

Eggatarian.remove('egg')

Eggatarian.remove('brinjal')

Eggatarian

>>{'chilli'}
```

```
Eggatarian.remove('chilli')

Eggatarian

>>set()
```

discard: This removes a element from a Set by its value. Similar to remove but this method does not throw KeyError if the value is not found.

```
Vegetables1

>>{'carrot', 'lemon', 'chilli', 'brinjal', 'tomato'}

Vegetables1.discard('carrot')

Vegetables1.discard('carrot')
```

clear: This clears the entire Set by removing all the values.

```
Vegetables2

>>{'chilli', 'brinjal'}

Vegetables2.clear()

Vegetables2

>>set()
```

del: command removes the identifier from memory. Any attempt to access after deleting will throw NameError.

```
del Vegetables1
```

DICTIONARY - DATA TYPE

CREATE:

A dictionary contains pairs of values, similar to a language dictionary. There will be word and a meaning in each pair in a language dictionary. Similarly in python dictionary (we will call it dict) contains key and value as a pair. And a single dict can contains any number of key-value pairs. Keys in a dict has to be unique.

To create a dict, identifier = {}, this will create a empty dict. To create dict with values, identifier = {key1:value1,key2:value2} every key and value has to be separated by a colon (:) and each pair has to be separated by a coma(,)

```
Yield = {"cabbage":25,"brinjal":10,"tomato":5}
```

Note that each Key and a Value can be of any data type. To add a new pair,

```
Yield["lemon"]=10

Yield

>>{'cabbage': 25, 'brinjal': 10, 'tomato': 5, 'lemon': 10}
```

We can use the same method above to edit as well.

```
Yield ["lemon"] = 25
```

ACCESS:

keys: To access all the keys of dictionary.

```
Yield.keys()

>>dict_keys(['cabbage', 'brinjal', 'tomato', 'lemon'])
```

values: To access all the Values in dictionary.

```
Yield.values()

>>dict_values([25, 10, 5, 25])
```

items: Retrieve all pairs of values from dictionary

```
Yield.items()

>>dict_items([('cabbage', 25), ('brinjal', 10), ('tomato', 5), ('lemon', 25)])
```

get: Retrieve value using its key from dictionary.

```
Yield.get('cabbage')

>>25
```

MODIFY:

update: Update a existing key

```
Yield.update({'tomato':20})

Yield

>>{'cabbage': 25, 'brinjal': 10, 'tomato': 20, 'lemon': 25}
```

copy: copies a dictionary into new identifier. Both copies become independent and updates does not affect each other.

Re-initialize dictionary:

```
Yield = {'cabbage':25,'brinjal': 10, 'tomato': 20, 'lemon': 25}

Produce = Yield.copy()

Produce.popitem()
```

```
>>('lemon', 25)

Produce

>>{'cabbage': 25, 'brinjal': 10, 'tomato': 20}

Yield

>>{'cabbage': 25, 'brinjal': 10, 'tomato': 20, 'lemon': 25}
```

fromkeys: creates a new dictionary using keys from an iterable and all values set to None or a default same value.

```
fert = ['tomato','brinjal','potato']

fert_dict = dict.fromkeys(fert)

fert_dict

>>{'tomato': None, 'brinjal': None, 'potato': None}

fert_dict = dict.fromkeys(fert,100)

fert_dict

>>{'tomato': 100, 'brinjal': 100, 'potato': 100}
```

setdefault: Returns value of a key if present, else inserts key and sets value to None.

```
fert_dict.setdefault('tomato')

>>100

fert_dict.setdefault('ginger')

>>{'tomato': 100, 'brinjal': 100, 'potato': 100, 'ginger': None}
```

DELETE:

pop: Remove a key value pair from dictionary, using key as input.

```
Yield.pop('cabbage')

>>25

Yield

>>{'brinjal': 10, 'tomato': 20, 'lemon': 25}
```

popitem: Removes the last element from dictionary. When there is no element to remove throws KeyError

```
Yield.popitem()
```

```
>>('lemon', 25)
Yield.popitem()
>>('tomato', 20)
Yield.popitem()
>>('brinjal', 10)
Yield.popitem()
>>Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

clear: Deletes all items from dictionary.

```
Produce.clear()
Produce
>>{}
```

Data Type Conversions

Data type conversions can be of 2 types,

1) Implicit conversion and

2) Explicit conversion

Below are some of the explicit conversions on the basic data types.

Following is a table of conversions possible from source data type to other data types.

Source Data Type	Target Data Type -1	Target Data Type -2	Target Data Type -3
Integer	Float	String	Bool
Float	Integer	String	Bool
String	Integer	Float	Bool
Bool	Integer	Float	String

Example Values in place of above Data Types. Apply the above 4X4 matrix below. Eg. In first row,

1 is attempted conversion to Float, String & Boolean and results are filled in table.

Source Data Type	Target Data Type -1	Target Data Type -2	Target Data Type -3
1	1.0	"1"	True
1.0	1	"1.0"	True
"1"	1	ValueError	True
False	0	1.0	"False"

To note, whenever there is no valid conversion possible the statement results in an error and python program stops execution.

Refer below conversion codes and results Try for yourself below commands to understand better.

```
#string to integer
int("1500")
>>1500
int("hundred")
>>Traceback (most recent call last):
File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hundred'
#string to float
float("1500")
>>1500.0
float("hundred")
>>Traceback (most recent call last):
File "<input>", line 1, in <module>
```

```
ValueError: could not convert string to float: 'hundred'
#string to boolean
bool("hundred")
>>True
bool('')
>>False
#integer to float
float(1500)
>>1500.0
#integer to string
str(1500)
>>'1500'
#integer to boolean
bool(100)
>>True
bool(0)
>>False
#float to integer
int(1500.8)
>>1500
#float to string
str(1500.8)
>>'1500.8'
#float to boolean
bool(1500.8)
>>True
bool(0.0)
>>False
```

```
#boolean to integer
int(True)
>>1
int(False)
>>0
#boolean to float
float(True)
>>1.0
float(False)
>>0.0
#boolean to string
str(True)
>>'True'
str(False)
>>'False'
```

Making calculations with Operators

Operators are mathematical operations, these are used in conditional and looping statements to make decisions.

Find below the list of operators under different types.

Arithmetic	- + - / * %
Relational	- < > < = > = == !=
Logical	and or not
Bitwise	& | ~ ^ >> <<
Assignment	Applies to arithmetic & bitwise (+= -= *= etc)
Membership	in, not in
Identity	is, is not

Lets see some examples for **Arithmetic operators**.

```
#Arithmetic Operators
a = 5
b = 10
print(a + b)
>>15
print(a - b)
>>-5
print(a * b)
>>50
print(a / b)
>>0.5
print(a % b)
>>5
```

Outcome of **Relational operators** will be either True or False

```
#Relational Operators
a = 5
b = 10
```

```
a < b
>>True
a > b
>>False
a <= b
>>True
a >= b
>>False
a != b
>>True
```

Logical Operators follow the truth table of **AND OR NOT**

AND Truth Table

Input 1	Input 2	Output
True	True	True
True	False	False
False	True	False
False	False	False

OR Truth Table

Input 1	Input 2	Output
True	True	True
True	False	True
False	True	True
False	False	False

NOT Truth Table

Input	Output
True	False
False	True

```
#Logical Operators
True and False # logical and
>>False
True or False # logical or
>>True
True and True # logical and
>>True
```

Bitwise operators perform bitwise computation on the provided decimal inputs.

This also follow the Truth table but decimal inputs are converted to binary,

Applies bitwise computation to each bit and the final binary data is converted back to decimal.

```
#Bitwise Operators
2 & 4 # bitwise and
>>0
2 & 6 # bitwise and
>>2
2 | 4 # bitwise or
>>6
2 | 6 # bitwise or
>>6
```

Assignment operators perform certain computations on a source data and assigns back the result to the same identifier.

This is applicable to all Bitwise and arithmetic operators.

```
#Assignment Operators
a = 5
b = 10
a += b # assignment operator with addition
# Above statement is equivalent to "a = a + b"
print(a)
>>15
```

Membership operator is used to identify if a element being searched is present in a collection or not. Collection may represent any of derived data type.

Outcome of Membership operator is True or False

```
fruits = ["apple","orange","grape"]
"grape" in fruits
>>True
"guava" in fruits
>>False
"guava" not in fruits
>>True
```

Identity operator is used to check if two different elements are one and same.

```
flower = "rose"
"rose" is flower
>>True
"jasmine" is flower
>>False
"jasmine" is not flower
>>True
```

Conditional Statements

So far we have been writing statements in a single line. And it was linear, every action is performed one after another.

There is no decision making involved. Suppose we had to perform one of 2 different actions we need to make a decision at that point. Such statements are called decision making statements which leads to different outcomes (or paths of execution)

Suppose you are asked to sow “Beans” if soil PH is greater than 6.0 and if PH less than 6.0 then sow sweet potatoes. We are only going to use a print statement to indicate decision to sow which vegetable.

To implement this recollect the Input – Process – Output from lesson 1.1 Do you remember that system.

Here Inputs are going to be PH value,

Process will be a decision statement that we will define now,

And output is the action to be taken, we will print our decision to the console.

Types of Conditional Statements:

if else condition:

We will also read input from the user in run time.

```
PH = float(input(“Please enter the PH value of soil”)) # 1)

if PH > 6.0: #2)

print(“Plant Beans”) #3)

else: #4)

print(“Plant sweet potato”)
```

1) 'input' is the keyword that prompts the user in the console to enter a input. The program waits until a RETURN is received from keyboard.

2) 'if' is the keyword to define conditional statement. It has to be terminated by a colon : character. 'if' is followed by a relational or any logic operators leading to a boolean output.

3) Followed by 'if', all lines that should execute when the condition satisfies to True are to be *indented* with spaces like this specific line, there has to be minimum 1 space character additional to the line starting 'if'. Usual is a 4 space additional to the 'if' line.

4) An else statement and lines belonging to it are optional. An 'else' line has to start in the same column number as the 'if' line statement. This portion executes only when the condition defined in 'if' evaluates to False.

So the above if statement leads to 2 different decision path. For a given value of PH only one path shall execute.

if elif else: We will see some extensions from this if else statement. Suppose my requirement becomes complex and I start defining multiple conditions,

Suppose you are asked to

- sow “Beans” if soil PH is greater than 6.0 and min temp is greater than equal to 20.0
- Sow “Sweet potatoes” if PH less than 6.0 and min temp is greater than equal 23.0
- Sow “Potatoes” if both conditions do not match. We are only going to use a print statement to indicate decision to sow which vegetable. We will use elif to check multiple conditions.
- ```
  PH = float(input(“Please enter the PH value of soil”))
  MINT = float(input(“Please enter the MINT value”))
  if PH > 6.0 and MINT > 20.0 then #1)
  Print(“Plant Beans”)
  elif PH < 6.0 and MINT > 23.0 then
  Print(“Plant Sweet Potatoes”)
  else: #2)
  Print(“Plant Potato”)
  ```

1) Similar to a 'if else' statement, only one condition executes in 'if elif else'. If multiple condition becomes True, the first condition that evaluated True, its statements will only be executed.

2) When both 'if' and 'elif' conditions evaluates to False, 'else' block when defined executes.

Indentation: is applicable to certain keywords only, all decision, conditional, looping, functions, class, exception its statements have to be indented.

Looping Statements
  ```

Looping statements are used to repeat certain tasks or group of statements multiple times. This is behavior will be required in certain places and you will get more clarity as we will implement a loop statement.

Loop statement can be implemented using two ways,

1) While loops

2) For loops

While loops:

Suppose I want to print numbers from 1 to 10. Inputs are integers, print step will be repeated here. To achieve this we can write as following,

```
i = 0

while i < 10:

print(i)

i = i + 1
```

Here print and increment statement are inside a loop which is repeated multiple times using the while keyword. Lets look at one more similar example,

Suppose I want to water the plants until the soil humidity is up to required level. An automatic sprinkler is configured to water the area. And I know the humidity will reach required level in 10 mins. (We will simplify the solution to only consider the time duration to switch on and off the sprinkler. Current time, sprinkler system – On/Off

Note : We will need to use

```
import time; time.time()
```

to fetch the current time value. Do not worry how this works for now. Time is one of inbuilt function present in python. And also here time is treated as a increasing float value representing seconds.

```
Sprinkler_on = True

10_min = 60 * 10

start_time = time.time()

while time.time() - start_time < 600:

print(“Sprinkling”)
```

```
time.sleep(5)

else:

Sprinkler_on = False
```

for loops: To access and print all seed names in list one by one. for loops can be used in 2 ways.

1) Iterate by value - Access subsequent value present inside the list in each iteration. An iteration is one processing of the list item in every looping cycle.

2) Iterate by index - Access subsequent index position present inside the list in each iteration. Using the index position list item can be accessed further. An iteration is one processing of the list item in every looping cycle.

```
seed_names = [‘tomato’,’brinjal’,’lemons’,’chilli’]

for seed in seed_names: #1)

print(seed)

for i in range(len(seed_names)): #2)

print(seed_names[i])
```

1) Iterate by value method

2) Iterate by index method

break: This command is used to terminate any loop.

```
seed_names = [‘tomato’,’brinjal’,’lemons’,’chilli’]

for seed in seed_names: #This loop breaks when "lemons" is encountered.

if seed == 'lemons':

break

print(seed)
```

continue: This command is used to skip processing specific iteration of a loop variable.

```
seed_names = [‘tomato’,’brinjal’,’lemons’,’chilli’]

for seed in seed_names: #This loop skips processing when "lemons" is encountered.

if seed == 'lemons':

continue
```

```
print(seed)
```

Functions & it’s types

Functions:

A function is used to group together statements and reuse it from any where in the program. This brings easier organization of code and reusability.

A function can be of different types based on the inputs supplied to the program.

First lets understand a simple function.

```
def printplantname(name): #1) 2) 3)

print(name) #4)

return #5)
```

1) def is the keyword to represent function definition

2) followed by 'def' is a name given to the function by the user. It follows the same rule as naming the identifier.

3) In the same line after function name is the input values referred as arguments to the function.

4) This is a example line indented to show a sample process or step that can be written inside a function.

5) After processing, a function exits by default when there are no more indented lines read. A function also exits when it encounters a 'return' statement. Any values if present in a return statement are sent to the function calling line as well.

The different types of arguments are

1) Positional Arguments

2) Keyword Arguments

3) Arbitrary Arguments

Positional arguments:

Any function when called with values and it is matched based on the position of arguments defined in function definition, those arguments are called positional.

```
def plant_seeds (spacing, seed_name):

print(spacing, seed_name)

# Function call

plant_seeds ("5cm","tomato")
```

In above code "5cm" corresponds to spacing & "tomato" corresponds to seed_name. The values are assigned to name arguments based on the position order.

1) Order for a positional argument matters, because values are assigned based on positions only.

2) When few values are not passed in function call, it becomes a invalid call. Number of positional values passed should match number of positional arguments present in definition.

Keyword arguments:

A keyword argument is created by assigning default values to argument during function creation.

In a keyword argument values are passed along with argument name in a function call.

```
def plant_seeds (spacing, seed_name="chilli"):

print(spacing, seed_name)

# Function call

plant_seeds ("5cm",seed_name="tomato")
```

Now seed_name becomes a keyword argument.

1) Order for a keyword argument doesn’t matter, but it should appear after positional arguments only.

2) When no values are supplied during function call for a keyword argument, it is still a valid function call. The default values present in function definition will be used.

```
def plant_seeds (spacing, seed_name="chilli", area='10sqft'):

print(spacing, seed_name)

# Function call
```

```
plant_seeds ("5cm",area='20sqft',seed_name="tomato")
```

Arbitrary arguments: An arbitrary argument is used to receive variable number of inputs. In other types the number of values passed in calling statement has to be equal in case of positional and equal or less in case of keyword. But for a arbitrary the number of values passed doesn't matter, all the values will be combines and provided as a single combined data type, as a tuple.

First lets understand how a tuple is received,

```
def collect_produce(*vegetables):
print(type(vegetables))
for veg in vegetables:
print(veg)



collect_produce("carrot","potatoes","brinjal")
>><class 'tuple'>
>>carrot
>>potatoes
>>brinjal



collect_produce("carrot")
>><class 'tuple'>
>>carrot
```

In the above example an arbitrary argument is represented by defining a usual argument but along with a '*' symbol. This is the only change required.

When called with two different set of inputs for both outputs were produced above.

This is arbitrary argument used for a positional kind of values.

The same arbitrary args can be applied to a keyword kind as well. In which case that argument has be represented by double ** along with a usual argument.

Read the code below and explanation further,

```
def collect_produce(**vegetables):

print(type(vegetables))

for veg in vegetables.items():

print(veg)



collect_produce(veg1="carrot",yield1="5kg",veg2"potatoes",yield2="10kg",veg3="brinjal",yield3"7kg
")

<class 'dict'>

('veg1', 'carrot')

('yield1', '5kg')

('veg2', 'potatoes')

('yield2', '10kg')

('veg3', 'brinjal')

('yield3', '7kg')



collect_produce(veg1="carrot",yield1="5kg")

<class 'dict'>

('veg1', 'carrot')

('yield1', '5kg')
```

Here the argument vegetable now acts as a dictionary because of the ** and it can receive key pair values from the function call.

Special purpose functions

lambda:

lambda is a keyword to define a function in one single line. It is not mandatory to use this kind of function in any place in using the language. But it offers interesting ways to write our code.

Followed by the lambda keyword is any argument for this function and separated by comas in case of multiple arguments.

After that is a colon : and followed by any calculation or operation that needs to be performed. It can be any valid python statement.

Below is an example definition of one such function to Calculate total price of vegetable produce. Multiple cost per kg * quantity to calculate the output.

