Python

Programming

The Ultimate Guide

Hello world!

CONTENTS

*****

UNIT I

1

INTRODUCTION

Unit Structure

1.0 OBJECTIVES

After reading through this chapter, you will be able to –
* To understand and use the basic of python.
* To understand the history and features of python programming.
* To understand the installation of python.
* To handle the basis errors in python.
* To understand the difference between brackets, braces and parenthesis.

1.1   INTRODUCTION: THE   PYTHON PROGRAMMING LANGUAGE

* Python is an object-oriented, high level language, interpreted, dynamic and multipurpose programming language.
* Python is not intended to work on special area such as web programming. That is why it is known as multipurpose because it can be used with web, enterprise, 3D CAD etc.

1

- We don't need to use data types to declare variable because it is dynamically typed so we can write a=10 to declare an integer value in a variable.

- Python makes the development and debugging fast because there is no compilation step included in python development.

1.2 HISTORY

- Python was first introduced by Guido Van Rossum in 1991 at the National Research Institute for Mathematics and Computer Science, Netherlands.

- Though the language was introduced in 1991, the development began in the 1980s. Previouslyvan Rossum worked on the ABC language at CentrumWiskunde & Informatica (CWI) in the Netherlands.

- The ABC language was capable of exception handling and interfacing with the Amoeba operating system. Inspired by the language, Van Rossum first tried out making his own version of it.

- Python is influenced by programming languages like: ABC language, Modula-3, Python is used for software development at companies and organizations such as Google, Yahoo, CERN, Industrial Light and Magic, and NASA.

- Why the Name Python?

- Python developer, Rossum always wanted the name of his new language to be short, unique, and mysterious.

- Inspired by Monty Python's Flying Circus, a BBC comedy series, he named it Python.

1.3 FEATURES

There are a lot of features provided by python programming language as follows

1. Easy to Code:
- Python is a very developer-friendly language which means that anyone and everyone can learn to code it in a couple of hours or days.
- As compared to other object-oriented programming languages like Java, C, C++, and C#, Python is one of the easiest to learn.

2. Open Source and Free:
- Python is an open-source programming language which means that anyone can create and contribute to its development.

- Python has an online forum where thousands of coders gather daily to improve this language further. Along with this Python is free to download and use in any operating system, be it Windows, Mac or Linux.

3. Support for GUI:

- GUI or Graphical User Interface is one of the key aspects of any programming language because it has the ability to add flair to code and make the results more visual.

- Python has support for a wide array of GUIs which can easily be imported to the interpreter, thus making this one of the most favorite languages for developers.

4. Object-Oriented Approach:

- One of the key aspects of Python is its object-oriented approach. This basically means that Python recognizes the concept of class and object encapsulation thus allowing programs to be efficient in the long run.

5. Highly Portable:

- Suppose you are running Python on Windows and you need to shift the same to either a Mac or a Linux system, then you can easily achieve the same in Python without having to worry about changing the code.

- This is not possible in other programming languages, thus making Python one of the most portable languages available in the industry.

6. Highly Dynamic

- Python is one of the most dynamic languages available in the industry today. What this basically means is that the type of a variable is decided at the run time and not in advance.

- Due to the presence of this feature, we do not need to specify the type of the variable during coding, thus saving time and increasing efficiency.

7. Large Standard Library:

- Out of the box, Python comes inbuilt with a large number of libraries that can be imported at any instance and be used in a specific program.

- The presence of libraries also makes sure that you don't need to write all the code yourself and can import the same from those that already exist in the libraries.

1.4 INSTALLING PYTHON

- To install Python, firstly download the Python distribution from official website of python (www.python.org/ download).

- Having downloaded the Python distribution now execute it.

- Setting Path in Python:

Before starting working with Python, a specific path is to set to set path follow the steps:

Right click on My Computer--> Properties -->Advanced System setting -->Environment Variable -->New

In Variable name write path and in Variable value copy path up to C://Python (i.e., path where Python is installed). Click Ok ->Ok.
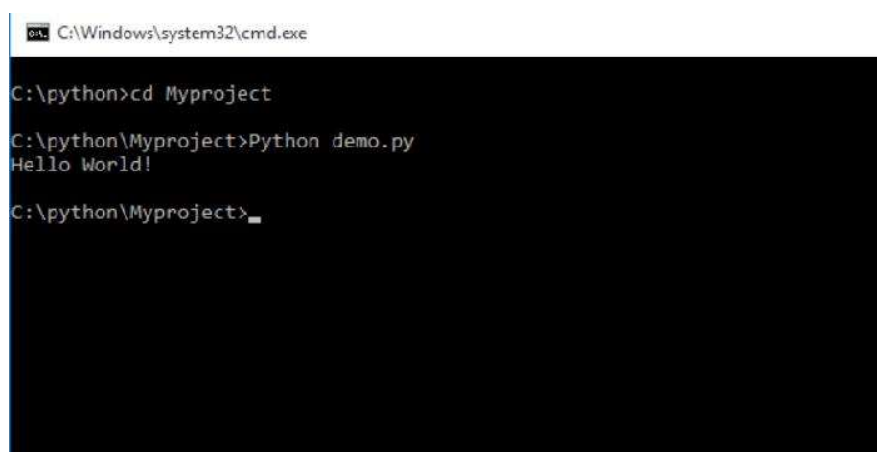
1.5 RUNNING PYTHON PROGRAM:

There are different ways of working in Python:

1) How to Execute Python Program Using Command Prompt:
If you want to create a Python file in .py extension and run. You can use the Windows command prompt to execute the Python code.

Example:

- Here is the simple code of Python given in the Python file demo.py. It contains only single line code of Python which prints the text "Hello World!" on execution.

- So, how you can execute the Python program using the command prompt. To see this, you have to first open the command prompt using the 'window+r' keyboard shortcut. Now, type the word 'cmd' to open the command prompt.

- This opens the command prompt with the screen as given below. Change the directory location to the location where you have just saved your Python .py extension file.

- We can use the cmd command 'cd' to change the directory location. Use 'cd..' to come out of directory and "cd" to come inside of the directory. Get the file location where you saved your Python file.


```
C:\Windows\system32\cmd.exe

C:\python>cd Myproject

C:\python\Myproject>Python demo.py
Hello World!

C:\python\Myproject>_
```

- To execute the Python file, you have to use the keyword 'Python' followed by the file name with extension.py See the example given in the screen above with the output of the file.

4

2) Interactive Mode to Execute Python Program:

- To execute the code directly in the interactive mode. You have to open the interactive mode. Press the window button and type the text "Python". Click the "Python 3.7(32 bit) Desktop app" as given below to open the interactive mode of Python.

- You can type the Python code directly in the Python interactive mode. Here, in the image below contains the print program of Python.

- **Press the enter button** to execute the print code of Python. The output gives the text "Hello World!" after you press the enter button.

- Type any code of Python you want to execute and run directly on interactive mode.

5

3) Using IDLE (Python GUI) to Execute Python Program:

- Another useful method of executing the Python code. Use the Python IDLE GUI Shell to execute the Python program on Windows system.

- Open the Python IDLE shell by pressing the window button of the keyboard. Type "Python" and click the "IDLE (Python 3.7 32 bit)" to open the Python shell.

- Create a Python file with .py extension and open it with the Python shell. The file looks like the image given below.

- It contains the simple Python code which prints the text "Hello World!". In order to execute the Python code, you have to open the 'run' menu and press the 'Run Module' option.
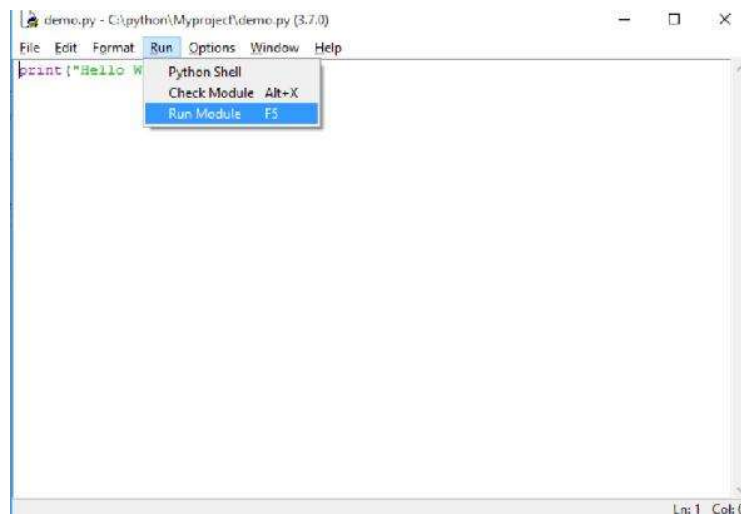
- A new shell window will open which contains the output of the Python code. Create your own file and execute the Python code using this simple method using Python IDLE.
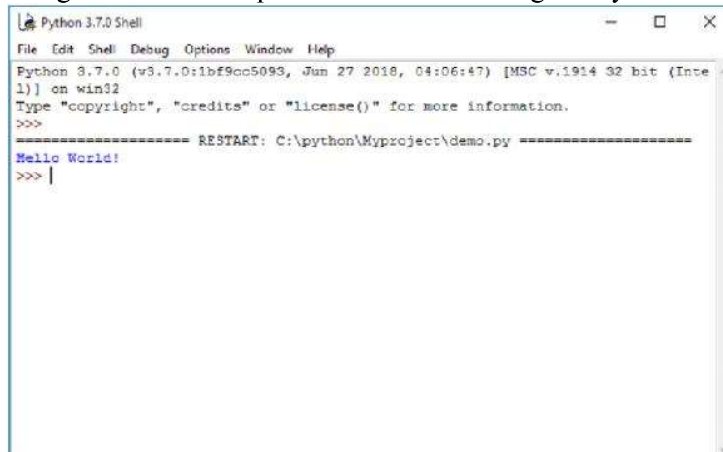
1.6 DEBUGGING

- Debugging means the complete control over the program execution. Developers use debugging to overcome program from any bad issues.

- debugging is a healthier process for the program and keeps the diseases bugs far away.

- Python also allows developers to debug the programs using pdb module that comes with standard Python by default.

- We just need to import pdb module in the Python script. Using pdb module, we can set breakpoints in the program to check the current status

- We can Change the flow of execution by using jump, continue statements.

1.6.1 Syntax Error:

- Errors are the mistakes or faults performed by the user which results in abnormal working of the program.

- However, we cannot detect programming errors before the compilation of programs. The process of removing errors from a program is called Debugging.

- A syntax error occurs when we do not use properly defined syntax in any programming language. For example: incorrect arguments, indentation, use of undefined variables etc.

Example:
age=16
if age>18:
print ("you can vote") # syntax error because of not using indentation
else
print ("you cannot vote") #syntax error because of not using indentation

1.6.2 Runtime Errors:

- The second type of error is a runtime error, so called because the error does not appear until after the program has started running.

- These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

- Some examples of Python runtime errors:
  - division by zero
  - performing an operation on incompatible types
  - using an identifier which has not been defined
  - accessing a list element, dictionary value or object attribute which doesn't exist
  - trying to access a file which doesn't exist

1.6.3 Semantic Errors:

- The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else.

- The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong.

- Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

8

1.6.4 Experimental Debugging:

- One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

- Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again.

- For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want.

- The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.

1.7 FORMAL AND NATURAL LANGUAGES

- Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

- Formal languages are languages that are designed by people for specific applications.

- For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules.

- Programming languages are formal languages that have been designed to express computations.

- Formal languages tend to have strict rules about syntax. For example, 3 + 3 = 6 is a syntactically correct mathematical statement.

- Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements.

- The second type of syntax rule pertains to the structure of a statement; that is, the way the tokens are arranged. The statement 3+ = 3 is illegal because even though + and = are legal tokens, you can't have one right after the other.

1.8 THE DIFFERENCE BETWEEN BRACKETS, BRACES, AND PARENTHESES:

- **Brackets [ ]:**
Brackets are used to define mutable data types such as list or list comprehensions.

Example:

To define a list with name as L1 with three elements 10,20 and 30

\>>> L1 = [10,20,30]

\>>> L1

[10,20,30]

- Brackets can be used for indexing and lookup of elements

Example:

\>>>L1[1] = 40

\>>>L1

[10,40,30]

Example: To lookup the element of list L1

\>>> L1[0]

10

- Brackets can be used to access the individual characters of a string or to make string slicing

Example:

Lookup the first characters of string

str>>>'mumbai'

\>>> str [0]

'm'

Example: To slice a string

\>>> str [1:4]

'umb'

Braces {}

- Curly braces are used in python to define set or dictionary.

Example:

Create a set with three elements 10,20,30.

\>>> s1 = {10,20,30}

\>>> type(s1)

<class 'set'>

Example:

Create a dictionary with two elements with keys, 'rollno' and 'name'

\>>> d1= {'rollno':101, 'name':' Vivek'}

\>>> type(d1)

<class 'dict'>

- Brackets can be used to access the value of dictionary element by specifying the key.