```
#per kg 12
cost = 12



produce = lambda quantity:cost*quantity



result = produce(7)
print(result)
>>84
```

Here produce is the function created because of the lambda statement. And it can be used as any normal function. Call that by passing the values it expects. The result is stored into a variable and printed in next line.

filter:

A filter is used when we want to separate specific values from a collection (derived data types, it has to be specifically). For examples, there are weight of coconuts stored in a list. And you want to separate coconuts into 2 separate categories based on its weight. Less than 600 gram as category 1 and greater than 600 gram as category 2.

```
# store in grams
coconuts = [450,500,650,800570,590580,650]
category1 = list(filter(lambda wt:wt <= 600,coconuts)) #1)
```

```
print(category1)

>>[450, 500, 570, 590, 580]

category2 = list(filter(lambda wt:wt > 600,coconuts)) #2)

print(category2)

>>[650, 800, 650]
```

An iterable is any datatype which has multiple values and can be accessed one by one using an iterator loop.

In above code lets understand 1) & 2)

1) list converts the object inside it into a list datatype. Here is a filter function created by filter keyword. The filter function takes 2 arguments - one a user defined function that returns a boolean value and two the input iterable collection.

- The user defined function can be a regular function written using def keyword or a lambda function as well. But the return from function should be either True or False. If returned True then the particular value is stored into filter object, for false those values are not stored.
- So obviously length of output result can be less than or equal to input length.

2) Similarly filter for wt greater than 600 gram and its result output is shown as well.

map:

A map is another special function used to apply a user defined function to transform values of a collection.

Example, if there are weights of apples available after measurement, now to arrive final weight of an apple we have to add packaging weight to each apple. The packaging weight is given as 1.5 g. Produce a new list with new weights.

```
weigts = [450,500,480,495600]



new_weights = list(map(lambda wt:wt+1.5, weights))
```

```
print(new_weights)
```

```
>>[451.5, 501.5, 481.5, 496.5, 601.5]
```

map function takes 2 arguments, one the user defined function that reads values from a collection applies the function defined to each element of collection and returns a new value to filter object. This filter object is converted to readable values as a list. This output length will be equal to the input collection data type length.

Object Oriented Programming (OOP)

Object oriented programming is one of the most important aspect of a programming language, because it implements principles in writing a program or code. In short OOP, principles are implemented using class and objects concept. We will see in a while how class and objects are created using class keyword.

Concepts of object oriented programming are,

1) Encapsulation

2) Abstraction

3) Inheritance

4) Polymorphism

Encapsulation

Encapsulation represents the basic property of Object oriented programming (OOP), where variables (can be called as attributes) and functions (called as methods) are grouped together as one single entity (group of relevant items). Creation of such an entity is achieved using the class keyword. Lets create a class.

```
class Farm:

area = "5acre"
```

```
def get_area(self):
return(self.area)


Farm.area
>>'5acre'
```

Here followed by class keyword is any user defined name for the class, and then a colon :

All items inside class are also indented, to represented they all are one single entity.

This class contains attributed and methods as follows, attribute - area method - get_area

After creation of this class by executing, there will be no output. We have to either make use of class name or object to access its attributes and methods.

We directly used the class name to access attribute area and printed it. If you have observed about the method, there is an argument 'self' and also in the return statement area is accessed using self. self represents that whatever being accessed belongs to the same class and its own property. It is mandatory to use this keyword in this way.

- Every class method expects a self argument by default.
- Every attribute of class has to be accessed using the self keyword.
- The operator '.' is used to access members of an entity.

Lets try accessing the method.

```
class Farm:
area = "5acre"


def get_area(self):
return(self.area)
```

```
Farm.get_area()

>> TypeError: get_area() missing 1 required positional argument: 'self'
```

Accessing methods requires a different approach, we need an object for this class to be able to access the methods. An object is a copy of a class, any number of objects can be created for one single class.

Now lets create a object by calling the class and access the method. The object can be given any user defined name.

```
objFarm = Farm()

print(objFarm.get_area())

>>'5acre'
```

Passing Values to class

Whenever a class is called (), an object for the class is created by using a default constructor function. We don’t need to do define anything. This constructor function name is init

But if we want to provide inputs to our class while calling, then this constructor has to be defined explicitly. That will override the default constructor.

```
class Farm:

def __init__(self,area,crop)

self.area = area

self.crop = crop



def get_details(self):

return(self.area,self.crop)



objFarm = Farm("5acre","rice") #1)
```

In this case Farm class constructor, function *init* is defined and it is created with 2 arguments 'area' and 'crop'. 1) These values passed in class object creation step is received by this constructor. It is stored into a class attribute (self.area, self.crop) so that same can be accessed in get_details method as well.

Abstraction

The property of abstraction is hiding certain features to others. Usually this achieved using access modifiers in other languages cpp, java, c# etc. Access modifiers define the visibility for an identifier like public, private or protected. But in python this behavior is not implemented in the language. By default all remain visible in public as long as it is available.

Inheritance

The property of inheritance provides ability to make use of the properties of one class inside another class. For example there is a class that contains area, crop and get_details() this is accessible from Farm class only. If I want to use these features in another class Marketplace I no need to create all of these again inside the Marketplace class, I can simply inherit Farm class into Marketplace.

Before understanding the above example lets see how a class gets created in first place, because that is based on inheritance principle only. Every class is created by inheriting the object class only. That is not required to be mentioned to the interpreter, whenever we use the class keyword that happens implicitly. To inherit we have to use the 1st class name also referred the parent or base class inside the 2nd class name also referred as the child class or derived class definition. Note, that the base class definition can be present elsewhere or inside the same python file.

The below definition shows inheriting object class inside Myclass, which is implicit and not required to be performed for class definition.

```
class MyClass:
pass



class MyClass(object):
pass
```

The object class brings along with it three methods. The three methods are - new_(), </emphasis>init() and </emphasis>str(). </emphasis>new() will not require any modification & it takes responsibility for creating our object. </emphasis>init() and </emphasis>str__() methods will have no definition and we have to override in our user defined class to meaning to this. init is the constructor that we just saw under the introduction how to pass values to a class. str is another method through which we can provide a string descriptor to our class. Whenever the class object is printed the value returned by this str method is returned.

```
class Farm:

def __init__(self,area,crop,landid)

self.area = area

self.crop = crop

self.landid = landid



def get_details(self):

return(self.area,self.crop)



def __str__(self): #3)

return(self.landid)



objFarm = Farm("5acre","rice","DIST101") #1)

print(objFarm) #2)

>>'DIST101'
```

1) I have passed a 'DIST101' as a unique identifier during object creation step

2) When attempted to print the object it prints the unique identifier

3) The str definition returns the landid or any identifier that we want to map with an object instance.

Types of Inheritance

There are different ways to achieve inheritance. The types are

1) Single level inheritance

2) Multi level inheritance

3) Multiple inheritance

Single Level Inheritance

To understand this type, we will create one base class and inherit it to child class. Lets define the Farm class and inherit it into Marketplace class.

```
class Farm: #1)
def __init__(self, quantity, crop, landid):
self.quantity = quantity
self.crop = crop
self.landid = landid
return



def get_farm_details(self):
return (self.quantity, self.crop)



def __str__(self):
return (self.landid)




class Marketplace(Farm): #2)
def __init__(self, quantity, crop, landid, storeid, price):
Farm.__init__(self, quantity, crop, landid)
self.storeid = storeid
```

```
self.price = price
return


def get_price_details(self):
return(self.crop, self.price)


def __str__(self):
return self.storeid


# 3)
objFarm = Marketplace("5kg", "carrots", "DIST101", "STR199", "400")
print("Quantity: ",objFarm.quantity)
print("Crop: ",objFarm.crop)
print("landID: ",objFarm.landid)
print("storeID: ",objFarm.storeid)
print("Price: ",objFarm.price)
print("Farm Details: ",objFarm.get_farm_details())
print("Price Details: ",objFarm.get_price_details())


>>Quantity: 5kg
>>Crop: carrots
>>landID: DIST101
>>storeID: STR199
>>Price: 400
>>Farm Details: ('5kg', 'carrots')
```

```
>>Price Details: ('carrots', '400')
```

1) Create Farm as the base class

- Define constructor with 3 arguments, quantity, crop & landid
- Define get_farm_details and print quantity and crop to the console
- Define str method and return landid

2) Create Marketplace as child class and inherit Farm into it using brackets as above.

- Define constructor with 5 arguments, quantity, crop, landid, storeid, price
- Initialize the Farm class constructor by calling it inside this constructor and send all required values along with it
- Define get_price_details and print crop and price to the console
- Define str method and return storeid In all the class methods mandatorily define the self argument. Also define self for class attributes that should be available for use by the object name.

3) Create an Object for Marketplace giving all inputs. With the created object access attributes and methods of both the parent class and child class.

Multi level Inheritance

We can also inherit multiple levels of class hierarchy. In the previous case there was only one level of hierarchy and it was called single level. To understand multiple level lets define a example where we will inherit twice. And from the lowest child we will be able to access all the parent as well as its own attributes and methods.

```
class Farmer: #1)

def __init__(self,name,age,location):

self.name = name

self.age = age

self.location = location



def get_farmer_details(self):
```

```
return (self.name,self.age,.location)



def __str__(self):
return (self.name)



class Farm(Farmer): #2)
def __init__(self, name, age, location, quantity, crop, landid):
Farmer.__init__(self,name,age,location)
self.quantity = quantity
self.crop = crop
self.landid = landid
return



def get_farm_details(self):
return (self.quantity, self.crop)



def __str__(self):
return (self.landid)




class Marketplace(Farm): #3)
def __init__(self, name, age, location, quantity, crop, landid, storeid, price):
Farm.__init__(self, name, age, location, quantity, crop, landid)
self.storeid = storeid
```

```
self.price = price
return


def get_price_details(self):
return (self.crop, self.price)


def __str__(self):
return self.storeid


#4)
objFarm = Marketplace("Barani","31","Tamil Nadu","5kg", "carrots", "DIST101", "STR199", "400")
print("Name: ", objFarm.name)
print("Age: ", objFarm.age)
print("Location: ",objFarm.location)
print("Quantity: ",objFarm.quantity)
print("Crop: ",objFarm.crop)
print("LandId: ",objFarm.landid)
print("StoreId: ", objFarm.storeid)
print("Price: ", objFarm.price)
print("Farmer Details: ", objFarm.get_farmer_details())
print("Farm Details: ", objFarm.get_farm_details())
print("Price Details: ", objFarm.get_price_details())


>>Name: Barani
>>Age: 31
```

```
>>Location: Tamil Nadu
>>Quantity: 5kg
>>Crop: carrots
>>LandId: DIST101
>>StoreId: STR199
>>Price: 400
>>Farmer Details: ('Barani', '31', 'Tamil Nadu')
>>Farm Details: ('5kg', 'carrots')
>>Price Details: ('carrots', '400')
```

1) Create Farmer as the base class

- Define constructor with 3 arguments, name, age & location
- Define get_farmer_details and return name, age and location when the function is called
- Define str method and return name

2) Create Farm as the first child class and inherit Farmer into it as shown above.

- Define constructor with 6 arguments, name, age, location, quantity, crop & landid
- Define get_farm_details and print quantity and crop to the console
- Define str method and return landid

3) Create Marketplace as second child class and inherit Farm into it using brackets as above.

- Define constructor with 8 arguments, name, age, location, quantity, crop, landid, storeid, price
- Initialize the Farm class constructor by calling it inside this constructor and send all required values along with it
- Define get_price_details and print crop and price to the console
- Define str method and return storeid In all the class methods mandatorily define the self argument. Also define self for class attributes that should be available for use by the object name.

4) Create an Object for Marketplace giving all inputs. With the created object access attributes and methods of both the parent class and all child classes.

Multiple Inheritance

A multiple inheritance takes the approach of inheriting simultaneously 1 or more base class into a child class. The requirement for a multiple approach is that base classes need not be related to each other but they may be related to the inheriting class individually. Creating hierarchy in such cases will not be meaningful.

```
class Farmer: #1)
def __init__(self,name,age,location):
self.name = name
self.age = age
self.location = location
return


def get_farmer_details(self):
return (self.name,self.age,.location)


class Fertilizer: #2)
def __init__(self,fertilizer,quantity):
self.fertilizer = fertilizer
self.quantity = quantity
return


def get_fert_details(self):
return (self.fertilizer,self.quantity)
```

```
class Farm(Farmer,Fertilizer): #3)
def __init__(self, name, age, location, fertilizer, quantity,area, crop):
Farmer.__init__(self,name,age,location)
Fertilizer.__init__(self, fertilizer, quantity)
self.area = area
self.crop = crop
return


def get_farm_details(self):
return (self.area, self.crop)


#4)
objFarm = Farm("Barani","31","Tamil Nadu","phosphate""5kg","2acre""carrots")
print("Farm Details: ",objFarm.get_farm_details())
print("Fertilizer Details: ",objFarm.get_fert_details())
print("Farmer Details: ",objFarm.get_farmer_details())
```

1) Create Base class Farmer

- Accept values in constructor, name, age and location.
- Define get_farmer_details method and return all its class attributes.

2) Create one more base class Fertilizer

- Accept values in constructor, fertilizer and quantity.
- Define get_fert_details method and return all its class attributes.

3) Create class Farm and inherit Farmer & Fertilizer class as shown separated by commas in the class definition.

- Accept values name, age, location, fertilizer, quantity,area, crop in its constructor.
- Initialize constructor of Farmer and Fertilizer inside its constructor.
- Define get_farm_details method and return area and crop.

4) Create an object for Farm derived class and access all methods of parent and self from this object.

Polymorphism

The property of polymorphism is that one occurring in several different forms. With respect to object oriented programming, a same object displaying different behavior based on its circumstances. We shall have a class method in each of child and parent and also in the same name. In this case the child method and parent method definitions can be different thereby exhibiting different behavior.

Consider the below example. There are 2 class defined, Farm and it is inherited into Marketplace. Both has get_id() method returning different values.

```
class Farm: #1)

def __init__(self, quantity, crop, landid):

self.quantity = quantity

self.crop = crop

self.landid = landid

return



def get_farm_details(self):

return (self.quantity, self.crop)



def get_id(self):

return (self.landid)
```

```
class Marketplace(Farm): #2)
def __init__(self, quantity, crop, landid, storeid, price):
Farm.__init__(self, quantity, crop, landid)
self.storeid = storeid
self.price = price
return


def get_price_details(self):
return(self.crop, self.price)


def get_id(self):
print(self.storeid)
return


def get_id_parent(self):
print(super().get_id())
return


# 3)
objFarm = Marketplace("5kg", "carrots", "DIST101", "STR199", "400")
objFarm.get_id()
objFarm.get_id_parent()
```

1) Create Farm class

- Define constructor and accept quantity, crop and landid as inputs
- Define get_farm_details method and return quantity and crop details
- Define get_id method and return landid

2) Create Marketplace class

- Define constructor and accept quantity, crop, landid, storeid, price as inputs
- Define get_price_details method and return crop and price details
- Define get_id method and return storeid
- Define get_id_parent method and use - super().get_id() to access get_id of its parent. The super keyword provides access to elements present in the parent.
- super() method refers to the immediate parent class only. So if a super method was used in a multi level inheritance we cannot access method present in parent that are 2 levels above itself.

3) Create an object for Marketplace, access get_id and get_id_parent methods to see the different results of child and parent respectively.

Class Method Types

Methods inside a class can be of 3 different types,

1) Instance Methods

2) Static Methods

3) Class Methods

Each has different access and lets understand them.

Instance Method

Any method that is created inside a class and that has a self argument inside are instance method. In most examples in previous section the methods created are instance methods. - All class variables will be accessible inside an instance method. - An object created for a class will be able to access instance method.

Static Method

A static method is created inside a class with help of a decorator @staticmethod.

- In presence of decorator @staticmethod self argument is not supported and cannot be given as an argument.
- Class variables will not be accessible inside static method. Doing so will throw NameError.
- Static method can be accessed by class directly. It can also be accessed by instance objects.
- Static methods are mostly used to define utility functions inside a class that do not depend on other variables of the class.

In below example create Marketplace class and create instance methods and static methods. Use static methods to create and access unit conversion utilities.

```
class Marketplace(): #1)
def __init__(self, quantity, product, perkgcost):
self.quantity = quantity
self.product = product
self.perkgcost = perkgcost
return



def get_price_details(self): #4)
grams = Marketplace.kgtograms(self.quantity)
price = Marketplace.pergramcost(self.perkgcost)
return(grams*price)



@staticmethod #2)
def kgtograms(kg):
return (kg*1000)
```

```
@staticmethod #3)
def pergramcost(cost):
return (cost/1000)




obj = Marketplace(3.5,"carrots",80)
print(obj.get_price_details()) # 5)
print(obj.kgtograms(3)) # 6)
>>280.0
>>3000
```

1) Create constructor accepting values quantity, product and perkgcost as inputs.

2) Create static method kgtograms using the decorator. This method can be accessed outside the class using its object as well as the class name directly. Return kg to gram converted value.

3) Create static method pergramcost using the decorator. This method can be accessed outside the class using its object as well as the class name directly. Returns price of 1 g.

4) Calculate price of the product using per gram cost and total quantity in grams inside get_price_details. To access static method that are inside the class itself class name has to be used as in Marketplace.kgtograms(self.quantity).

5) Create object for Marketplace and call the instance method.

6) Access static method using instance object and see the output.

Class Method

A class method is created using decorator @classmethod.

- We need to pass argument cls to a class method.
- This method cannot access regular instance attributes and methods because the self argument is not available to this.
- Purpose of class method is to create a instance object. This can be seen as an alternate constructor another version of init function.

- Returning a cls object will create the instance object.

In below example create Marketplace class and create class methods.