\>>> d1['rollno']

101

Parentheses ( )

- Parentheses can be used to create immutable sequence data type tuple.

Example: Create a tuple named 't1' with elements 10,20,30
>>> t1= (10,20,30)
>>> type(t1)
<class 'tuple'>

- Parentheses can be used to define the parameters of function definition and function call.

Example:
Multiply two numbers using a function
def mul(a,b):
        returns a*b
x=2
y=3
z=mul (2,3)
print(x,'*',y,'='z)

- In the function definition def mul(a,b) formal parameters are specified using parentheses

- In the function call z=mul (2,3) actual values are specified using parenthesis.

1.9 SUMMARY

- In this chapter we studied Introduction, history, features of Python Programming Language.

- We don't need to use data types to declare variable because it is dynamically typed so we can write a=10 to declare an integer value in a variable.

- In this chapter we are more focused on types of errors in python like syntax error, runtime error, semantic error and experimental debugging.

- Execution of Python Program Using Command Prompt and Interactive Mode to Execute Python Program.

- Also, we studied Difference between Brackets, Braces, and Parentheses in python.

1.10 REFERENCES

- www.journaldev.com
- www.edureka.com
- www.tutorialdeep.com

- www.xspdf.com

- Think Python by Allen Downey 1st edition.

- Python Programming for Beginners By Prof. Rahul E. Borate, Dr. Sunil Khilari, Prof. Rahul S. Navale.

1.11 UNIT END EXERCISE

1. Use a web browser to go to the Python website http: // python.org. This page contains information about Python and links to Python-related pages, and it gives you the ability to search the Python documentation.

   For example, if you enter print in the search window, the first link that appears is the documentation of the print statement. At this point, not all of it will make sense to you, but it is good to know where it is.

2. Start the Python interpreter and type help () to start the online help utility. Or you can type help ('print') to get information about the print statement.

*****

2

VARIABLES AND EXPRESSION

Unit Structure

2.0 OBJECTIVES

After reading through this chapter, you will be able to –
* To understand and use the basic datatypes of python.
* To understand the type conversion of variables in python programming.
* To understand the operators and operands in python.
* To understand the interactive mode and script mode in python.
* To understand the order of operations in python.

2.1 INTRODUCTION

* Variables in a computer program are not quite like mathematical variables. They are placeholders for locations in memory.

* Memory values consists of a sequence of binary digits (bits) that can be 0 or 1, so all numbers are represented internally in base 2.

* Names of variables are chosen by the programmer.

- Python is case sensitive, so myVariable is not the same as Myvariable which in turn is not the same as MyVariable.

- With some exceptions, however, the programmer should avoid assigning names that differ only by case since human readers can overlook such differences.

2.2 VALUES AND TYPES

- A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'.

- These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so-called because it contains a "string" of letters. Youcan identify strings because they are enclosed in quotation marks.

- If you are not sure what type a value has, the interpreter can tell you.
  >>>type ('Hello, World!')
  <type 'str'>
  >>> type (17)
  <type 'int'>

- Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating-point.
  >>> type (3.2)
  <type 'float'>

- What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.
  >>> type ('17')
  <type 'str'>
  >>> type ('3.2')
  <type 'str'>
  They are strings.

2.2.1 Variables:

- One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.
- An assignment statement creates new variables and gives them values:
  >>> message = 'Welcome to University of Mumbai'
  >>> n = 17
  >>> pi = 3.1415926535897932

- The above example makes three assignments. The first assigns a string to a new variable named message, the second gives the integer 17 to n, the third assigns the (approximate) value of π to pi.

- A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value.
  ```
  >>> type(message)
  <type 'str'>
  >>> type(n)
  <type 'int'>
  >>> type(pi)
  <type 'float'>
  ```

2.2.2 Variable Names and Keywords:

- Programmers generally choose names for their variables that are meaningful they document what the variable is used for.

- Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter.

- The underscore character, _, can appear in a name. It is often used in names with multiple words, such as my_name or airspeed_of_unladen_swallow.

- If you give a variable an illegal name, you get a syntax error:
  ```
  >>> 76mumbai= 'big city'
  SyntaxError: invalid syntax
  >>> more@ = 1000000
  SyntaxError: invalid syntax
  >>> class = 'Advanced python'
  SyntaxError: invalid syntax
  ```

- 76mumbai is illegal because it does not begin with a letter. more@ is illegal because it contains an illegal character, @. But what's wrong with class?

- It turns out that class is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

- Python has a lot of keywords. The number keeps on growing with the new features coming in python.

- Python 3.7.3 is the current version as of writing this book. There are 35 keywords in Python 3.7.3 release.

- We can get the complete list of keywords using python interpreter help utility.

- $ python3.7
  ```
  >>> help ()
  help> keywords
  ```

Here is a list of the Python keywords. Enter any keyword to get more help.

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

- You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

2.3 TYPE CONVERSION

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

2.3.1 Implicit Type Conversion:

- In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.
- example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

Example 1: Converting integer to float

num_int = 123

num_flo = 1.23

num_new = num_int + num_flo

print ("datatype of num_int:", type(num_int))

print ("datatype of num_flo:", type(num_flo))

print ("Value of num_new:", num_new)

print ("datatype of num_new:", type(num_new))

When we run the above program, the output will be:

datatype of num_int: <class 'int'>

datatype of num_flo: <class 'float'>

Value of num_new: 124.23

datatype of num_new: <class 'float'>

- In the above program, we add two variables num_int and num_flo, storing the value in num_new.

- We will look at the data type of all three objects respectively.

- In the output, we can see the data type of num_int is an integer while the data type of num_flo is a float.

- Also, we can see the num_new has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.

Example 2: Addition of string(higher) data type and integer(lower) datatype

num_int = 123

num_str = "456"

print ("Data type of num_int:", type(num_int))

print ("Data type of num_str:", type(num_str))

print(num_int+num_str)

When we run the above program, the output will be:

Data type of num_int: <class 'int'>

Data type of num_str: <class 'str'>

Traceback (most recent call last):

  File "python", line 7, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

- In the above program, we add two variables num_int and num_str.

- As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.

- However, Python has a solution for these types of situations which is known as Explicit Conversion.

2.3.2 Explicit Type Conversion:

- In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

- This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax:

<required_datatype>(expression)

Typecasting can be done by assigning the required data type function to the expression

Example 3: Addition of string and integer using explicit conversion
num_int = 123
num_str = "456"

print ("Data type of num_int:", type(num_int))
print ("Data type of num_str before Type Casting:", type(num_str))
num_str = int(num_str)
print ("Data type of num_str after Type Casting:", type(num_str))

num_sum = num_int + num_str

print ("Sum of num_int and num_str:", num_sum)
print ("Data type of the sum:", type(num_sum))

When we run the above program, the output will be:
Data type of num_int: <class 'int'>
Data type of num_str before Type Casting: <class 'str'>

Data type of num_str after Type Casting: <class 'int'>

Sum of num_int and num_str: 579

Data type of the sum: <class 'int'>

- In the above program, we add num_str and num_int variable.

- We converted num_str from string(higher) to integer(lower) type using int() function to perform the addition.

- After converting num_str to an integer value, Python is able to add these two variables.

- We got the num_sum value and data type to be an integer.

2.4 OPERATORS AND OPERANDS:

- Operators are particular symbols which operate on some values and produce an output.

- The values are known as Operands.

Example:

4 + 5 = 9

Here 4 and 5 are Operands and (+), (=) signs are the operators.

They produce the output 9

- Python supports the following operators:
  Arithmetic Operators.

Relational Operators.
Logical Operators.
Membership Operators.
Identity Operators

- **Arithmetic Operators:**

Operators	Description
//	Perform Floor division (gives integer value after division)
+	To perform addition
-	To perform subtraction
*	To perform multiplication
/	To perform division
%	To return remainder after division (Modulus)
**	Perform exponent (raise to power)

- **Arithmetic Operator Examples:**

>>> 10+20
30
>>> 20-10
10
>>> 10*2
20
>>> 10/2
5
>>> 10%3
1
>>> 2**3
8
>>> 10//3
3

- **Relational Operators:**

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

- **Relational Operators Examples:**

>>> 10<20

True

>>> 10>20

False

>>> 10<=10

True

>>> 20>=15

True

>>> 5==6

False

>>>5!=6

True

Logical Operators:

Operators	Description
and	Logical AND (When both conditions are true output will be true)
or	Logical OR (If any one condition is true output will be true
not	Logical NOT (Compliment the condition i.e., reverse)

Logical Operators Examples:

a=5>4 and 3>2

print(a)

True

b=5>4 or 3<2

print(b)

True

c=not(5>4)

print(c)

False

Membership Operators:

Operators	Description
in	Returns true if a variable is in sequence of another variable, else false.
not in	Returns true if a variable is not in sequence of another variable, else false.

Membership Operators Examples:

a=10

b=20

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

if (a in list):

   print ("a is in given list")

else:

    print ("a is not in given list")

if (b not in list):

   print ("b is not given in list")

else:

    print ("b is given in list")

Output:

>>>

a is in given list

b is given in list

Identity operators:

Operators	Description
is	Returns true if identity of two operands are same, else false
is not	Returns true if identity of two operands are not same, else false.

Identity operators Examples

a=20

b=20

if (a is b):

   print ("a, b has same identity")

else:

   print ("a, b is different")

b=10

if (a is not b):

   print ("a, b has different identity")

else:

   print ("a, b has same identity")

>>>

a, b has same identity

a, b has different identity

2.5 EXPRESSIONS

- An expression is a combination of values, variables, and operators.

- A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

17

x

x + 17

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statement: print and assignment.

- Technically an expression is also a statement, but it is probably simpler to think of them as different things. The important difference is that an expression has a value; a statement does not.

2.6 INTERACTIVE MODE AND SCRIPT MODE:

- One of the benefits of working with an interpreted language is that you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

- For example, if you are using Python as a calculator, you might type

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

The first line assigns a value to miles, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. So we learn that a marathon is about 42 kilometers.

- But if you type the same code into a script and run it, you get no output at all.

- In script mode an expression, all by itself, has no visible effect. Python actually evaluates the expression, but it doesn't display the value unless you tell it to:

miles = 26.2

print (miles * 1.61)

This behavior can be confusing at first.

- A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

- For example

```
print 1
x = 2
print x
```
produces the output
```
1
2
```
The assignment statement produces no output.

2.7 ORDER OF OPERATIONS

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first,

- 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.

- Exponentiation has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.

- Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.

- Operators with the same precedence are evaluated from left to right (except exponentiation). So, in the expression degrees / 2 * pi, the division happens first and the result is multiplied by pi. To divide by 2π, you can use parentheses or write degrees / 2 / pi.

- Example for Operator Precedence
```
>>> 200/200+(100+100)
201.0
>>>
>>> a=10
>>> b=20
>>> c=15
>>> (a+b)*(c+b)-150
900
>>> (a+b)*c+b-150
320
>>> a+b**2
410
```

```
>>> a or b + 20
10
>>> c or a + 20
15
>>> c and a + 20
30
>>> a and b + 20
40
>>> a>b>c
False
>>>
```

2.8 SUMMARY

- In this chapter we studied how to declare variables, expression and types of variables in python.

- We are more focuses on type conversion of variables in this chapter basically two types of conversion are implicit type conversion and explicit type conversion.

- Also studied types of operators available in python like arithmetic, logical, relational and membership operators.

- Focuses on interactive mode and script mode in python and order of operations.

2.9 UNIT END EXERCISE

1. Assume that we execute the following assignment statements:

   width = 17

   height = 12.0

   delimiter = '.'

For each of the following expressions, write the value of the expression and the type (of the value of
the expression).

   1. width/2

   2. width/2.0

   3. height/3

   4. 1 + 2 * 5

   5. delimiter * 5

Use the Python interpreter to check your answers

2. Type the following statements in the Python interpreter to see what they do:

5

x = 5

x + 1

Now put the same statements into a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

3. Write a program add two numbers provided by the user.

4. Write a program to find the square of number.

5. Write a program that takes three numbers and prints their sum. Every number is given on a separate line.

2.9 REFERENCES

- Think Python by Allen Downey 1st edition.

- Python Programming for Beginners By Prof. Rahul E. Borate, Dr. Sunil Khilari, Prof. Rahul S. Navale.

- https://learning.rc.virginia.edu/

- www.programiz.com

- www.itvoyagers.in

*****

3

CONDITIONAL STATEMENTS, LOOPING, CONTROL STATEMENTS

Unit Structure

3.0 OBJECTIVES

After reading through this chapter, you will be able to –

- To understand and use the conditional statementsin python.

- To understand the loop control in python programming.

- To understand the control statements in python.

- To understand the concepts of python and able to apply it for solving the complex problems.

3.1 INTRODUCTION

- In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability.

- The simplest form is the if statement:
  if x > 0:
   print 'x is positive'

The boolean expression after if is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens.