```
class Marketplace(): #1)
def __init__(self,product, quantity, cost, expiry):
self.product = product
self.quantity = quantity
self.expiry = expiry
self.perkgcost = cost
return



def get_product_details(self): #6)
grams = self.kgtograms(.quantity)
price = self.pergramcost(.perkgcost)
print("Item: ",self.product)
print("Price: ",grams*price)
print("Expiry days: ", self.expiry)
return



@classmethod #2)
def vegetables(cls,product, quantity, cost):
expiry = 7
return cls(product, quantity, cost, expiry)



@classmethod #3)
def fruits(cls,product, quantity, cost):
expiry = 3
```

```
return cls(product, quantity, cost, expiry)



def kgtograms(self,kg): #4)
return (kg*1000)



def pergramcost(self,cost): #5)
return (cost/1000)



# 7)
obj = Marketplace("Egg",2,80,14)
obj.get_product_details()



# 8)
obj1 = Marketplace.vegetables("carrots",3.5,80)
obj1.get_product_details()



# 9)
obj1 = Marketplace.fruits("mango",3.5,100)
obj1.get_product_details()



>>Item: Egg
>>Price: 160.0
>>Expiry days: 14
>>Item: carrots
```

```
>>Price: 280.0
>>Expiry days: 7
>>Item: mango
>>Price: 350.0
>>Expiry days: 3
```

1) Create a Marketplace class and constructor accepting values product, quantity, cost, and expiry.

2) To have different variations of this class, create method using the decorator @classmethod, get required input values as arguments to this function, perform processing and return the output as a cls object.

- cls object returned should have values same as *init* constructor inputs.
- This will be the new object similar to a class instance and using which all attributes and methods can be accessed.
- Create a first classmethod vegetables. cls should be the default argument and followed any other inputs - product, quantity, cost.
- For example to illustrate, expiry is changed for a class instance produced using vegetable classmethod. Assigned a value 7 by default.
- Return the cls object with product, quantity, cost & expiry.

3) Create the second classmethod fruits with arguments cls, product, quantity, cost.

- Similar to vegetable method, create a expiry attribute with value 3 here. Every class instance created using fruit will have this value.
- Return the cls object with product, quantity, cost & expiry. This will produce the class instance and can access all other attributes and methods present.

4) Write a instance method kgtograms.

5) Write a instance method pergramcost.

6) Write get_product_details instance method. Print Item, Price and Expiry Days to console when this is called.

7) Create an object for Marketplace, this accepts 4 values - product, quantity, cost, expiry. Access the get_product_details using this object and see the

output.

8) Create an object using Marketplace.vegetables class method. This accepts 3 input values - product, quantity, cost. Access the get_product_details using this object and see the output.

9) Create an object using Marketplace.fruits class method. This accepts 3 input values - product, quantity, cost. Access the get_product_details using this object and see the output.

Unified Modeling Language (UML) is used to represent the class and relations between them graphically. It helps to document and also to communicate software design with other developers and product owners. There are both free to use and propreitary tools to implement. But the language paradigm for UML is a industry standard. you can draw that on a power point presentation also.

Specific Features of Python

Some of the following features may or may not be available in other languages. Thats why it is separated into a different section. Lets understand each of them. These are good to have features in a program.

Iterator

Recollect the concept of iteration in loops. The loop identifier reads collection values in each iteration. Without using the loop identifier still it is possible to iteratively access elements of a collection and that is made possible with help of iterator object. We will see how to creator such a object and use.

```
l = ["roses","tulip","lavendar","hibiscus"]

iterObj = iter(l)



iterObj.__next__()

>>'roses'

iterObj.__next__()

>>'tulip'

iterObj.__next__()
```

```
>>'lavendar'
iterObj.__next__()
>>'hibiscus'
iterObj.__next__()
>> Traceback (most recent call last):
>> File "/Users/barani/PycharmProjects/trading/tradevenv/lib/python3.8/
>> site-packages/IPython/core/interactiveshell.py", line 3418, in run_code
>> exec(code_obj, self.user_global_ns, self.user_ns)
>> File "<ipython-input-9-8674e40a217e>", line 1, in <module>
>> iterObj.__next__()
>>StopIteration
```

In above example iterator object is created with keyword iter and within brackets a collection data is provided as input. Now, using the iterator object dot *next*() the collection can be iterated. It is seen individual value is accessed each time and upon exhausting the collection it throws an error Stop Iteration.

A loop operation works similar to this iterator but it doesn't throw the Stop iteration error.

Generator

A generator can be defined as a function that returns output multiple times. If you remember in a normal function whenever return is encountered the function exits with return values if provided anything. A value can be returned back from a function using keyword **yield**, this doesn't cause exiting the function but keep continuing the execution further if there are any lines left inside the function. Usage of keyword yield inside a function to achieve this behavior makes it a generator function.

Since outputs are received back from a function asynchronously we have to access this function through a loop, only then the generator function executes.

```
def user_genertaor(vegetables):
for veg in vegetables:
if veg.find('carrot') >= 0:
yield(veg,40)
```

```
elif veg.find('brinjal') >= 0:
yield(veg,30)
elif veg.find('potatoes') >= 0:
yield(veg,45)
return



vegetables = ['carrot','brinjal','potatoes']
for v in user_genertaor(vegetables):
print("price of {0} is {1}".format(v[0],v[1]))



>>price of carrot is 40
>>price of brinjal is 30
>>price of potatoes is 45
```

We passed a list input (vegetables) to generator function and this starts execution when the for loop starts execution. Every time yield statement executes, the output is sent back to for loop variable and continues further execution.

If you wonder what purpose it serves, the processing inside the generator function does not wait for the complete list to be processed to send the values. As and when one element of list is processed it is sent back to the generator calling place. This saves time if the list to be processed is big.

Decorator

The next in this chapter is the concept of decorator. This feature provides ability to modify behavior of a existing function. The behavior can be modified in few ways, by receiving the inputs and manipulating it before calling the actual function or decide not to call that specific function, instead perform a different action. Write log results etc. There are many possibilities and use cases once you understand how a decorator works.

```
import functools
```

```
def add(a,b): #1)
"""
This is add function
"""
print(a+b)
return(a+b)


def modified_add(userAdd): #2)
@functools.wraps(userAdd) #6)
def wrapper(*args):
"""
This is a decorator function
"""
print("Start of function")
args = tuple([i+ 1.5 for i in args])
userAdd(*args) #3)
print("End of function")
return
return wrapper


add = modified_add(add) #4)
add(5,6) #5)
```

Lets understand the code above to understand decorators.

1) There is a simple add function defined. IT can accept 2 numbers, add them and produce a output number. Eg: 5 + 6 = 11

2) But I want to modify this behavior by changing the inputs, every input is incremented by 1.5 and then added. For this a new function is created.

- The input to this modified_add is the original function.
- Any arguments passed to the original function can be received as arbitrary arguments inside a nested function (wraps) It can be given any user defined name.
- The argument values are received and updated.

3) It is then converted back to a tuple so that it is used to call in the original function - This nested function is returned as object whenever the decorator is called.

4) The decorator function is called to produce the decorator object.

5) When this runs the nested function in previous step actually executes, giving us the new behavior.

6) This line is entirely optional. This serves the purpose of assigning original functions name and doc string to the wrapper function. check this below showing it correctly.

```
add.__doc__

>>'\n This is add function\n '

add.__name__

>>'add'
```

If the line 6) is removed and executed the names will not be assigned correctly. It will only show the properties of wrapper. It might lead to confusion if anyone checks.

```
add.__name__

>> 'wrapper'

add.__doc__

>> '\n This is a decorator function\n '
```

Another way of creating decorator object:

Lets consider the same code example as above. Instead of calling the decorator function explicitly we can use the @ decorator operator to inform python how the function is being decorated.

on top of the original function issue the command @ followed by the decorator function.

```
import functools



def modified_add(userAdd): #1)
@functools.wraps(userAdd)
def wrapper(*args):
"""
This is a decorator function
"""
print("Start of function")
args = tuple([i+ 1.5 for i in args])
userAdd(*args)
print("End of function")
return
return wrapper



@modified_add #2)
def add(a,b):
"""
This is add function
"""
print(a+b)
return(a+b)



add(5,6) #3)
```

1) The decorator function has to exist before using it.

2) Decorator is issued on top of the original function. This automatically reads below function as input and runs the decorator.

3) The original function can be called directly since the decorator operator is already used in function definition.

This is the common approach followed. Decorators find extensive use in web programming, to process inputs and outputs. It is also used in unit testing libraries.

Multithreading for efficiency

By default a python program execute in a liner fashion, this execution can be considered the main thread that executes. After completing one line or block of lines only it goes to the next one for execution. This provides room for improvement. You can expect computer processor memory to be available and made use of when the main thread executes. We will run a separate operation that is not dependent on current execution in another separate thread called as child thread.

Consider following operations to execute in different threads.

- Get inputs from the user on what crop to sow & area in main thread.
- Monitor, if the water level is less than 5 m in water tank then switch on motor and if greater than 40 m switch off the motor.
- Manually supply inputs for the water level in tank for simplicity.
- This needs to checked continuously and switch on/off.

```
current_water_level = 3

low_level = 5

high_level = 40

motor = False



while True:

crop = input("Enter Crop to sow")

area = input("Enter area to sow in sqft")
```

```
print ("{0} will be sowed in {1} sqft".format(crop,area))



if (current_water_level <= low_level):
motor = True
elif (current_water_level >= high_level):
motor = False
```

The above code achieves the defined steps but not completely. When the user is waiting to enter the inputs, water level is not monitored. This is the reason for executing main thread and child threads. Let us re-organize the code and create a child thread.

```
import threading
import time



current_water_level = 3
low_level = 5
high_level = 40
motor = False



def monitor_water_level(): #1)
global current_water_level
while True:
time.sleep(10)
if (current_water_level <= low_level):
motor = True
print("motor is switched ON\n")
```

```
elif (current_water_level >= high_level):
motor = False
print("motor is switched OFF\n")



t1 = threading.Thread(target=monitor_water_level) #2)
t1.start() #3)



while True: # 4)
crop = input("Enter Crop to sow\n")
area = input("Enter area to sow in sqft\n")
current_water_level = int(input("Enter current water level\n"))
print ("{0} will be sowed in {1} sqft\n".format(crop,area))
```

1) We will split the monitoring of water level code into function monitor_water_level. This will make use of current_water_level the global variable (hence the global keyword is used inside the function) and uses the same logic to ON/OFF. But runs continuously inside a infinite While loop and wait 10 sec between executions.

2) But before the inputs are obtained "threading.Thread" defines thread operation. This requires library threading to be imported in the program. Input to target argument is the function name which has to be executed as the child thread. By now only a thread object is created. Still it hasn't started execution.

3) Thread object dot start command starts the specific child thread execution. Now the execution of function - monitor_water_level starts. Also the program proceeds to the next block of codes. It wouldn't wait in the same line for function to complete and return back since it is a thread execution.

4) In the main thread, there is also a infinite while loop, which repeatedly gets inputs from the user, the crop and area. Since it is only a dummy functionality we have for current_water_level, we are taking that input also from the user for testing. We can observe by changing its values and see how

motor is turned ON/OFF, the corresponding print statement can also be observed in terminal.

Thread Management

To understand further concepts lets consider another example. We will perform some operations as follows.

Write 2 functions that will count individually from 1 to 10 and 1 to 15 separately with a sleep time of 1 sec for every count.

```
import threading #1)
import time



start_time = time.time() #2)
print(start_time)



def print_func(s,lock): #5)
lock.acquire()
print(s)
lock.release()



def func1(n,lock): #3)
for i in range(n):
print_func("func1 " + str(i),lock)
time.sleep(1)



def func2(n,lock): #4)
for i in range(n):
print_func("func2 " + str(i),lock)
```

```
time.sleep(1)



lock = threading.Lock() #6)
t1 = threading.Thread(target=func1,args=(10,lock)) #7)
t2 = threading.Thread(target=func2,args=(15,lock))



t1.start() #8)
t2.start()
t1.join() #9)
t2.join()



print("Exiting main thread") #10)
end_time = time.time()
print(end_time)
print(end_time - start_time) #11)
```

We are going to execute 2 different functions, func1 and func2 and see,

- Time is saved because of multi threading
- Waiting for a thread execution to complete
- Problem encountered in accessing a common shared resource
- Using thread lock object to manage shared resource

1) Import threading and time library

2) Save the start time

3) Write func1 to count from 1 to 10 with a delay of 1 sec. It accepts arguments - count value and the lock object that will be created. We will only receive this object and pass it to another function that we will create in next steps.

4) Write func2 to count from 1 to 15 with a delay of 1 sec. It is similar to func1 only. It also accepts arguments - count value and the lock object that will be created. We will only receive this object and pass it to another function that we will create in next steps.

5) Both func1 and func2 doesn't print the output count value in their functions itself. Because the print statement prints output to console window, this is a common shared resource, when 2 statements execute to print their respective text both gets overlapped. The only way to manage is by separating the common shared resource and having an understanding if the resources is busy or available. This achieved by creating the function print_func. This has 2 arguments - print text (s) and lock object.

6) The lock object is a global one, created in this line. This is already passed as input when calling the print_func

- This print_func is now being called by 2 functions - func1 & func2 at almost same time one after another immediately. Whoever accesses the print_func first takes the lock (lock.acquire). When func1 is in possession of lock only that particular thread can execute remaining statements until releasing the lock (lock.release)
- The lines of code between lock.acquire and lock.release is called critical section. Only one thread can execute the critical section. When func2 thread executes thread.acquire it has to wait until lock becomes available after lock.release is executed by func1 thread.
- Similarly if func2 thread has acquired lock first then func1 should have waited in that case. By using locks the print resources are not overwritten.

7) Two thread objects t1 and t2 are created for func1 and func2 respectively.

8) Thread execution starts with t1.start() & t2.start()

9) After starting the thread executions the main thread proceeds to next lines available in code. The main thread execution will complete if there are no more lines. But the child threads that were started executes independently and completes its execution.

- But here we have issued t1.join command. This will make the program to wait in this line until t1 completes. This is a blocking statement. Only

after t1 completes t2.join executes - if t2 has already completed it will proceed to next line if not waits in this line until t2 completes.

10) This print has no significance, just to show main thread has still not exited.

11) Calculate and print the total time taken from the beginning. This will be roughly 15 second only, even though we had 2 different counters running from 1 to 10 and 1 to 15. Because both were counting at the same time because of multi-threaded operation.

Running Thread using classes

In previous section we executed functions inside a thread. Similar to that we can use a class object inside a thread. Remaining every other concept will be the same.

Lets rewrite the same example as before to count numbers 1 to 10 and 1 to 15 using a class method and run it as two different threads.

```
import threading #1)
import time
class myThread(threading.Thread): #2)
def __init__(self,id,name,counter): #3)
threading.Thread.__init__(self)
self.id = id
self.name = name
self.delay = 1
self.counter = counter
def run(self): #4)
print_func(self.name,.delay,self.counter)



def print_func(name,delay,counter): #5)
for i in range(counter):
lock.acquire()
print(name,i)
lock.release()
time.sleep(delay)



lock = threading.Lock() #6)
```

```
thread1 = myThread(1,"Thread-1",10) #7)
thread2 = myThread(2,"Thread-2",15)



thread1.start() #8)
thread2.start()



thread1.join() #9)
thread2.join()



print("Exiting main thread")
end_time = time.time()
print(end_time)
print(end_time - start_time) #10)
```

Lets break down the above code and understand.

1) Import required threading libraries

2) Create a user defined class, any name can be given and inherit threading.Thread (a system defined class for a Thread)

3) Define constructor for the thread class. Get inputs - an id, name and counter for the thread object that has to be created.

4) Write the run method. run is the default method that is present inside a thread base class. we are overwriting this with the functionality the thread has to implement. This run method will execute when we call the thread object start method further below in code. The run method also calls the common shared resource function - print_func, in a similar fashion what we saw in previous section.

5) Define the print_func and add lock.acquire and lock to release to define the critical section. Inside the loop counter lock is defined only for the print statement.

6) Thread lock is created in outer global scope so that it becomes accessible inside print_func

7) Thread objects are created simply by creating objects for the user defined class. since the class inherits thread base class, the same object will become the thread object.

8) To start execution of the thread, thread1.start() and thread2.start() can be issued respectively.

9) To wait for thread completion, thread1.join() and thread2.join() are issued. These are blocking statements. Waits in the same line until completion.

10) Finally the total time taken for program completion is printed. This result should be almost same as the previous attempt using functions. This offers a more compact usage using classes.

Text & CSV file processing

Any data that is stored in non-volatile memory is stored in file system. These files are of multiple formats and serve multiple purposes. For categorization, we can group files into 3 main categories,

1) Unstructured files

2) Semi-structured files

3) Structured files

Unstructured files

These are files that store data from which it is not efficient to read and write data. It stores data in a continuous format with poor organization. To infer meaning from the stored data is also not easy.

Examples, Word documents, text files, images, audio, video etc. The data can be of any type - text, audio, video etc. And there can be encoding or no encoding as well.

Semi-structured files

A semi-structured file has defines some format and template in which data needs to be stored. This provides opportunity to read and write data a little more efficiently.

Examples, XML, JSON, HTML etc., are all semi-structured file formats. A semi-structured file format can be used for storing application configurations, used to send communication messages over internet to other applications etc.

Structured files

A structured file is a database. A database stores structured details inside cells of a row and column. In every cell field only the expected data can be stored. A single database can contain multiple tables and each table will have rows and columns. More on this we will see in later sections of the book.

This chapter we will cover the basics to read, edit and write contents to a text based file.

File modes

Any text file can be opened in python in 3 modes,

1) read - A file is opened for read only process. When opened in this mode, text contents cannot be written to that file. The file intended to open in this mode, should exist in computer, if that file doesn't exist then FileNotFound Error will be thrown.

2) write - A file is opened for write purpose. When open in this mode contents can written to that file. By default the file starts writing from initial starting position overwriting previous existing values in that place. The file intended to open in this mode,if it doesn't exist will be created in the provided file directory.

3) append - A file is opened for write purpose but starting from the end of current content. It does not overwrite existing text present. The file intended to open in this mode,if it doesn't exist will be created in the provided file directory.

File Operations

In this section lets see how to create a text file and write contents to this file.

Write to Text File: Text files are most commonly used to store data in many places. We can use this to store application log, data extracted from internet, and many other purposes simple to store data permanently.

In following example we can write data collected from console input and writ it into text file.