- if statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called compound statements.

- There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements. In that case, you can use the pass statement, which does nothing.

if x < 0:

pass   # need to handle negative values!

3.2 CONDITIONAL STATEMENTS

Conditional Statement in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

Story of Two if's:

Consider the following if statement, coded in a C-like language:

if (p > q)

{ p = 1;

q = 2;

}

Now, look at the equivalent statement in the Python language:

if p > q:

p = 1

q = 2

- ➤ what Python adds
- ➤ what Python removes

1) Parentheses are optional

if (x < y) ---> if x < y

2) End-of-line is end of statement

C-like languages   Python language

x = 1;   x = 1

3) End of indentation is end of block

Why Indentation Syntax?

A Few Special Cases

a = 1;   b = 2;   print (a + b)

You can chain together only simple statements, like assignments, prints, and function calls.

3.2.1 if statement:

Syntax

if test expression:

    statement(s)

- Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.

- If the test expression is False, the statement(s) is not executed.

- In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end.

- Python interprets non-zero values as True. None and 0 are interpreted as False.

Example: Python if Statement

```
# If the number is positive, we print an appropriate message
num = 3
if num > 0:
    print (num, "is a positive number.")
print ("This is always printed.")

num = -1
if num > 0:
    print (num, "is a positive number.")
print ("This is also always printed.")
```

When you run the program, the output will be:

3 is a positive number.

This is always printed.

This is also always printed.

- In the above example, num > 0 is the test expression.

- The body of if is executed only if this evaluates to True.

- When the variable num is equal to 3, test expression is true and statements inside the body of if are executed.

- If the variable num is equal to -1, test expression is false and statements inside the body of if are skipped.

- The print() statement falls outside of the if block (unindented). Hence, it is executed regardless of the test expression.

3.2.2 if-else statement:

Syntax

if test expression:

    Body of if

else:

    Body of else

- The if...else statement evaluates test expression and will execute the body of if only when the test condition is True.

- If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

Example of if...else

Program checks if the number is positive or negative

And displays an appropriate message

num = 3

Try these two variations as well.

num = -5

num = 0

if num >= 0:

    print ("Positive or Zero")

else:

    print ("Negative number")

Output:

Positive or Zero

- In the above example, when num is equal to 3, the test expression is true and the body of if is executed and the body of else is skipped.

- If num is equal to -5, the test expression is false and the body of else is executed and the body of if is skipped.

- If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.

3.2.3 if...elif...else Statement:

Syntax

if test expression:

    Body of if

elif test expression:

    Body of elif

else:

    Body of else

- The elif is short for else if. It allows us to check for multiple expressions.
- If the condition for if is False, it checks the condition of the next elif block and so on.
- If all the conditions are False, the body of else is executed.
- Only one block among the several if...elif...else blocks is executed according to the condition.
- The if block can have only one else block. But it can have multiple elif blocks.

Example of if...elif...else:

'''In this program,

we check if the number is positive or

negative or zero and

display an appropriate message'''

num = 3.4

Try these two variations as well:

num = 0

num = -4.5

if num > 0:

print ("Positive number")

elif num == 0:

   print("Zero")

else:

print ("Negative number")

- When variable num is positive, Positive number is printed.
- If num is equal to 0, Zero is printed.
- If num is negative, Negative number is printed.

3.2.4 nested if –else:

- We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.
- Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

Python Nested if Example

'''In this program, we input a number

check if the number is positive or

negative or zero and display

an appropriate message

This time we use nested if statement'"

```python
num = float (input ("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
print ("Positive number")
else:
print ("Negative number")
```

Output1:

Enter a number: 5

Positive number

Output2:

Enter a number: -1

Negative number

Output3:

Enter a number: 0

Zero

3.3 LOOPING STATEMENTS

- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

- Programming languages provide various control structures that allow for more complicated execution paths.

- A loop statement allows us to execute a statement or group of statements multiple times.

3.3.1 for loop:

- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

- Syntax of for Loop

```python
for val in sequence:
    Body of for
```

- Here, val is the variable that takes the value of the item inside the sequence on each iteration.

- Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

- **Example: Python for Loop**

Program to find the sum of all numbers stored in a list

List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

variable to store the sum

sum = 0

iterate over the list

for val in numbers:

    sum = sum+val

print ("The sum is", sum)

When you run the program, the output will be:

The sum is 48

The range () function:

- We can generate a sequence of numbers using range () function. range (10) will generate numbers from 0 to 9 (10 numbers).

- We can also define the start, stop and step size as range (start, stop,step_size). step_size defaults to 1, start to 0 and stop is end of object if not provided.

- This function does not store all the values in memory; it would be inefficient. So, it remembers the start, stop, step size and generates the next number on the go.

- To force this function to output all the items, we can use the function list().

Example:

print(range(10))

print(list(range(10)))

print (list (range (2, 8)))

print (list (range (2, 20, 3)))

Output:

range (0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

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

[2, 5, 8, 11, 14, 17]

- We can use the range () function in for loops to iterate through a sequence of numbers. It can be combined with the len () function to iterate through a sequence using indexing. Here is an example.

\# Program to iterate through a list using indexing

city = ['pune', 'mumbai', 'delhi']

\# iterate over the list using index

for i in range(len(city)):

   print ("I like", city[i])

Output:

I like pune

I like mumbai

I like delhi

for loop with else:

- A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.
- The break keyword can be used to stop a for loop. In such cases, the else part is ignored.
- Hence, a for loop's else part runs if no break occurs.

Example:

digits = [0, 1, 5]

for i in digits:

   print(i)

else:

   print("No items left.")

When you run the program, the output will be:

0

1

5

No items left.

- Here, the for loop prints items of the list until the loop exhausts. When the for-loop exhausts, it executes the block of code in the else and prints No items left.
- This for...else statement can be used with the break keyword to run the else block only when the break keyword was not executed.

Example:

\# program to display student's marks from record

student_name = 'Soyuj'

marks = {'Ram': 90, 'Shayam': 55, 'Sujit': 77}

33

```
for student in marks:
    if student == student_name:
        print(marks[student])
        break
else:
    print ('No entry with that name found.')
```

Output:

No entry with that name found.

3.3.2 while loop:

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

- We generally use while loop when we don't know the number of times to iterate beforehand.

- Syntax of while Loop in Python

  while test_expression:

      Body of while

- In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True.

- After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

- In Python, the body of the while loop is determined through indentation.

- The body starts with indentation and the first unindented line marks the end.

- Python interprets any non-zero value as True. None and 0 are interpreted as False.

Example: Python while Loop

```
# Program to add natural
# numbers up to
# sum = 1+2+3+...+n
# To take input from the user,
# n = int (input ("Enter n: "))
n = 10
# initialize sum and counter
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1    # update counter
```

print the sum

print ("The sum is", sum)

When you run the program, the output will be:

Enter n: 10

The sum is 55

- In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program).

- We need to increase the value of the counter variable in the body of the loop. This is very important. Failing to do so will result in an infinite loop (never-ending loop).

While loop with else:

- Same as with for loops, while loops can also have an optional else block.

- The else part is executed if the condition in the while loop evaluates to False.

- The while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

- Example:

'''Example to illustrate

the use of else statement

with the while loop'''

counter = 0

while counter < 3:

   print ("Inside loop")

   counter = counter + 1

else:

   print ("Inside else")

Output:

Inside loop

Inside loop

Inside loop

Inside else

- Here, we use a counter variable to print the string Inside loop three times.

- On the fourth iteration, the condition in while becomes False. Hence, the else part is executed.

3.3.3 Nested Loops:

- Loops can be nested in Python similar to nested loops in other programming languages.

- Nested loop allows us to create one loop inside another loop.
- It is similar to nested conditional statements like nested if statement.
- Nesting of loop can be implemented on both for loop and while loop.
- We can use any loop inside loop for example, for loop can have while loop in it.

Nested for loop:

- For loop can hold another for loop inside it.
- In above situation inside for loop will finish its execution first and the control will be returned back to outside for loop.

Syntax

for iterator in iterable:

    for iterator2 in iterable2:

        statement(s) of inside for loop

    statement(s) of outside for loop

- In this first for loop will initiate the iteration and later second for loop will start its first iteration and till second for loop complete its all iterations the control will not be given to first for loop and statements of inside for loop will be executed.
- Once all iterations of inside for loop are completed then statements of outside for loop will be executed and next iteration from first for loop will begin.

Example1: of nested for loop in python:

for i in range (1,11):

    for j in range (1,11):

        m=i*j

        print (m, end=' ')

    print ("Table of ", i)

Output:

1 2 3 4 5 6 7 8 9 10 Table of 1

2 4 6 8 10 12 14 16 18 20 Table of 2

3 6 9 12 15 18 21 24 27 30 Table of 3

4 8 12 16 20 24 28 32 36 40 Table of 4

5 10 15 20 25 30 35 40 45 50 Table of 5

6 12 18 24 30 36 42 48 54 60 Table of 6

7 14 21 28 35 42 49 56 63 70 Table of 7

8 16 24 32 40 48 56 64 72 80 Table of 8

9 18 27 36 45 54 63 72 81 90 Table of 9

10 20 30 40 50 60 70 80 90 100 Table of 10

Example2: of nested for loop in python:

for i in range (10):

   for j in range(i):

      print ("*", end=' ')

   print (" ")

Output:

*

* *

* * *

* * * *

* * * * *

* * * * * *

* * * * * * *

* * * * * * * *

* * * * * * * * *

Nested while loop:

- While loop can hold another while loop inside it.
- In above situation inside while loop will finish its execution first and the control will be returned back to outside while loop.

Syntax

while expression:

   while expression2:

      statement(s) of inside while loop

   statement(s) of outside while loop

- In this first while loop will initiate the iteration and later second while loop will start its first iteration and till second while loop complete its all iterations the control will not be given to first while loop and statements of inside while loop will be executed.
- Once all iterations of inside while loop are completed than statements of outside while loop will be executed and next iteration from first while loop will begin.
- It is also possible that if first condition in while loop expression is False then second while loop will never be executed.

Example1: Program to show nested while loop

p=1

```
while p<10:
    q=1
    while q<=p:
        print (p, end=" ")
        q+=1
    p+=1
    print (" ")
```

Output:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9
```

Exampleb2: Program to nested while loop

```
x=10
while x>1:
    y=10
    while y>=x:
        print (x, end=" ")
        y-=1
    x-=1
    print(" ")
```

Output:

```
10
9 9
8 8 8
7 7 7 7
6 6 6 6 6
5 5 5 5 5 5
4 4 4 4 4 4 4
3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2 2
```

3.4 CONTROL STATEMENTS:

- Control statements in python are used to control the order of execution of the program based on the values and logic.
- Python provides us with three types of Control Statements:

Continue

Break

3.4.1 Terminating loops:

- The break statement is used inside the loop to exit out of the loop. It is useful when we want to terminate the loop as soon as the condition is fulfilled instead of doing the remaining iterations.
- It reduces execution time. Whenever the controller encountered a break statement, it comes out of that loop immediately.
- Syntax of break statement

```
for element in sequence:
    if condition:
        break
```

Example:

```
for num in range (10):
    if num > 5:
        print ("stop processing.")
        break
 print(num)
```

Output:

0

1

2

3

4

5

stop processing.

3.4.2 skipping specific conditions:

- The continue statement is used to skip the current iteration and continue with the next iteration.
- Syntax of continue statement:

```
for element in sequence:
    if condition:
        continue
```

Example of a continue statement:

for num in range (3, 8):

   if num == 5:

     continue

   else:

     print(num)

Output:

3

4

6

7

3.5 SUMMARY

- In this chapter we studied conditional statements like if, if-else, if-elif-else and nested if-else statements for solving complex problems in python.

- More focuses on loop control in python basically two types of loops available in python like while loop, for loop and nested loop.

- Studied how to control the loop using break and continue statements in order to skipping specific condition and terminating loops.

3.6 UNIT END EXERCISE

1. Print the squares of numbers from 1 to 10 using loop control.

2. Write a Python program to print the prime numbers of up to a given number, accept the number from the user.

3. Write a Python program to print the following pattern

1

23

456

78910

1112131415

4. Write a Python program to construct the following pattern, using a nested for loop.

         *

        * *

       * * *

      * * * *

     * * * * *

      * * * *

       * * *

        * *

         *

5. Write a Python program to count the number of even and odd numbers from a series of numbers.

Sample numbers: numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)

Expected Output:

Number of even numbers: 5

Number of odd numbers: 4

6. Write a Python program that prints all the numbers from 0 to 6 except 3 and 6

Note: Use 'continue' statement.

Expected Output: 0 1 2 4 5

7. Print First 10 natural numbers using while loop

8. Print the following pattern
   1
   1 2
   1 2 3
   1 2 3 4
   1 2 3 4 5

9. Display numbers from -10 to -1 using for loop

10. Print the following pattern

   *
   **
   ***
   *****
   ******

3.7 REFERENCES

- Think Python by Allen Downey 1st edition.
- Python Programming for Beginners By Prof. Rahul E. Borate, Dr. Sunil Khilari, Prof. Rahul S. Navale.
- www.programiz.com
- https://itvoyagers.in
- https://www.softwaretestinghelp.com
- https://pynative.com

*****

UNIT II

4

FUNCTIONS

Unit Structure

4.0 OBJECTIVES

After reading through this chapter, you will be able to –
- To understand and use the function calls.
- To understand the type conversion functions.
- To understand the math function.
- To adding new function.
- To understand the Parameters and Arguments.
- To understand the fruitful functions and void functions.
- To understand the boolean functions, Recursion, checking types etc.

4.1 INTRODUCTION

- One of the core principles of any programming language is, "Don't Repeat Yourself". If you have an action that should occur many

times, you can define that action once and then call that code whenever you need to carry out that action.

- We are already repeating ourselves in our code, so this is a good time to introduce simple functions. Functions mean less work for us as programmers, and effective use of functions results in code that is less error.

4.2 FUNCTION CALLS

What is a function in Python?

- In Python, a function is a group of related statements that performs a specific task.

- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

- Furthermore, it avoids repetition and makes the code reusable.

Syntax of Function

def function_name(parameters):

    """docstring"""

    statement(s)

Above shown is a function definition that consists of the following components.

1. Keyword def that marks the start of the function header.

2. A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.

3. Parameters (arguments) through which we pass values to a function. They are optional.

4. A colon (:) to mark the end of the function header.

5. Optional documentation string (docstring) to describe what the function does.

6. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).

7. An optional return statement to return a value from the function.

Example:

def greeting(name):

    """

    This function greets to

    the person passed in as

43

a parameter

"""

print ("Hello, " + name + ". Good morning!")

How to call a function in python?

- Once we have defined a function, we can call it from another function, program or even the Python prompt.

- To call a function we simply type the function name with appropriate parameters.

    >>> greeting('IDOL')

    Hello, IDOL. Good morning!

4.3 TYPE CONVERSION FUNCTIONS

- The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. Implicit Type Conversion

2. Explicit Type Conversion

1. Implicit Type Conversion:

- In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

- Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

Example 1: Converting integer to float

num_int = 123

num_float = 1.23

num_new = num_int + num_float

print ("datatype of num_int:", type(num_int))

print ("datatype of num_float:" type(num_float))

print ("Value of num_new:", num_new)

print ("datatype of num_new:", type(num_new))

Output:

datatype of num_int: <class 'int'>

datatype of num_float: <class 'float'>

Value of num_new: 124.23

datatype of num_new: <class 'float'>

44

Example 2: Addition of string(higher) data type and integer(lower) datatype

num_int = 123

num_str = "456"

print ("Data type of num_int:", type(num_int))

print ("Data type of num_str:", type(num_str))

print(num_int+num_str)

Output:

Data type of num_int: <class 'int'>

Data type of num_str: <class 'str'>

Traceback (most recent call last):

File "python", line 7, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

- In the above program,

- We add two variables num_int and num_str.

- As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.

- However, Python has a solution for these types of situations which is known as Explicit Conversion.

2. Explicit Type Conversion:

- In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

- This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax:

<required_datatype>(expression)

Example 3: Addition of string and integer using explicit conversion

num_int = 123

num_str = "456"

print ("Data type of num_int:", type(num_int))

print ("Data type of num_str before Type Casting:", type(num_str))

num_str = int(num_str)

print ("Data type of num_str after Type Casting:", type(num_str))

num_sum = num_int + num_str

45

print ("Sum of num_int and num_str:", num_sum)

print ("Data type of the sum:", type(num_sum))

Output:

Data type of num_int: <class 'int'>

Data type of num_str before Type Casting: <class 'str'>

Data type of num_str after Type Casting: <class 'int'>

Sum of num_int and num_str: 579

Data type of the sum: <class 'int'>

- Type Conversion is the conversion of object from one data type to another data type.
- Implicit Type Conversion is automatically performed by the Python interpreter.
- Python avoids the loss of data in Implicit Type Conversion.
- Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.
- In Type Casting, loss of data may occur as we enforce the object to a specific data type.

4.4 MATH FUNCTIONS

- The math module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using import math.

- For example
  # Square root calculation
  import math
  math.sqrt(4)

- Functions in Python Math Module

  Pi is a well-known mathematical constant, which is defined as the ratio of the circumference to the diameter of a circle and its value is 3.141592653589793.
  >>> import math

  >>>math.pi

  3.141592653589793

- Another well-known mathematical constant defined in the math module is **e**. It is called **Euler's number** and it is a base of the natural logarithm. Its value is 2.718281828459045.
  >>> import math
  >>>math.e
  2.718281828459045

46

- The math module contains functions for calculating various trigonometric ratios for a given angle. The functions (sin, cos, tan, etc.) need the angle in radians as an argument. We, on the other hand, are used to express the angle in degrees. The math module presents two angle conversion functions: degrees () and radians (), to convert the angle from degrees to radians and vice versa.

  >>> import math

  >>>math.radians(30)

  0.5235987755982988

  >>>math.degrees(math.pi/6)

  29.999999999999996

- math.log()

The math.log() method returns the natural logarithm of a given number. The natural logarithm is calculated to the base e.

>>> import math

>>>math.log(10)

2.302585092994046

- math.exp()

The math.exp() method returns a float number after raising e to the power of the given number. In other words, exp(x) gives e**x.

>>> import math

>>>math.exp(10)

22026.465794806718

- math.pow()

The math.pow() method receives two float arguments, raises the first to the second and returns the result. In other words, pow(4,4) is equivalent to 4**4.

>>> import math

>>>math.pow(2,4)

16.0

>>> 2**4

16

- math.sqrt()

The math.sqrt() method returns the square root of a given number.

>>> import math

>>>math.sqrt(100)

10.0

```
>>>math.sqrt(3)
```

1.7320508075688772

4.5 ADDING NEW FUNCTIONS

- So far, we have only been using the functions that come with Python, but it is also possible to add new functions.

- A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.

- Example:

```python
def print_lyrics():
    print ("I'm a lumberjack, and I'm okay.")
    print ("I sleep all night and I work all day.")
```

- def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

- The empty parentheses after the name indicate that this function doesn't take any arguments.

- The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented.

- By convention, the indentation is always four spaces .The body can contain any number of statements.

- The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote appears in the string.

- Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called repeat_lyrics:

```python
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call repeat_lyrics:

```python
>>> repeat_lyrics()
```