```
f = open("invoice.txt",mode = 'w') #1)
f.write("{0:<15} {1:<15} {2:<15}\n".format("ITEM","QUANTITY""PRICE")) #2)



while(True): #3)
item = input("Enter Product name: ")
quantity= input("Enter Quantity: ")
price = input("Enter Price: ")
if item!="" and quantity!="" and price!="":
f.write("{0:<15} {1:<15} {2:<15}\n".format(item,quantity,price))
else:
break
f.close() #4)
```

1) Use the open method and pass inputs as file path and mode in which the file has to be opened. The mode can be one of,

- 'w' - for write mode
- 'r' - for read mode
- 'a' - for append mode. This method returns the file handle using which we can read and write files.

2) Use f.write() and pass a string input. This will be written to text file. Here Item, Quantity and Price are written in equal spacing using the string format function.

3) Create a while loop, inside which get inputs item, quantity and price from console and write to text file. If any one of three values is received empty then break the loop.

4) Close the file handle in command f.close()

When write() method is used new line is not automatically inserted in the file. Hence we explicitly add '\n' new line character inside the string.

Append to Text File:

When a file is opened in append mode, further write operation will be written from end of file. If opened in write mode, further write operation will be written from start of the file.

In other words append will retain existing content and add data whenever file is opened for processing. In a write mode whenever file is opened it will start writing from the start of file.

To open in append mode change the mode argument to 'a' in open method.

```
f = open("invoice.txt",mode = 'a') 1)
```

Read from Text File: To read text files we will use the same open method and change mode to 'r'.

```
f = open("invoice.txt",mode = 'r')

items = f.readlines() #1)

for i in items: #2)

print(i.replace('\n',''))

f.close() #3)
```

1) With the created file handle use f.readlines() method to read all lines from the text file. The returned data will be in a list. Each list item will contain a line from the text file. (A line is separated by a new line character '\n')

2) Iterate the list item and print it to console. In each element there will be a new line character and print statement will also print to a new line each time print function is called , hence remove the new line present in the list element using string replace function.

3) Close the file before exiting. Otherwise file might get corrupted if not properly closed.

CSV processing

Write to CSV File:

There are many different file formats as we introduced in the beginning. We will introduce a little more formatted text data and more commonly used for data processing.

A csv file contains text data. Each line shall have a separator element, for example it can be one of , | ; : etc. This kind of text data when opened in office application will display the data inside rows and columns. Every line

will be in a row. For every column the data picked by making use of the separator. At every separator occurrence, data will be show in consecutive cell in the same row.

Lets write and read some data in a CSV file.

```
import csv



with open('invoice.csv','a') as invoicefile: #1)
while (True): #2)
item = input("Enter Product name: ")
quantity = input("Enter Quantity: ")
price = input("Enter Price: ")
invoice = [] #3)
if item != "" and quantity != "" and price != "":
invoice.append([item,quantity,price]) #4)
f = csv.writer(invoicefile, delimiter=',') #5)
f.writerows(invoice) #6)
else:
break
```

1) Import the csv library, Open the csv file using with command, follow the syntax as provided - this is only an alternate method. we could have opened the file as in previous case that is perfectly fine too. The with command provides the file handle invoicefile.

- Open the file in append mode - 'a'
- When a file is opened in with command, there is no need to use the file close method.
- All lines making use of the file handle has to be intended inside the with command. When the intend block exits the handle will be automatically closed.

2) Create a while loop, inside take values from user and write it to csv file. When any one of received string is empty break the while loop.

3) Create empty list invoice in each iteration and add value received from the console.

4) Append item,quantity,price as one single list element into invoice list.

5) Create csv.writer object using the file handle invoicefile. In the writer object specify the column separator character that has to be used in the delimiter argument.

6) With the csv.writer object f and method writerows() write the data present in list into text file. The data contained in invoice - is a list of list values. When this is passed to writerows every list element will be written as separate row. And the nested list items will be joined using the separator and written to csv file.

Read from CSV File: To read from csv file also we will import the csv library and make of reader method. We will read the invoice file that we wrote to in previous step.

```
import csv



print('Item,Quantity,Price\n')
with open('invoice.csv') as invoicefile: #1)
invoices = csv.reader(invoicefile, delimiter=',') #2)
for invoice in invoices: #3)
print(invoice[0], invoice[1],invoice[2])
```

1) open the invoice csv file and get the file handle invoicefile

2) use csv.reader() method, pass the file handle and delimiter as arguments to get the csv reader object.

3) Using the csv reader object iterate in a for loop to get individual row values. print the list element index to the console.

Modules & Packages

We can organize variables, functions and classes into python files as packages and modules. A single python file is nothing but a module. One or more python files when present inside a special directory then it becomes a

package. We can import and make use of the variable, functions or classes present in package or module anywhere as required.

Creating a Module

To make use of a module in a python script by importing, it needs to be available in Module search path, below are the system locations the import statement searches for the mentioned module

1) Built in module (Python Installation path)

2) Directories defined in sys.path

3) Current working directory

4) Python path variable

Lets define a module utility.py and use them in file app.py. Have both the files in same directory and set the cut working directory to file contained directory.

```
# Create contents of this file in utility.py

# Update current working directory to the directory in which this module will exist

# os.chdir('path')



def get_price_details(quantity,perkgcost):

grams = quantity * 1000

price = perkgcost / 1000

return(grams*price)
```

Create another file called app.py. In this file import utility package. Access package name dot function name and call the function. Pass required input values.

```
import utility #1)

price = utility.get_price_details(3, 70) #2)

print(price)
```

1) import the module, if the module is found in one of module search path locations then it will be imported.

2) Use dot operator and access function/class from inside the module. Call the function and store the return result.

Say someone updated module utility in runtime, The new changes will not be recognition unless python interpreter restarts and loads modules a gain. Instead the module can be reloaded as follows without restarting.

```
import imp

imp.reload(utility)

# P.S - Any module should be already loaded to execute a reload operation
```

Importing a class from module:

Create the below class in file marketplace.py and import it into app.py file for execution. This illustrates usage of class files in a module.

```
# Have the below code in file marketplace.py



class Marketplace():

def __init__(self, quantity, product, perkgcost):

self.quantity = quantity

self.product = product

self.perkgcost = perkgcost

return



def get_price_details(self):

grams = Marketplace.kgtograms(self.quantity)

price = Marketplace.pergramcost(self.perkgcost)

return(grams*price)



@staticmethod

def kgtograms(kg):
```

```
return (kg*1000)



@staticmethod
def pergramcost(cost):
return (cost/1000)
```

Inside app.py file, import the required class, create an object and execute its methods.

```
from marketplace import Marketplace



obj = Marketplace(3.5,"carrots",80)
print(obj.get_price_details())
print(obj.kgtograms(3))
>>280.0
>>3000
```

Creating a Package

One or more modules together inside a special directory becomes a package. This package can be imported into other python script files for execution. Not any directory becomes a package, only if the directory contains init.py file then it becomes a directory. This file can be defined empty or it can contain other imports and class definitions.

The steps to create a package are,

1) Create a directory, name it suitably

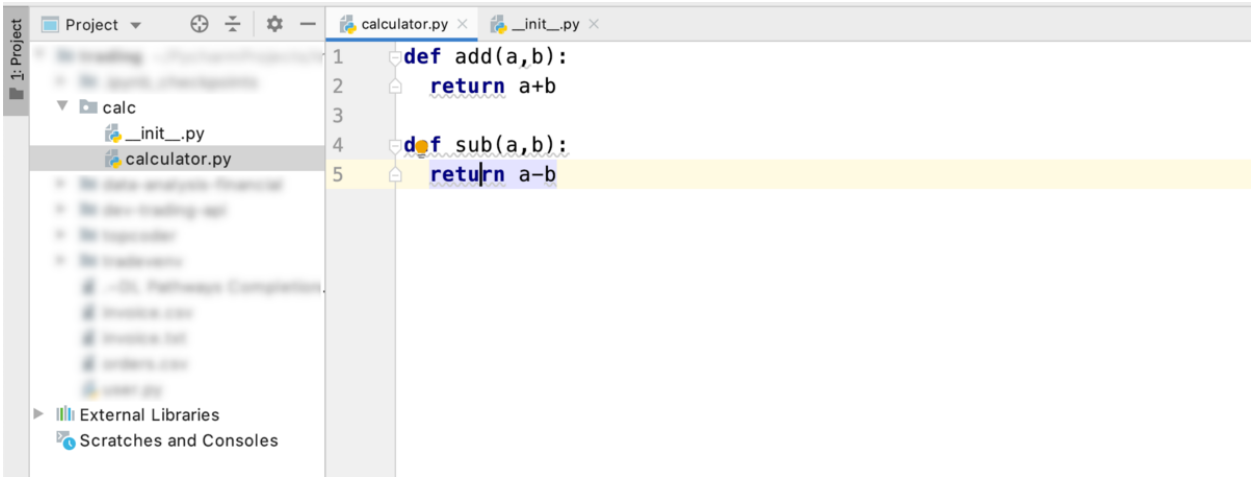
2) Create *init*.py file

3) Create module files inside the directory. Define class, functions or variables as required.

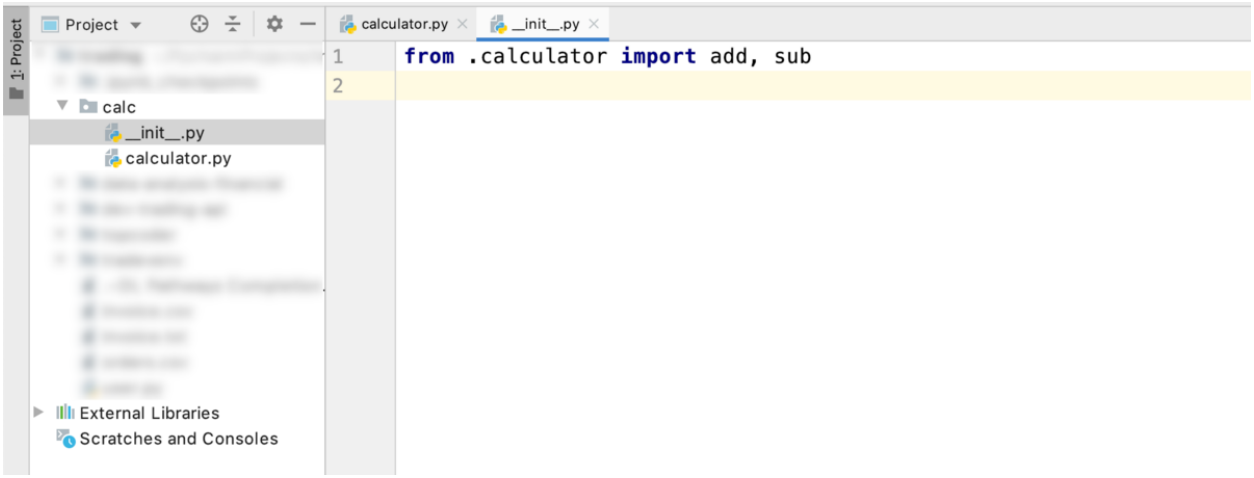
4) Additional optional step, if you want to avoid using the module name, then import class names inside init file, then this will become part of the package namespace and you will be allowed to import the class directly in other files only using the package name.

Create a directory named calc.

Add a module named calculator.py with functions defined in it for add & sub.



In the init file import the functions from its module. Refer to the screenshots for the setup.



In other python script for example app.py execute the following

```
from calc import add,sub



print(add(4,5))

print(sub(4,5))
```

```
>>9

>>-1
```

Error handling using Exceptions

An error occurs when a particular program statement is not understood by the python interpreter. The program exits when there is an error. This could have been the result of a syntax error or semantic error. It is responsibility of the programmer to be aware of error scenario and implement the program accordingly. Only few of the errors for which programmer is responsible can be handled correctly. In most scenarios it can occur in run time and varied scenarios, we should not allow the interpreter to quit abruptly. Instead these errors need to be elegantly handled using an except block which we will define further.

Inside the except block either we shall retry the same operation, create log of the error that has occurred, or simply continue executing further operations to be performed. The programmer only need to ensure the behavior shouldn't be annoying to the end user whatever may be the case. There has to be proper notification and guidance to the end user.

System defined exceptions

The error or exception that are produced in a program can be of 2 types,

1) Built in exceptions,

2) User defined exceptions.

1. Built in exceptions: Under built-in exceptions there are more than ~35+ exception categories. Some of the most common exception types we might have encountered in the previous chapters.

To see the list of exception categories refer to the documentation on python help manual, https://docs.python.org/3/library/exceptions.html

To handle any exception we will be using three keywords in combination, try, except and finally

Consider following example, where we will produce few exceptions and handle them.

```
import sys #1)
```

```
value = 0 #2)
try: #3)
result = 1/int(value)
print(result)
except: #4)
print(sys.exc_info())
finally: #5)
print("End of try except block")
#6)
print("The input is: ",value," and the result is: ",result)
```

1) Import sys library, we will be using this later

2) Store the input in variable value. Assign value to 0.

3) Create the try block, all python statements has to be intended inside the try block. And there should be a colon at the end of try keyword.

- Inside the try block define operations that need to be monitored for exceptions, if there arises any errors/exceptions then the code will enter into the except block that is defined below the try block.
- In case of an exception in a particular statement the try block will not execute further, it will directly enter into except block.

4) Define except block using the keyword and a colon. Indent all code inside this block. Define operations that needs to performed incase of an exception. It can be any operation up to the developer, can be a retry attempt that failed, a logging operation or print the error and simply continue further execution.

- Retrieve the exception message using sys.exc_info() method

5) Define the finally segment using keyword followed by colon. All code inside finally has to be indented. Any operation defined inside finally will be executed irrespective of whether exception occurred or not.

- Also defining the finally block is optional. This can be skipped as well. In presence of a try block, only an except block is mandatory.

6) print the input and the result. This statement will be executed if there was no exception itself, or if the exception occurred and it is handled.

Handling multiple exceptions: An except block shall execute for any exception or for a specific category of exception as well. For this we need to define multiple exception blocks along with conditions. Define 3 except blocks as below and lets understand what will happen in this case.

```
import sys



value = [1]



try:
result = 1/int(value[2])
print(result)
except IndexError: #1)
print(sys.exc_info())
except ZeroDivisonError: #2)
print(sys.exc_info())
except: #3)
print(sys.exc_info())
finally:
print("End of try except block")



print("The input is: ",value," and the result is: ",result)
```

1) If the input value will be a List data type and an attempt to access non existent index will throw IndexError. Only when the exception is IndexError this except block will be executed.

- No other except block will be executed.

2) If the input value will be a integer 0, then in this ZeroDevisionError will be raised and this except block only executes.

3) If the input value was a string "Hello" or any other alphabetical characters then the will be a TypeError, in this case there is no exception category defined, hence it will enter into the generic exception block if defined like this in 3rd point.

- If this generic exception is not defined then it will lead to throwing the exception to the user and the program will stop execution immediately.

User defined exceptions

All the exceptions so far we saw were inbuilt system defined exceptions, we can also define our own exceptions. This can be achieved in 2 steps.

1) First create a exception by inheriting the Exception class.

2) Throw exception on the required condition using the raise keyword.

Consider example problem, where if an individuals age is less than 18 then VotingAgeError should be raised as a exception.

```
class VotingAgeError(Exception): #1)
pass



age = 17
voting_age = 18



try: #2)
if age < voting_age:
raise VotingAgeError
else:
print("Voting age is met")



except VotingAgeError: #3)
```

```
print("Voting Age is not met")

print(sys.exc_info())
```

1) Create user defined class VotingAgeError as a exception by inheriting the Exception class

2) Write a try block, check inside if a individual age is less than 18 and then use the keyword raise to throw user defined exception VotingAgeError

3) Write a except block for VotingAgeError to catch if this exception occurs and print the message to console.

assert

Assert is used to check if a python expression is True or False, and raise an AssertionError if the expression was False and print a optional message to the console.

Consider a simple statement below and observe the AssertionError that occurs. If the age integer was greater than 18 then there will be no error.

```
age = 17

assert age > 18,"age is less than voting age"



>>Traceback (most recent call last):

>> File "<input>", line 2, in <module>

>>AssertionError: age is less than voting age
```

Database operations for storing data

Database is a structured data, that is used to store our data in the form of rows and columns.

- The data is stored in continuous memory inside database file, because of which read, write operations will be faster and efficient.
- Special software is required to create, read, write and communicate to database files.
- We also require a special command line language to communicate with the database file, Structure query language (SQL). This applies to all the relational database system. (The scope in this section is limited to

relational database only. There is also another approach to building database, non-relational database. An example is Mongo DB.)

There are many DBMS (Database Management Software) software providers. Few of them to mention are Microsoft SQL server, PostgreSQL, MySQL, Sqlite, Oracle, etc,.

We will use Sqlite because that is easy to install and understand if you are a beginner in DBMS. There will be little difference when using other database and increased features which should be easy to adapt using.

Sqlite is a portable software and it is less secure. We can interact using a graphical application as well to understand the SQL commands, using application DB Browser. This can be installed from here. https://sqlitebrowser.org/dl/

We will use the python sqlite3 library to interact with database. Before continuing further we will understand some of the basic SQL commands.

CRUD operations

SQL commands can be categorized into 4 different operations - Create, Read, Update and Delete. This can be termed as CRUD operations.

SQL commands

CREATE: This command is used to create a table inside database. While defining the table we also have to define what are the columns that will be present along with the data type. This is also termed as the schema of the table.

READ: This command is used to read if there are any data present in the table already. This retrieves rows of information present.

UPDATE: This command is used to edit cells present in the table. Either a single cell in a row or for a group of rows.

DELETE: Delete operation is used to delete rows of information. Again either a single row can be deleted or a group of rows.

We will see the above commands and its usage from a python program.

SQL in Python

For communicating with any database there will be a special library required. To execute SQL command in a Sqlite database we will require sqlite3 library.

Below is a program to create a table - EMPLOYEE inside database employee.db and perform the basic operations in defining the schema, adding some records into the table, modifying the data and delete records if required.

```
import sqlite3 #1)



conn = sqlite3.connect('employee.db') #2)



cur = conn.cursor() #3)



#4)
sql = """CREATE TABLE EMPLOYEE (
FIRST_NAME CHAR(20) ,
LAST_NAME CHAR(20),
AGE INT,
SEX CHAR(1),
INCOME FLOAT )"""



cur.execute(sql) #5)
conn.commit() #6)
conn.close() #7)
```

1) import sqlite3 library

2) use sqlite3.connect method to connect to a specific database file. This file if doesn't exist in the provided path, it will create a new file, and open a connection to it. The returned object is called a connection object.

3) using the connection object create a cursor which will act as a pointer inside the database file. We will use this cursor for further operations.

4) create a SQL command for defining the schema for creating the table EMPLOYEE. The command is CREATE TABLE TABLE_NAME(column_name data_type); We did not see SQL commands in detail, but this is pretty much a basic SQL command to create a table inside database. Encourage to read more on SQL and executing it directly in DB browser application to understand them better.

- There are 5 columns defined here along with its data type.
- This SQL command is stored as a multiline string within triple quotes.

5) To execute sql command use cur.execute(sql) providing the sql string as input created in previous step. This performs the required operation on database file.

6) If there are modifications done to database then it needs to be saved. Use the conn.commit() - commit method to save the changes.

7) Finally the database connection needs to be closed. Use the conn.close() method. The above steps will remain the same except changing the sql command that needs to executed in rest of operations below.

Lets add some data to table by defining the corresponding INSERT SQL command for same. The syntax is INSERT INTO TABLE_NAME(column_names) VALUES (values); Refer to the exact command used in below code to understand better.

```
import sqlite3

conn = sqlite3.connect('employee.db')

cur = conn.cursor()



#1)

sql = """INSERT INTO EMPLOYEE(FIRST_NAME,

LAST_NAME, AGE, SEX, INCOME)

VALUES ('Geetha', 'Lakshmi', 22 , 'F', 40000),

('Ramya', 'Raghu', 23, 'F', 45000),

('Renu', 'Lakshmi', 22, 'F', 50000),
```

```
('Shan', 'Kumar', 21, 'M', 30000),
('Shyam', 'Sundar', 22, 'M', 35000),
('Shan', 'Sam', 23, 'M', 60000);
"""



#2)
cur.execute(sql)
conn.commit()
conn.close()
```

1) Define the SQL command to insert few rows of information in respective table an columns.

2) Issue the execute command, save and close the connection. It is not required to close and reopen the connection each time. We can perform all the operations once the database is open and finally close them as well. Just to keep the examples independent it is kept this way in above code.

Now, lets read data that is available in EMPLOYEE table. The syntax to read data is SELECT column_names FROM TABLE_NAME. Refer to below command in the code.

```
import sqlite3
conn = sqlite3.connect('employee.db')
cur = conn.cursor()



#1)
sql = """SELECT * FROM EMPLOYEE"""
cur.execute(sql)
#2)
result = cur.fetchall()
```

```
for r in result:
print(r)



#3)
sql = "SELECT * FROM EMPLOYEE \
WHERE INCOME > '%d'" % (40000)
try:
#4)
cur.execute(sql)
results = cur.fetchall()
for row in results:
fname = row[0]
lname = row[1]
age = row[2]
sex = row[3]
income = row[4]
print ("fname = %s,lname = %s,age = %d,sex = %s,income = %d" % \
(fname, lname, age, sex, income ))
except:
print ("Error: unable to fetch data")
```

1) Create the SQL command - SELECT * FROM EMPLOYEE and execute it.

2) This is a read operation and it is expected to return values back. The returned result will be in a temporary buffer like space, which needs to fetched using the command - cur.fetchall(), The returned value will be a list of tuples. Every row data will be a tuple that is present inside the list. Which can be iterated and printed to the console using a for loop.

3) We can also filter data when we read data from the table. The SQL command has certain clauses for same. Use the WHERE keyword to define filter conditions. SELECT * FROM EMPLOYEE WHERE INCOME > 10000 After the table name use the WHERE keyword and define a relational logic that applies to the column name(s) present in the table. Now only those rows that match the criteria will be returned.

4) Execute the sql command, use the fetchall() method to retrieve result. Also iterate and format the returned result before printing it to console.

To update data present in a table, use the following syntax UPDATE TABLE_NAME SET COLUMN_NAME=value WHERE COLUMN_NAME=condition. Refer to the code below for better understanding. Here the Age is increased by 2 for a particular FIRST_NAME

```
import sqlite3

conn = sqlite3.connect('employee.db')

cur = conn.cursor()



#1)

sql = "UPDATE EMPLOYEE SET AGE = AGE + 2 WHERE FIRST_NAME = '%s'" % ('SHYAM')



try:

#2)

cur.execute(sql)

conn.commit()

except:

#3)

conn.rollback()

conn.close()
```

1) Use the UPDATE command as in step 1 above. We have used the WHERE filter here to select the rows in which value needs to be updated.

2) Execute the sql command and issue the commit() method to save.

3) If you observe we have performed the sql execution steps inside a try block. If the update command fails and performs a partial update in database tables. Then it is not a desired behavior. The commit has also not occurred because of some exception in previous step, So we might need to undo the last unsuccessful operation.

- Issue the conn.rollback() method to undo the last execute() operation. This will bring the database to last known saved state.

To delete rows of information from database table use the syntax DELETE FROM TABLE_NAME WHERE column_name=condition. Lets delete all rows with age greater than 24.

```
import sqlite3

conn = sqlite3.connect('employee.db')

cur = conn.cursor()

#1)

sql = "DELETE FROM EMPLOYEE1 WHERE AGE >= '%d'" % (24)

try:

#2)

cur.execute(sql)

conn.commit()

except:

conn.rollback()



conn.close()
```

1) Use the command as defined in sql string to define the DELETE operation. Here also we use the WHERE filter to select the required rows that needs to be deleted.

2) Perform the execute operation, commit and close the connection finally.

We have successfully performed the basic CRUD operations on employee.db database file from python program. Explore more on SQL commands and it

should be simpler to use them as well in python program in similar fashion.

Presenting your application - GUI

Any application that is built requires a user and it is responsibility of developer to present it to the user. In the context of application that are built for desktop it can be built such that they are executed from either a terminal window or a Graphical User Interface.

It is best decided based on the audience who will use the application. To provide a user friendly graphical application we will use the tkinter library in python in this chapter. There are other libraries using which also we can build a graphical user interface. tkinter is available part of the python core package.

Widgets

widgets are elements that a user interacts with, from the graphical user interface. Certain widgets are used to provide input to the application and some provide display output back to the users.

tkinter has a defined set of widgets. We will look at some of the important widgets. Keys things to note in the UI construction are, tkinter root window, frame and individual widget.

Root window is mandatory and that contains all other elements.

A frame is optional, its use is to create sub sections inside a root window to group elements together. Directly inside root window or if a frame is present inside that widgets have to created and placed.

A widget can be a class object of Button, Checkbox, Listbox, Radiobutton, Label, Text, Entry etc. Now these widgets are to be aligned inside a window in a neat and presentable fashion. There are two ways to achieve this and referred as geometry positioning of widgets. 1.) Pack Method and 2.) Grid Method.

Pack

In this section lets see how to create some of the widgets and place them inside a root window using pack method. A widget will appear in the root UI window only if any of the geometry methods are used and positioned, otherwise only an empty root window will appear.

Lets see how to create and interact with following widgets Button, Label,Checkbutton, Entry, Text with help of below code.

```
import tkinter as tk #1)
from tkinter import messagebox


t = tk.Tk() #2)


def event(): #13)
messagebox.showinfo("Hi","Welcome")


def getcheckbox(): #14)
print(cb1.get())


def getentrybox(): #15)
print(e1.get())


def clearentrybox(): #16)
e1.delete(0, tk.END)


b=tk.Button(t,text="MessageBox",command=event) #3)
b.pack(anchor=tk.W,side = tk.LEFT) #4)


tk.Label(t,
text="PRODUCT NAME",
justify = tk.LEFT,
```

```
padx = 10).pack(anchor=tk.W,side = tk.LEFT) #5)



cb1 = tk.BooleanVar() #6)
c1=tk.Checkbutton(t,text="one",variable=cb1) #7)
c1.pack(anchor=tk.W,side = tk.LEFT)


cButton=tk.Button(t,text="Check Box",command=getcheckbox) #8)
cButton.pack(anchor=tk.W,side = tk.LEFT)


ecButton=tk.Button(t,text="Clear Entry",command=clearentrybox)
ecButton.pack(anchor=tk.W,side = tk.LEFT)


e1 = tk.Entry(t) #9)
e1.insert(0,'Product Name')
e1.pack(anchor=tk.W,side = tk.LEFT)


eButton=tk.Button(t,text="Entry Box",command=getentrybox) #10)
eButton.pack(anchor=tk.W,side = tk.LEFT)


t = tk.Text(t,height=20,width=30) #11)
t.insert(tk.END,"This is a text box")
t.pack(anchor=tk.W,side = tk.LEFT)
```

```
t.mainloop() #12)
```

1) Import required tkinter libraries

2) Create the Tk() class object which will act as the UI reference object for all other widgets

3) Button is a clickable element to perform a action. Create a Button object by providing following inputs to button as t UI reference, text="MessageBox" button display text, command=event the event function to be called upon clicking the button

4) pack() the button, this will add the button to UI. pack method has default configurations set to anchor=CENTER, and side = TOP. We have updated it to appear a little different, anchor is for widget alignment & side is for the way to add widgets to the UI.

- Different anchor configurations are,
- tk.W - left
- tk.E - right
- tk.N - top
- tk.S - bottom
- Different side configurations are
- tk.TOP - Adds one below the other
- tk.LEFT - Adds one after another rightwards
- tk.BOTTOM - Adds one above another
- tk.RIGTHT - Adds one after another leftwards

5) Label is a display text only. Add label with text - to display, justify - to align the label within the available space,padx - surrounding free space to provide. Also pack the button same as previous widget.

6) Creating a tkinter BooleanVar() cb1 object for storing the value of Checkbox.

7) Create a Checkbutton with display text as 'one' and mapping to variable cb1. And pack the checkbutton also

8) And a Button with display text as 'Check Box' and command to execute getcheckbox which will retrieve checkbutton value and print to console window. And one more Button with display text as 'Clear Entry' and command to execute clearentrybox which will clear the text present in entry box being cleared.

9) Add an Entry widget, insert default value as 'Product Name' and pack the widget as well.

10) Add another Button with display text as 'Entry Box' and command to execute as 'getentrybox' which will fetch the value present in Entry widget. Also pack the widget.

11) Finally add a Text widget defining width and height, insert default text value that shall display inside the widget and pack into the UI.

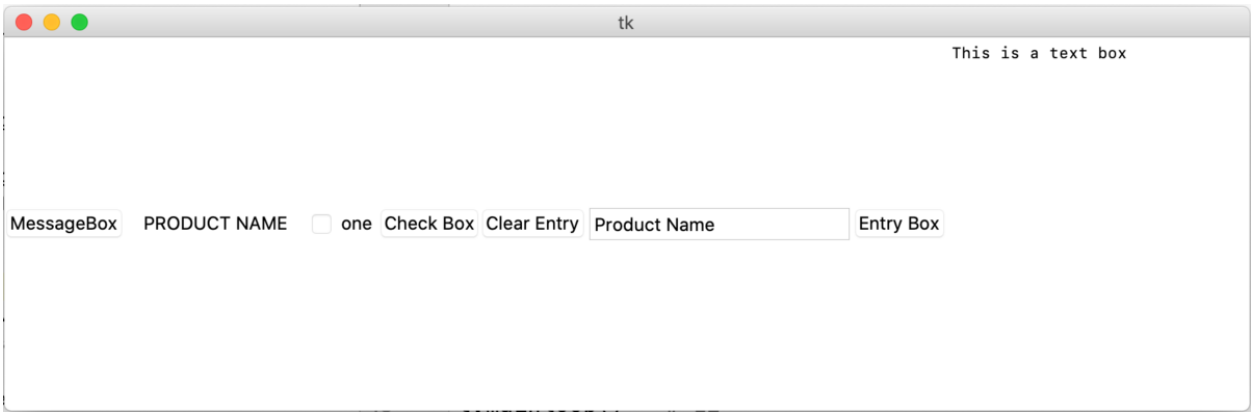
12) Start the mainloop, so that the UI will remain active until it is closed.

13) event function will execute when the 'Message Box' button is clicked. This function will display a message box with welcome message.

14) getcheckbox function will execute when 'Check Box' button is clicked. This will print the value of checkbutton as stored in cb1 into the console window.

15) 'getentrybox' function will execute when the 'Entry Box' button is clicked. This will print the value present in Entry widget into the console.

16) 'clearentrybox' function will execute when the 'Clear Entry' button is clicked. This will clear the value present in Entry widget.



Grid

We will build a similar UI as in previous section, but using a different strategy to place the widgets on UI. By applying grid system to place

widgets. The inputs to grid() method is (row,column) - row number and column number. This is imagined like a matrix, starts with row and column index as 0 starting from top left of a UI window.

Lets create - Label, Entry, Checkbox, Button, Text. This UI is to illustrate the working of few of the widgets. We will define a meaningful application in next topic.

```
import tkinter as tk #1)
from tkinter import messagebox



t = tk.Tk() #2)



def event(): #12)
messagebox.showinfo("Hi","Welcome")



def getcheckbox(): #13)
print(cb1.get())



def getentrybox(): #14)
print(e1.get())



def clearentrybox(): #15)
e1.delete(0, tk.END)



b=tk.Button(t,text="MessageBox",command=event) #3)
b.grid(row=0,column=0,sticky=tk.W)
```

```
# 4)
tk.Label(t,
text="PRODUCT NAME",
justify = tk.LEFT,
padx = 10).grid(row=0,column=1,sticky=tk.W)


cButton=tk.Button(t,text="Check Box",command=getcheckbox) #5)
cButton.grid(row=1,column=0,sticky=tk.W)


cb1 = tk.BooleanVar() #6)
c1=tk.Checkbutton(t,text="one",variable=cb1)
c1.grid(row=1,column=1,sticky=tk.W)


eButton=tk.Button(t,text="Entry Box",command=getentrybox) #7)
eButton.grid(row=2,column=0,sticky=tk.W)


e1 = tk.Entry(t) #8)
e1.insert(0,'Product Name')
e1.grid(row=2,column=1,sticky=tk.W)


ecButton=tk.Button(t,text="Clear Entry",command=clearentrybox) #9)
ecButton.grid(row=3,column=1,sticky=tk.W)
```

```
t = tk.Text(t,height=20,width=40) #10)
t.insert(tk.END,"This is a text box")
t.grid(row=4,column=0,columnspan2,sticky=tk.W)


t.mainloop() #11)
```

1) Import tkinter library as tk, and messagebox from tkinter.

2) Create the Tk() root window t, this element will be used as reference while creating the widgets.

3) Create a Button b with text as "MessageBox" and give command function as event which should execute upon clicking this button. Place the button using the grid method. row as 0 and column as 0. sticky is a argument that says how the button should be orient inside the cell. It can be one of tk.N,tk.S,tk.E,tk.W,tk.NE,tk.SE,tk.SW,tk.NW directions.

4) Create a Label with text "PRODUCT NAME", justify text with argument value tk.LEFT, and place the widget using grid method. row as 0 and column as 1.

5) Create a Button cButton and upon clicking that call getcheckbox function to read Checkbutton value. Place this button using the grid method. row as 1 and column as 0.

6) Create a tkinter Boolean variable cb1 and give this as input to c1 Checkbutton. Place this c1 using the grid method, row as 1 and columns as 1.

7) Create a Button eButton and upon clicking that call getentrybox function to read Entry box value. Place this button using the grid method. row as 2 and column as 0.

8) Create Entry box e1 and insert default value as 'Product Name'. Place this Entry box using the grid method, row as 2 and column as 1.

9) Create a Button ecButton and upon clicking that call clearentrybox function to clear contents of Entry box. Place this button using the grid method. row as 3 and column as 1.

10) Create a Text box t, insert some text value - 'This is a text box'. Place this Text box using the grid method, row as 4 and column as 0 and also columnspan as 2. columnspan defines how many cells this particular widget should cover along the column.

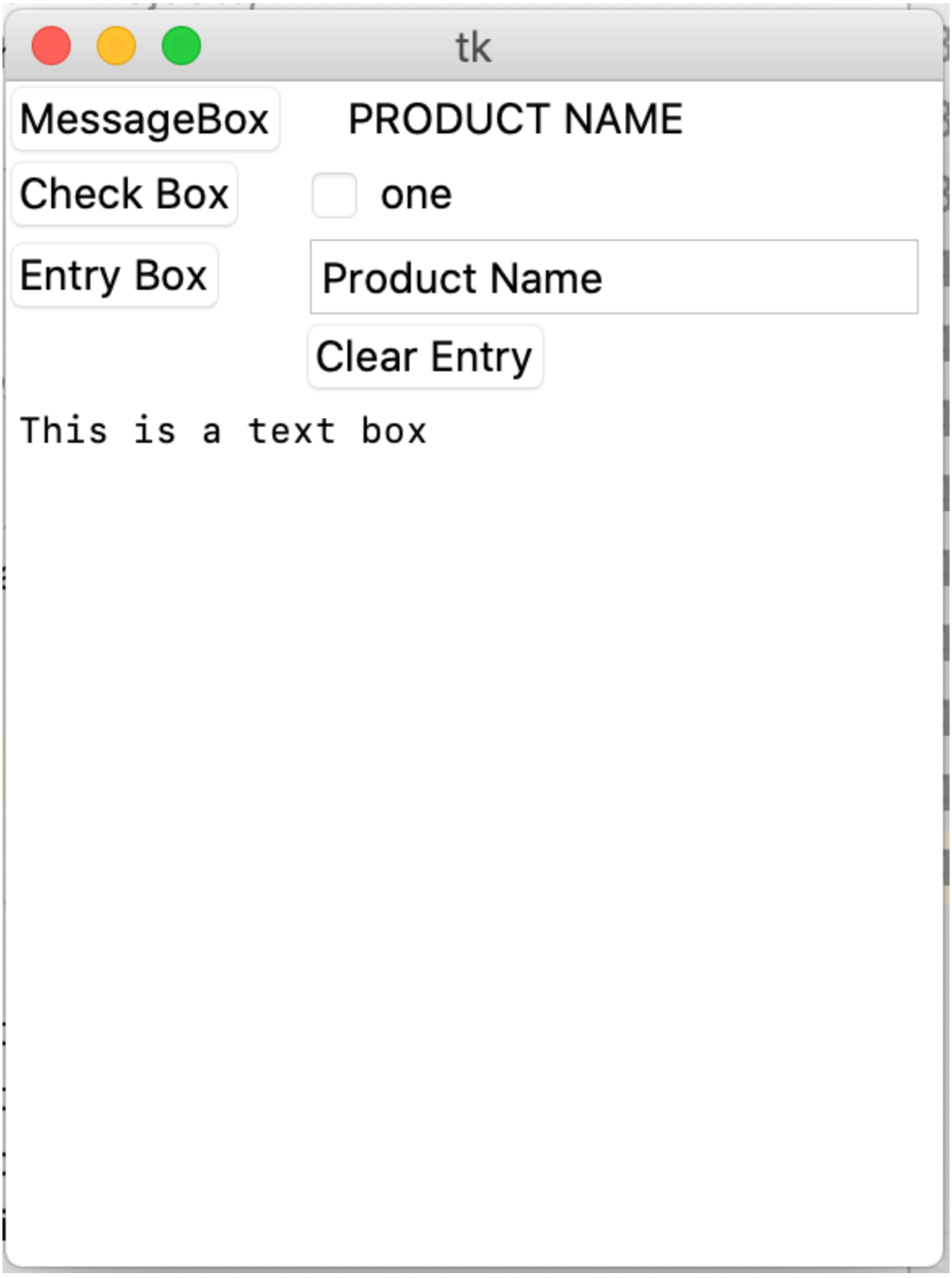
11) All contents between Tk() and Tk().mainloop() is kept active and running.

12) event function will execute when the 'Message Box' button is clicked. This function will display a message box with welcome message.

13) getcheckbox function will execute when 'Check Box' button is clicked. This will print the value of checkbutton as stored in cb1 into the console window.

14) 'getentrybox' function will execute when the 'Entry Box' button is clicked. This will print the value present in Entry widget into the console.

15) 'clearentrybox' function will execute when the 'Clear Entry' button is clicked. This will clear the value present in Entry widget.



Create a working UI and functions (Marketplace App)

Create a simple application, where inputs like PRODUCT NAME, QUANTITY, PRICE and Whether a BUY or SELL offer is being placed are given. All these records are displayed in a text box and the same is stored in a database table as well.

Lets define database, UI and functionality implementation.

First we will create required database and table using a separate script (something like a setup script as below). Otherwise every time when the application file is run attempt to create table will get executed.

```
#1)
import sqlite3
conn = sqlite3.connect('pythondb.db')
cur = conn.cursor()



#2)
sql = """CREATE TABLE APP(
TXNO INTEGER PRIMARY KEY AUTOINCREMENT,
PRODUCT_NAME CHAR(20),
PRODUCT_QTY FLOAT,
PRODUCT_PRICE FLOAT,
BUY_SELL INTEGER)"""



cur.execute(sql)
conn.commit()
```

1) Import sqlite3 library, connect to a database file - pythondb.db, if this does not exist it will get created.Create a cursor object.

2) Create a sql command for table creation as a string value. Create columns, TXNO, PRODUCT_NAME, PRODUCT_QTY, PRODUCT_PRICE & BUY_SELL with relevant data type. All these values will be from the UI.

The intention is to store all the transactions that can take place through the UI application.

Next define the UI and its functional implementation.

```
#1)

import tkinter as tk

from tkinter import messagebox

import sqlite3



#2)

conn = sqlite3.connect('pythondb.db')

cur = conn.cursor()

root = tk.Tk()

root.title("Farm Marketplace app")

root.geometry("600x400")



def submit(): #13)

product_name = e_product_name.get()

product_qty = e_product_qty.get()

product_price = e_product_price.get()

buy_sell_tuple = getbuysell()

buy_sell_print = ""

buy_sell = 0

if buy_sell_tuple[0] == True:

buy_sell = 1

buy_sell_print = "BUY"

elif buy_sell_tuple[1] == True:

buy_sell = 2
```

```
buy_sell_print = "SELL"


db_insert = (product_name, float(product_qty), float(product_price), buy_sell)


sql = """INSERT INTO APP(PRODUCT_NAME,PRODUCT_QTY,PRODUCT_PRICE,BUY_SELL)
VALUES ('%s','%f','%f%d')""" % db_insert


cur.execute(sql)
conn.commit()


sql = """SELECT * FROM APP"""
cur.execute(sql)
transactions = cur.fetchall()
for transaction in transactions:
t = [str(i) for i in transaction]
log = log + ','.join(t) + '\n'
txt_display.delete(0.0,tk.END)
txt_display.insert(0.0,log)


def getbuysell(): #14)
return (cb_bv_buy.get(),cb_bv_sell.get())


def enablebuyonly(): #15)
if cb_bv_buy.get():
```

```
cb_sell.configure(state='disabled')
if not cb_bv_buy.get():
cb_sell.configure(state='normal')


def enablesellonly(): #16)
if cb_bv_sell.get():
cb_buy.configure(state='disabled')
if not cb_bv_sell.get():
cb_buy.configure(state='normal')


#3)
l_product_name = tk.Label(root,
text="PRODUCT NAME",
justify = tk.LEFT,
padx = 10)
l_product_name.grid(row=0,column=0,sticky=tk.W)


#4)
l_product_qty = tk.Label(root,
text="QTY",
justify = tk.LEFT,
padx = 10)
l_product_qty.grid(row=0,column=1,sticky=tk.W)


#5)
```

```
l_product_price = tk.Label(root,
text="PRICE",
justify = tk.LEFT,
padx = 10)
l_product_price.grid(row=0,column=2,sticky=tk.W)


#6)
e_product_name = tk.Entry(root)
e_product_name.insert(0,'Product Name')
e_product_name.grid(row=1,column=0,sticky=tk.W)


#7)
e_product_qty = tk.Entry(root)
e_product_qty.insert(0,'0')
e_product_qty.grid(row=1,column=1,sticky=tk.W)


#8)
e_product_price = tk.Entry(root)
e_product_price.insert(0,'0')
e_product_price.grid(row=1,column=2,sticky=tk.W)


#9)
cb_bv_buy = tk.BooleanVar()
cb_bv_sell = tk.BooleanVar()
cb_buy = tk.Checkbutton(root,text="BUY",variable=cb_bv_buy,command=enablebuyonly)
```

```
cb_buy.grid(row=2,column=0,sticky=tk.W)
cb_sell = tk.Checkbutton(root,text="SELL",variable=cb_bv_sell,command=enablesellonly)
cb_sell.grid(row=2,column=1,sticky=tk.W)



#10)
b_submit = tk.Button(root,text="Submit",command=submit)
b_submit.grid(row=3,column=0,sticky=tk.W)



#11)
txt_display = tk.Text(root,height=10,width=60)
txt_display.grid(row=4,column=0,columnspan3,sticky=tk.W)
#12)
root.mainloop()
```

1) Import the required libraries

2) Create the database connection to the file that we created in previous step. - Create the cursor object - Create the UI root object from Tk() - Set root window title to "Farm Marketplace app" - Set dimensions of root window as "600x400"

3) Create a label to display "PRODUCT NAME" in grid location row = 0 and column = 0.

4) Create a label to display "PRODUCT QTY" in grid location row = 0 and column = 1.

5) Create a label to display "PRODUCT PRICE" in grid location row = 0 and column = 2.

6) Create a entry box to accept user input for "PRODUCT NAME". Set a default value with insert command. Place the widget in location row = 1 and column = 0.

7) Create a entry box to accept user input for "PRODUCT QTY". Set a default value with insert command. Place the widget in location row = 1 and

column = 1.

8) Create a entry box to accept user input for "PRODUCT PRICE". Set a default value with insert command. Place the widget in location row = 1 and column = 2.

9) Create separate tkinter boolean variables to store buy and sell decision.

- Create Checkbutton to display Buy, map boolean variable for buy decision. Place the widget in row = 2, column = 0.
- Create Checkbutton to display Sell, map boolean variable for sell decision. Place the widget in row = 2, column = 1.

10) Create a button for submitting the values, Execute submit command on clicking the button. Place the widget in row = 3, column = 0.

11) Create a text box, define height and width. Place the widget in row = 4, column = 0.

12) Execute root.mainloop() method to keep the UI active.

13) On clicking the submit button, fetch all widget state and store into variables - product_name as string, product_qty as string, product_price as string, buy_sell_tuple as a tuple storing 2 Check box states True/False.

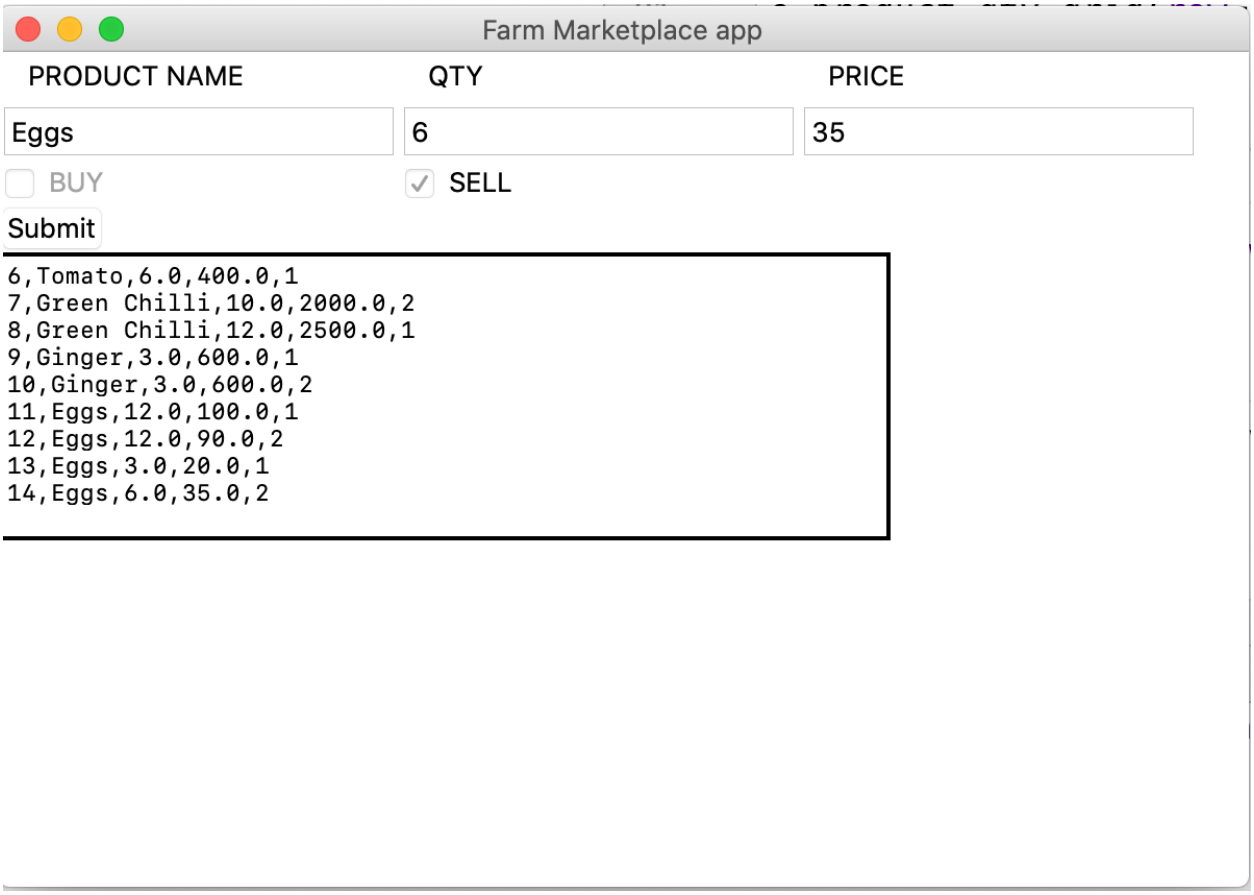
- Convert these into respective data type that is defined in database schema.
- product_name as string
- product_qty as float
- product_price as float
- buy_sell as integer. If Buy boolean is True then set integer value as 1, If the Sell boolean is True the set integer value as 2.
- Also at the same time only either of the boolean of Buy or Sell can be set to True.
- Create a tuple of all above values required for table in db_insert variable.
- Build a sql insert command as a string value.
- Execute the sql command and save the database with commit() command

- Now retrieve the data that is present in table using - "SELECT * FROM APP" query, fetchall results to store it as a string in log variable.
- Delete the existing content of text widget and insert the latest value that is stored in log variable.

14) Define getbuysell method. This method will return the boolean states of variable cb_bv_buy and cb_bv_sell (Boolean states on Buy and Sell Checkbutton). This is used in submit method.

15) Define enablebuyonly method. This is defined as a command method when the Buy Checkbutton is clicked. When the Buy button is clicked disable the Sell Checkbutton, so that only one boolean variable is set any point of time. When this Checkbutton is disabled then the other Sell Checkbutton will get enabled.

16) Define enablesellonly method. This is defined as a command method when the Sell Checkbutton is clicked. When the Sell button is clicked disable the Buy Checkbutton, so that only one boolean variable is set any point of time. When this Checkbutton is disabled then the other Buy Checkbutton will get enabled.



Calculator application (UI & functional implementation)

In this section lets look at another sample application and its explanation. We will see how a graphical calculator is implemented using tkinter library.

This implementation can be broadly seen in two parts

1) UI Implementation

2) Functionality implementation

```
from tkinter import * #1)



expression = ""



def press(num): #3)

global expression
```

```
expression = expression + str(num)
equation.set(expression)



def equalpress(): #4)
try:
global expression
total = str(eval(expression))



equation.set(total)
expression = total
except:



equation.set(" error ")
expression = ""



def clear(): #5)
global expression
expression = ""
equation.set("")




if __name__ == "__main__": # 2)
gui = Tk()
gui.configure(background="green")
```

```
gui.title("GUI Calculator")
gui.geometry("465x325")
equation = StringVar()
expression_field = Entry(gui, textvariable=equation)
expression_field.grid(columnspan=4,row0, column=0, ipadx=70)
equation.set('enter your expression')


button1 = Button(gui, text=' 1 ', fg='black', bg='red',
command=lambda: press(1), height=, width=7)
button1.grid(row=2, column0)


button2 = Button(gui, text=' 2 ', fg='black', bg='red',
command=lambda: press(2), height=1, width=7)
button2.grid(row=2, column1)


button3 = Button(gui, text=' 3 ', fg='black', bg='red',
command=lambda: press(3), height=1, width=7)
button3.grid(row=2, column)


button4 = Button(gui, text=' 4 ', fg='black', bg='red',
command=lambda: press(4), height=1, width=7)
button4.grid(row=3, column0)


button5 = Button(gui, text=' 5 ', fg='black', bg='red',
```

```
command=lambda: press(5), height=1, width=7)
button5.grid(row=3, column1)


button6 = Button(gui, text=' 6 ', fg='black', bg='red',
command=lambda: press(6), height=1, width=7)
button6.grid(row=3, column2)


button7 = Button(gui, text=' 7 ', fg='black', bg='red',
command=lambda: press(7), height=1, width=)
button7.grid(row=4, column0)


button8 = Button(gui, text=' 8 ', fg='black', bg='red',
command=lambda: press(8), height=1, width=7)
button8.grid(row=4, column1)


button9 = Button(gui, text=' 9 ', fg='black', bg='red',
command=lambda: press(9), height=1, width=7)
button9.grid(row=4, column2)


button0 = Button(gui, text=' 0 ', fg='black', bg='red',
command=lambda: press(0), height=1, width=7)
button0.grid(row=5, column0)


plus = Button(gui, text=' + ', fg='black', bg='red',
```

```
command=lambda: press("+"), height=1, width=7)
plus.grid(row=2, column3)


minus = Button(gui, text=' - ', fg='black', bg='red',
command=lambda: press("-"), height=1, width=7)
minus.grid(row=3, column)


multiply = Button(gui, text=' * ', fg='black', bg='red',
command=lambda: press("*"), height=1, width=7)
multiply.grid(row=4, column3)


divide = Button(gui, text=' / ', fg='black', bg='red',
command=lambda: press("/"), height=1, width=7)
divide.grid(row=5, column3)


equal = Button(gui, text=' = ', fg='black', bg='red',
command=equalpress, height=1, width=7)
equal.grid(row=5, column2)


clear = Button(gui, text='Clear', fg='black', bg='red',
command=clear, height=1, width=7)
clear.grid(row=5, column='1')


# start the GUI
```

```
gui.mainloop() # 6)
```

1) The required tkinter library is imported

2) The UI is defined under the if __name__ == "__main__:" condition.

- This part of the condition is executed only when the file is executed as the main file.
- The following are defined in this UI,
- Tk() root window,
- Background color,
- title,
- window size,
- tk String variable to store the operator and operand values for calculation,
- buttons for digits 0 - 9, math symbols for +, - , *, /, =, delete button.
- Write separate functions for button press (press()), calculation function (equalpress()) and delete operation(clear())

3) Each button that is pressed forms a mathematical expression containing operand and operator values. It gets stored in expression variable. It is responsibility of user to create a valid math expression from the UI.

- The press() function receives the button value that is pressed and stores into the expression variable.
- In the Button class there is a command attribute which informs the function to be called and send the values upon pressing the button.
- In the command attribute there should not be () braces otherwise the function will get invoked during the UI creation itself.
- But to pass values to press function here, we have to convert the function call with values into a lambda function. So that the lambda function executes only upon pressing the button.

4) In the equalpress() function there is a evaluate() function, this is a system inbuilt function that evaluates any valid math expression and provides result. We use this function to perform all the calculator operations.

5) Write clear() function to clear the math expression that is present in variable expression.

6) Create the gui and keep it active with gui.mainloop()

Software Development Life Cycle

For the success of any application that is being built, efficient management of the people and process are required.

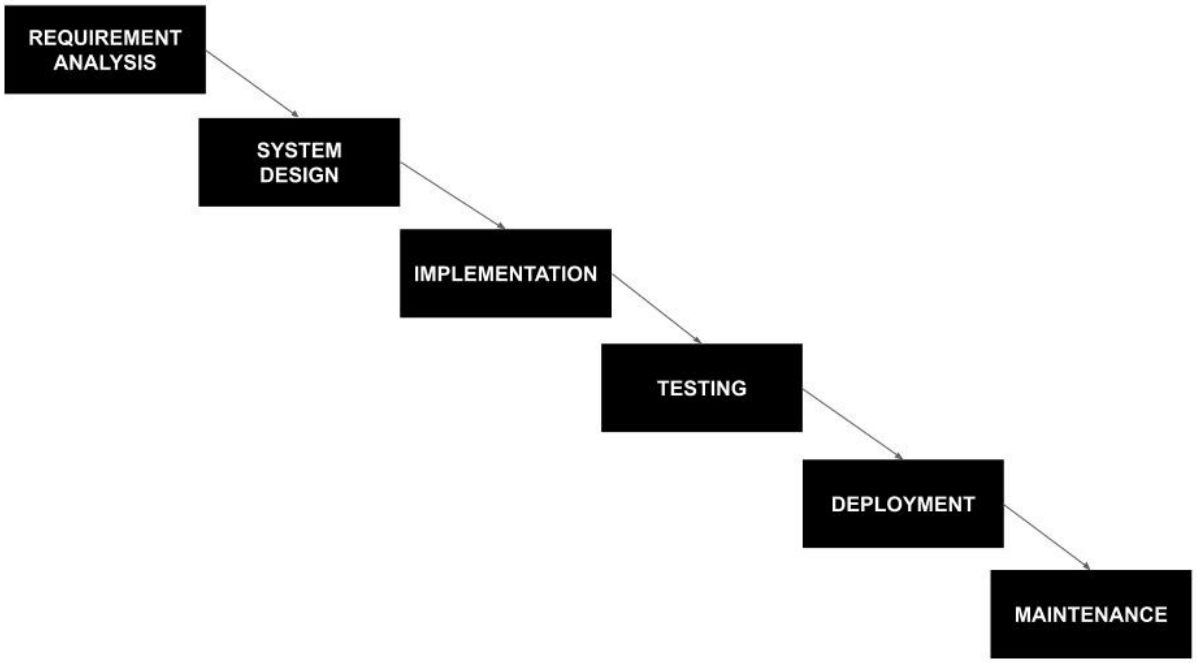
In building a software there are multiple stakeholders, it is not one person who conceives the idea and implements in an enterprise or a commercial application.

The most common types of SDLC models followed are,

- Waterfall model
- V-shaped model
- Iterative model
- Agile model

Waterfall model

In any software development process there are going to be multiple phases as follows,



In a typical waterfall model there can be 7 phases as follows, each phase is executed one after another. There is no going back to previous phase once the next phase is started, which means until a particular phase is complete it is never exited. Lets understand each and every phase one by one.

Requirement Analysis The requirement analysis phase captures all the user requirements and documents it either in a text document or a specialized software for storing requirements. Eg: DOORS - (Is a software application to store requirements. This kind of application is used in huge business projects & its applications)

System Design The System Design phase captures details as to how the software application implements user requirements. This will contain architecture diagrams that will illustrate the expected end software application. Also other necessary diagrams to communicate the behavior of application to other developers. These diagrams support the requirements in explaining itself to all stake holders including developers, testers and product owners.

Implementation This phase of implementation is where all the requirements are converted into working software. The development happens in this phase. Requirements are allocated to developers if there are many people involved. Each developer completes implementation and ensure that doesn’t affect the already existing functionality developed earlier.

Testing In the testing phase the developed application will be tested by engineers, either manually by using the software and observing its behavior or by automating the same process through another program for the purpose of testing. This testing is important for many reasons.

This will ensure the software is working as expected to all the stakeholders and gives confidence. so that it can be delivered to the end user of the application.

There can also be government regulations and documentation of testing before a particular software is put to use for the end user.

In case of any failures that will be documented and fixed in a future version, the timeline will depend upon the release schedule determined by the project.

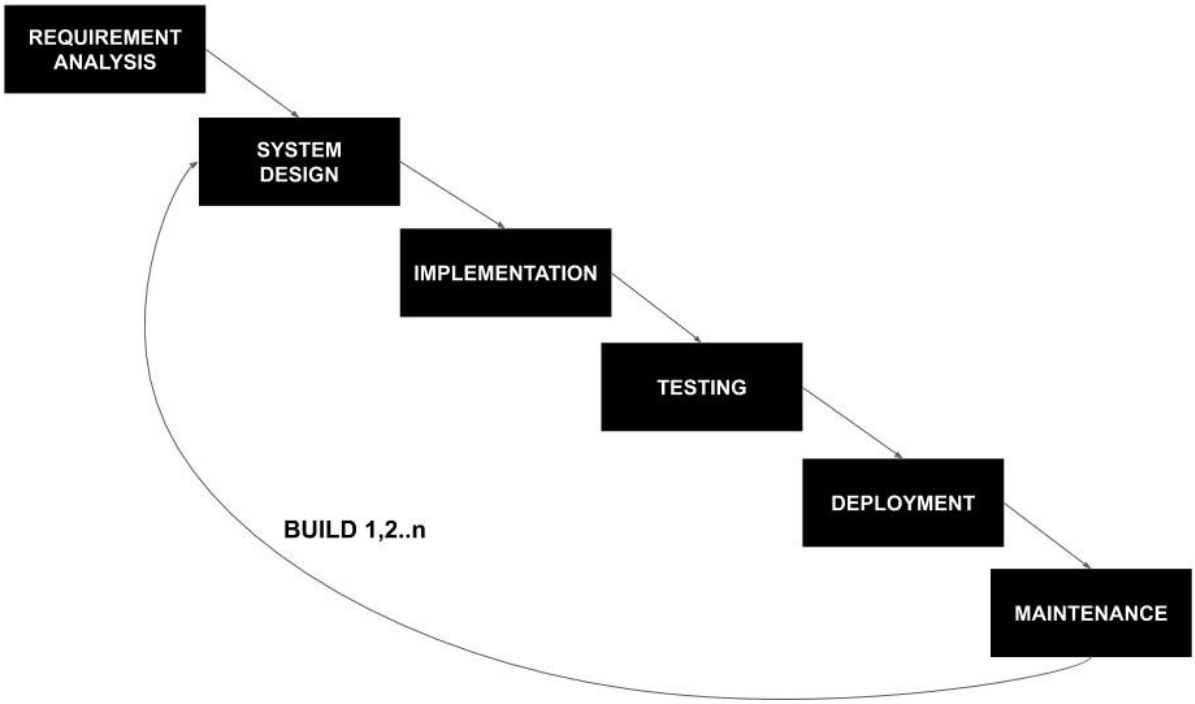
Deployment Deployment is when the software application is in a environment from where the end user starts using the software. This environment will have special requirements to meet purpose of use case criteria. Like for example,

- Network and hardware requirements to support multiple people accessing the software simultaneously.
- Write protected storage.
- Connectivity with backup servers, storage, etc.

Maintenance In the Maintenance phase the software is monitored for any errors, failures and they are resolved in the software by updating it in future versions.

The end software executable produced can be termed as software build. In a waterfall model there will be only one build produced for a single software release for deployment.

Iterative model

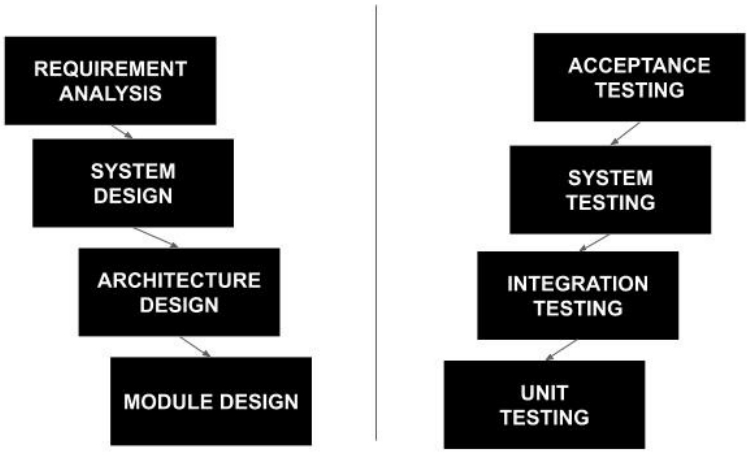


An iterative model is similar to waterfall, except that there can be multiple iterations in executing the complete process towards releasing a particular version to the end user.

Only a set of requirements are taken initially and it goes through all the phases in the process, the requirements are then incremented in the next iteration and a another version of software build is produced. Likewise until all the requirements are implemented only then the final software becomes complete. It is up to the product owner whether the incremental builds are to be provided to the end users.

The individual descriptions of each phase is similar to the water fall model and refer from previous section.

V-shaped model



A V-shaped model draws parallels between development and testing phases. In a typical SDLC process the longer the time taken to find a bug or error the costlier the effort to fix it. This kind of model is suitable for projects which require extensive testing.

The individual development phase and test phase it corresponds to are,

1) Module design corresponds to Unit testing,

2) Architecture design corresponds to Integration testing

3) System design corresponds to System Testing

4) Requirement analysis corresponds to Acceptance Testing.

lets look at each phase present in this model.

Requirement Analysis The requirement analysis phase captures all the user requirements and documents it either in a text document or a specialized software for storing requirements. Eg: DOORS - (Is a software application to store requirements. This kind of application is used in huge business projects & its applications)

System Design The System Design phase captures details as to how the software application implements user requirements. This will contain architecture diagrams that will illustrate the expected end software application. Also other necessary diagrams to communicate the behavior of application to other developers. These diagrams support the requirements in explaining itself to all stake holders including developers, testers and product owners.

Architecture Design An architecture design is a broken down version of overall system design. This will contain finer details of a specific sub system like class implementation details.

Module Design The finer details are implemented by writing code in the programming language. All sub systems are implemented one by one and integrated together.

Unit Testing In the unit testing phase individual subsystem referred as modules are tested by writing unit test cases.

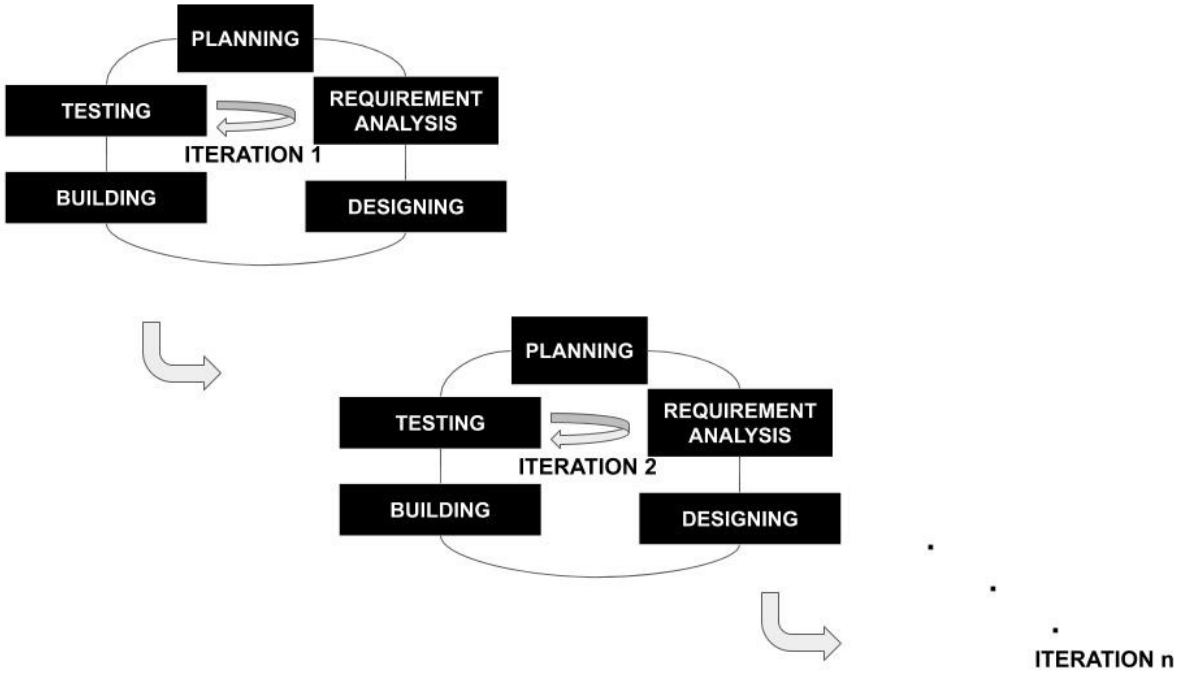
Integration Testing In the integration test phase multiple modules integrated together are testing by writing integration test code.

System Testing The system testing is another level of validation, it tests the behavior of the system. It can be in any form, either by writing automated test code that execute on a special test application or it can be a manual verification by using the software and its observation.

Acceptance Testing Acceptance testing is a final test performed, the sequence of tests are mutually agreed upon by all stakeholders.

In any of the above test phases all the procedure, sequence of steps and the observations are documented and reviewed by developers and testers.

Agile model



Agile model adopts flexible ways to implement a software. All its phases are explained below. The important criteria to be considered in agile are, the project is split in to multiple iterations bound by a fixed time frame. Outcome of each time bound iteration is a working build that can be used. Every iteration has incremental features added and it goes through all the phases of development.

Planning The required planning for resources, requirements and approvals are done in this phase.

Requirement Analysis In requirement analysis the detailed are requirements are documented and proceeded to next phase.

Designing In designing the architecture diagram for involved components are documented.

Building The required program/code is written in this phase of development.

Testing The testing of the new features and ensuring the general working test (also referred as sanity test) are done in this phase.

In agile all tasks are performed collaboratively and processes will be simplified for quick turn around time between people involved.

System Design Communication Diagrams using UML

A system design can be communicated to others with various design diagrams in order to bring clarity and uniform understanding to all the parties involved.

Some of the important design diagrams are,

1) Class Diagram - In any programming language class’es are created as fundamental building elements in a program. Illustrating how different class’es relate to each other gives a better idea before implementation. This is an important diagram from a architecture stand point. Inheritance relationship being visualized should be implemented in the program.

2) Sequence Diagram - Every important class entity should send messages to each other to achieve a functionality. A sequence diagram is used to capture this flow of messages. This is mostly from a development/testing, product owner point of view in understanding the application being developed.

3) Usecase Diagram - An use case diagram illustrates the perceived working of the entire application for an end user. It illustrates mostly the visible components for a end user or known components to him. Other internal components are abstracted.

UML Standards

Unified Modeling Language is a general purpose modeling language to visualize a system design. It is accepted as an industry standard to follow a common modeling language for visualizing.

There are multiple proprietary and free softwares to draw UML diagrams. It can be drawn in a powerpoint file also. But using specialized software will speed up the process and helps maintain revisions to the UML model.

One of the suggested tool I have is PlantUML, mainly because it is free to use. This can be used from inside pycharm IDE. However a little bit of commands are required to be learnt this way. There are other paid tools which will offer UI/UX features to draw UML models and offers commands as optional.

To install inside Pycharm, from the preferences window of pycharm, search for plugin and install - PlantUML by "Eugene Steinberg"

To know more about the standalone PlantUML or different design models that can be produced, take a look at the user guide of PlantUML -

http://plantuml.com/guide

Performing User Research & Presentation

Many people use python language for research and analysis of data they have. It is not necessary to use programming language only to build applications for someone else.

For example, a marketing team can have various data which they might be interested in analyzing and find insights quickly to make decisions for the next campaign or month. It will be quick and easy this way, instead of focusing on UI, UX, other aspects of the software if it is only going to be used by a very closed group of people to increase their productivity.

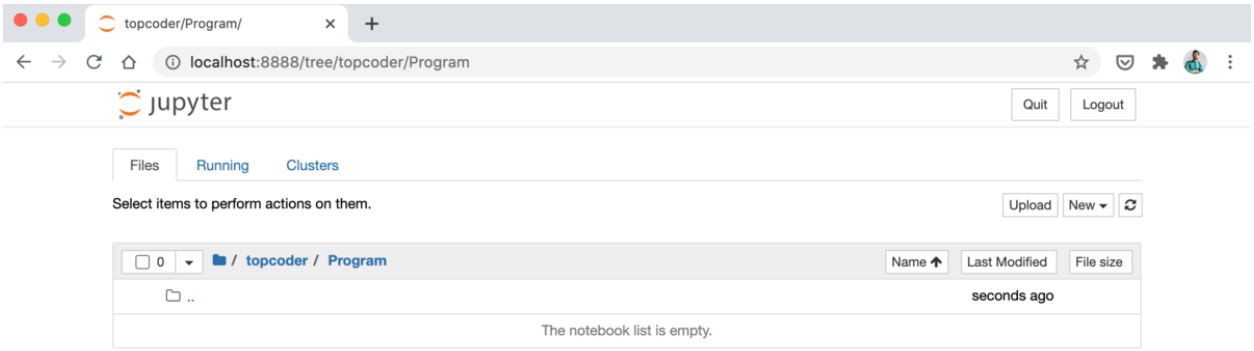
It can be used in any discipline of science & maths as well. It would be easy to review if the sequence of trials, steps performed and outcomes are stored for reference. And jupyter library in python serves this purpose.

A file is called a notebook in jupyter. Install jupyter library in terminal window and the following command. This will launch an external web application hosted in localhost.

```
pip install jupyter

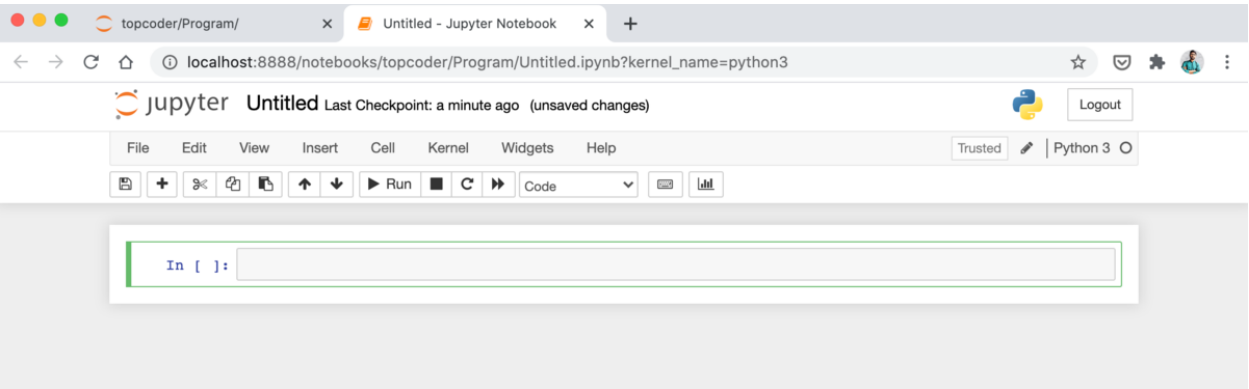
jupyter notebook
```

The web application looks similar to this view.



This shows the current working directory folder. Click on new to create a ipynb file, which is called a notebook. Inside which we will be able to do user research and store results.

A notebook view for sample,



There will be one cell displayed in a new file, where any number of lines of valid python statements can be typed. To execute contents present inside this cell, hit the play button. This will connect to a python interpreter instance, execute lines of code and returns output below the cell.

Similarly any number of cells can be created by simply clicking the + button from the task bar. The idea of using this ipynb notebook is to break down the code into smaller chucks and retrieve its results and store it in the notebook itself. This file can be saved and once opened again previous execution results will be present. It can executed again from the cells by clicking the + button.

This should provide a fair idea and introduction for this application, explore more by using the options from menu, most are self explainable and user friendly.

List of IDEs

I did not introduce any specific development tool in the installation section. Only installed python and started from terminal. You would have also used the same terminal or there are plenty of options to make typing and executing the code easier. This we can call as Integrated Development Environment. Few tools to be mentioned in this list,

- Python IDLE
- Anaconda & Spyder
- Pycharm
- Visual Studio Code
- Eclipse & PyDev

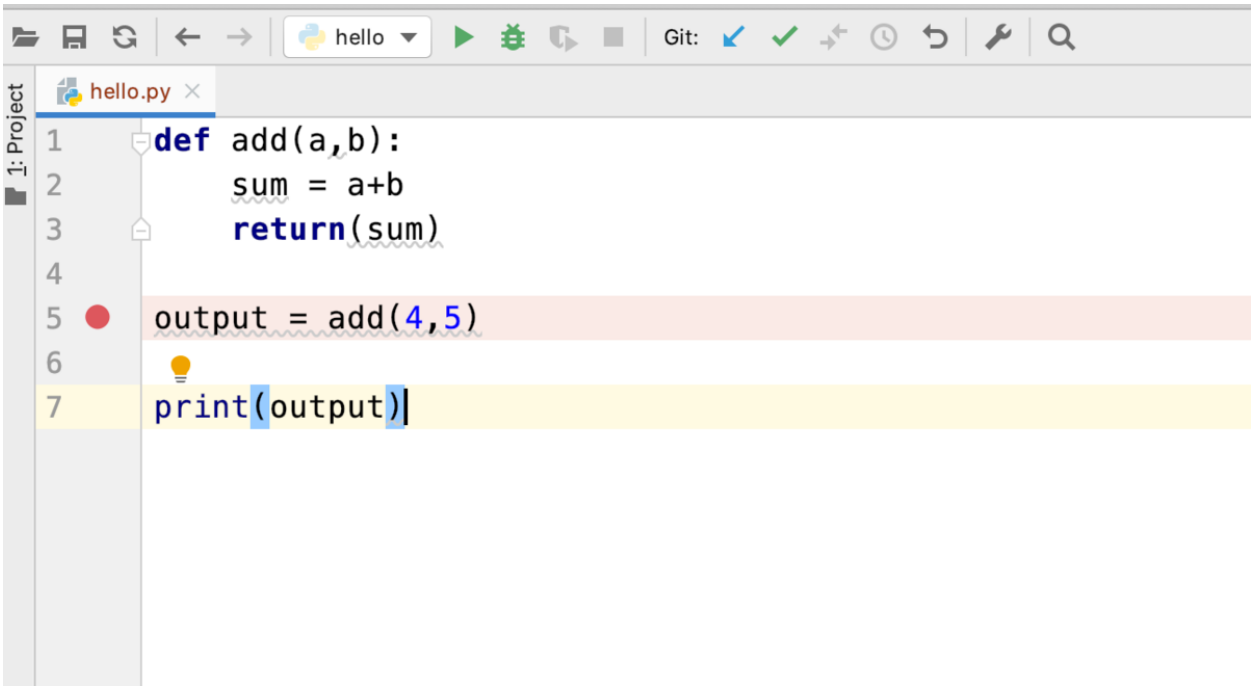
All of these are available as free to use & open source versions. My personal suggestion would be Pycharm, I am in no way associated with the company behind the software. But based on personal experience using the software,it offers powerful features and makes development easier.

Debugging & Performance Improvement

Python provides debugging support for the source code using inbuilt library pdb. We will have to update the source code to make use of its features. Before seeing the feature, let me introduce the need for debugging. Whenever we write some program in a file and execute there may be multiple modules and packages used. If there is error we will have to analyze the complete program to figure out the reason for error and correct it.

To analyze we will have to follow the code execution flow. By default when we run the code all lines will be executed one by one in a fraction of second and simply throw the error in the end and line in which error has occurred. But what happened during the execution is of real interest to identify the root cause. To control the execution only and manually follow through the code we will execute the code in debugging mode after setting breakpoints. Breakpoints are locations in code where program should pause and provide manual control to execute one line at a time.

Take a look at the preview image. This shows a sample code.



I used Pycharm, below are a sequence of steps to debug any code. These are features available in most of the IDEs mentioned above.

Create a breakpoint by clicking against the line number, gets created by indicating red dot.

To execute right click and hit Debug

Now the program pauses at the line where breakpoint is created.

Press F10 to execute one single after that. Analyze and note down the values for analysis. F10 is called step over.

Pressing F10 will provide the final result out of executing one single line, it doesn't follow the function call or class that line leads to.

You can also use F11 to go the function call and execute one statement there after. F11 is called step into.

The function keys may differ between OS and also between different IDEs.

Keep continuing the execution using Step over and Step into till it leads to the error. You can also resume execution at any point, that will execute the code again until it reaches any other breakpoint if present.

You can also observe the values present in memory using the variable explorer window. After observing the flow of data, using the error and stack

trace and identify root and make necessary corrections in the code.

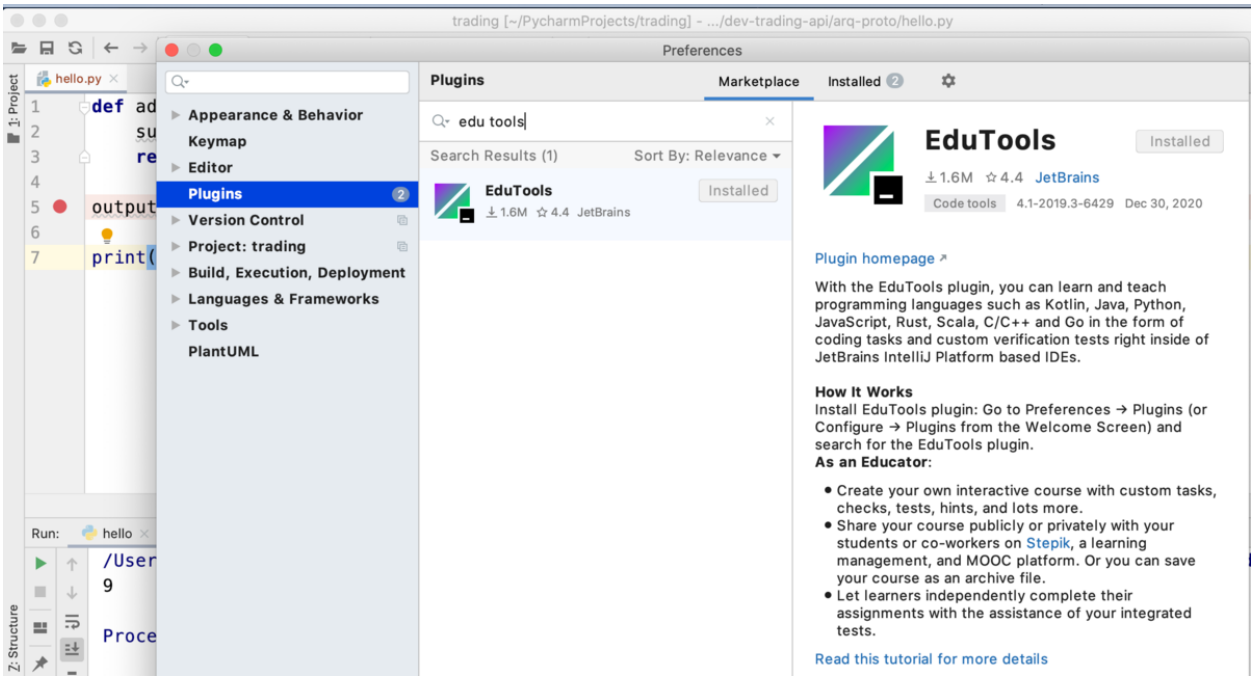
Following are some measures for performance improvement:

- Use lesser number of global variables. Prefer passing required values between class or functions. This reduces memory consumption.
- Remove unwanted libraries present in code. Delay loading libraries only when it is required instead of importing in the beginning.
- Try to reduce the dictionary usage in case of a memory constrained programming requirement. Usually shouldn’t be a problem for general purpose program.
- Tuples use lesser memory compared to a list. Ensure the right usage accordingly.
- Avoid using loops, instead see if a set and mathematical operation will provide the same result.
- Finally logic or the algorithm plays a vital role in speeding up the application. using lesser loops, avoiding repeated calculations, reduce file access times (a smaller file size or using a database, etc).

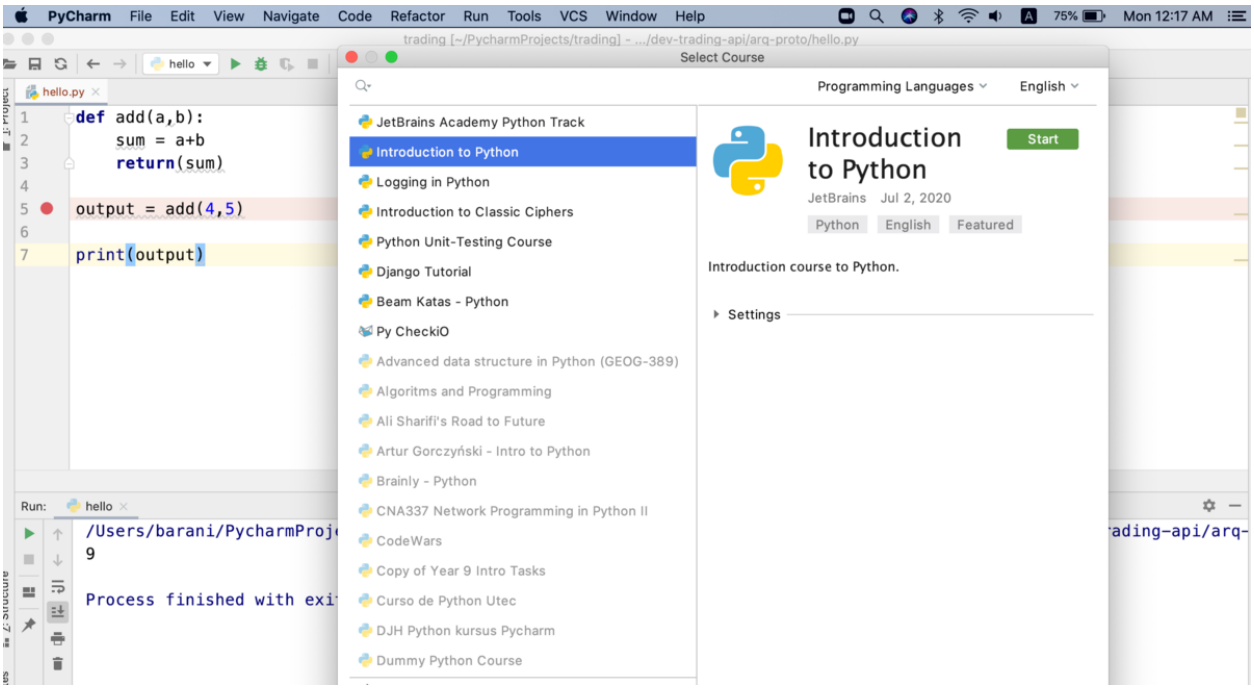
Learning Tools (Pycharm edu tools)

There are other fun ways to learn the basics. To keep you motivated and become comfortable with programming. My favorite recommendation would be to use the pycharm edu tools. This is a special feature inside pycharm IDE. Installing that will help you to access guided problem solving module.

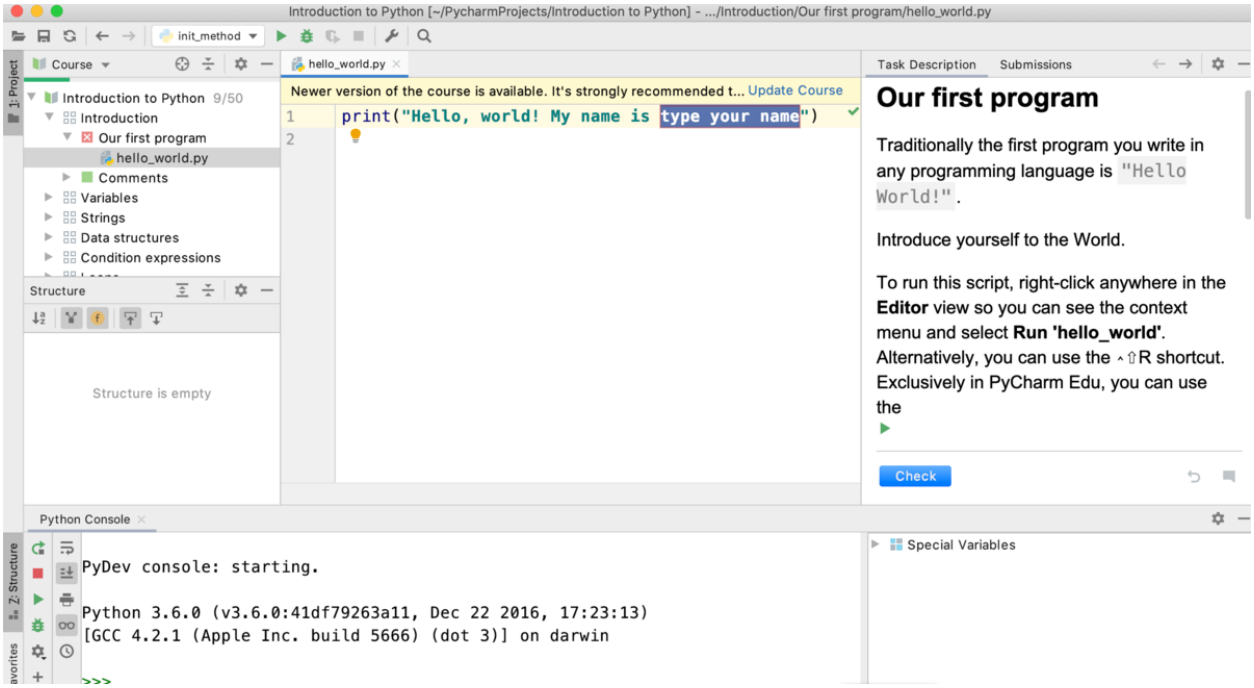
Let me provide few steps to setup. Install pycharm edu tools from the preference menu.



From the file menu, Select Learn & Teach and click browse courses. In the list of courses recommend to start with Introduction to Python and you can also try any other course as you wish.



This will ask to setup a python project. Once setup on the left side project view you will be able to see all lessons.



A course is organized into multiple Chapters and lessons. And individual lesson is a python script as you see in this image. For instructions or the problem statement read on the right panel.

Make edits and write solutions for the problem inside the already open python file. And run as directed. If the solution is right it would say PASS/ FAIL accordingly. There will be sufficient hint and data provided in the instruction panel, read it again if you get questions the first time. you can also google as a final resort if you are not able to arrive at a solution with provided directions.

We have come to the end of this Book. congratulations for reading it till this point. Thanks for buying this book, I believe you would have gained knowledge reading this material.

For suggestions or feedback you can right to my email – [hello@baranikumar.space]

Do subscribe to my newsletter @ [https://1pagepythonbook.club]

Your positive reviews on the book page is greatly appreciated.

Smashwords:

https://www.smashwords.com/books/view/1067036