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

48

4.6 DEFINITIONS AND USES

- Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics ():
    print ("I'm a lumberjack, and I'm okay.")
    print ("I sleep all night and I work all day.")


def repeat_lyrics ():
    print_lyrics ()
    print_lyrics ()


repeat_lyrics ()
```

- This program contains two function definitions: print_lyrics and repeat_lyrics. Function definitions get executed just like other statements, but the effect is to create function objects.

- The statements inside the function do not get executed until the function is called, and the function definition generates no output.

4.6.1 Flow of Execution:

- In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution.

- Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

- A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

- When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

4.6.2 Parameters and Arguments:

- Some of the built-in functions we have seen require arguments. For example, when you call math.sin you pass a number as an argument. Some functions take more than one argument: math.pow takes two, the base and the exponent.

- Inside the function, the arguments are assigned to variables called parameters. Here is an example of a user-defined function that takes an argument.

- def print_twice(bruce):

    print(bruce)

    print(bruce)

- This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter twice.

    >>> print_twice('Spam')

    Spam

    Spam

    >>> print_twice (17)

    17

    17

    >>> print_twice(math.pi)

    3.14159265359

    3.14159265359

- The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_twice.

    >>> print_twice ('Spam '*4)

    Spam SpamSpamSpam

    Spam SpamSpamSpam

    >>> print_twice(math.cos(math.pi))

    -1.0

    -1.0

The argument is evaluated before the function is called, so in the examples the expressions 'Spam '*4 and math.cos(math.pi) are only evaluated once.

4.6.3 Variables and Parameters Are Local:

- When you create a variable inside a function, it is local, which means that it only exists inside the function.

- For example
    def cat_twice(part1, part2):
        cat = part1 + part2
        print_twice(cat)

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

>>> line1 = 'Bing tiddle '

>>> line2 = 'tiddle bang.'

>>> cat_twice(line1, line2)

Bing tiddle tiddle bang.

Bing tiddle tiddle bang.

- When cat_twice terminates, the variable cat is destroyed. If we try to print it, we get an exception:

>>> print cat

NameError: name 'cat' is not defined

- Parameters are also local. For example, outside print_twice, there is no such thing as bruce.

4.6.4 Stack Diagrams:

- To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

- Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure.
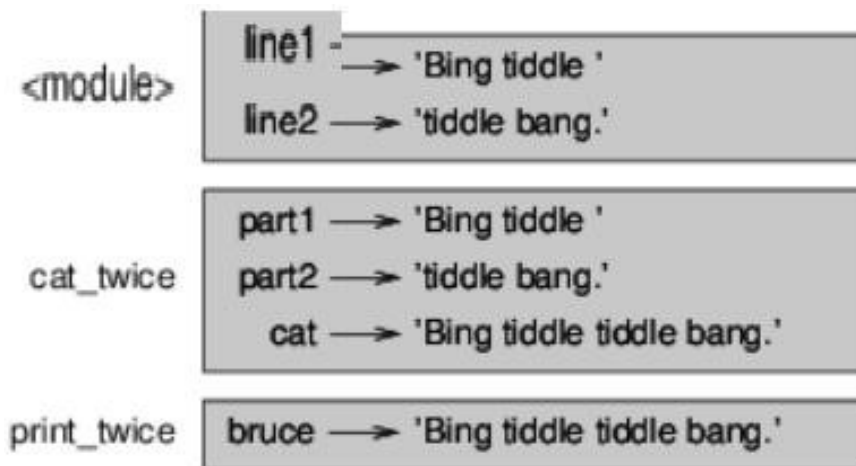
Fig. Stack Diagram

- The frames are arranged in a stack that indicates which function called which, and so on. In this example, print_twice was called by cat_twice, and cat_twice was called by __main__, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to __main__.

51

- Each parameter refers to the same value as its corresponding argument. So, part1 has the same value as line1, part2 has the same value as line2, and bruce has the same value as cat.

- If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called *that*, all the way back to __main__.

4.7 FRUITFUL FUNCTIONS AND VOID FUNCTIONS

- Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them fruitful functions. Other functions, like print_twice, perform an action but don't return a value. They are called void functions.

- When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

  x = math.cos(radians)

  golden = (math.sqrt(5) + 1) / 2

  When you call a function in interactive mode, Python displays the result:
  >>>math.sqrt(5)

  2.2360679774997898

- But in a script, if you call a fruitful function all by itself, the return value is lost forever

  math.sqrt(5)

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

- Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

  >>> result = print_twice('Bing')

  Bing

  Bing

  >>> print(result)

  None

The value None is not the same as the string 'None'. It is a special value that has its own type:

>>> print type(None)

<type 'NoneType'>

- The functions we have written so far are all void.

4.8 WHY FUNCTIONS?

- It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.

- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

4.9 IMPORTING WITH FROM, RETURN VALUES, INCREMENTAL DEVELOPMENT

- Importing with from:

Python provides two ways to import modules, we have already seen one:

>>> import math

>>> print math

<module 'math' (built-in)>

>>> print math.pi

3.14159265359

- If you import math, you get a module object named math. The module object contains constants like pi and functions like sin and exp.

But if you try to access pi directly, you get an error.

>>> print pi

Traceback (most recent call last):

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

NameError: name 'pi' is not defined

- As an alternative, you can import an object from a module like this:

>>> from math import pi

Now you can access pi directly, without dot notation.

>>> print pi

3.14159265359

Or you can use the star operator to import everything from the module:

>>> from math import *

>>> cos(pi)

-1.0

- The advantage of importing everything from the math module is that your code can be more concise.

- The disadvantage is that there might be conflicts between names defined in different modules, or between a name from a module and one of your variables.

4.10 BOOLEAN FUNCTIONS

- Syntax for boolean function is as follows

Bool([value])

- As we seen in the syntax that the bool() function can take a single parameter (value that needs to be converted). It converts the given value to True or False.

- If we don't pass any value to bool() function, it returns False.

- bool() function returns a boolean value and this function returns False for all the following values

1. None

2. False

3. Zero number of any type such as int, float and complex. For example: 0, 0.0, 0j

4. Empty list [], Empty tuple (), Empty String ".

5. Empty dictionary {}.

6. objects of Classes that implements __bool__() or __len()__ method, which returns 0 or False

- bool() function returns True for all other values except the values that are mentioned above.

- Example: bool() function

In the following example, we will check the output of bool() function for the given values. We have different values of different data types and we are printing the return value of bool() function in the output.

```
# empty list
lis = []
print(lis,'is',bool(lis))
# empty tuple
t = ()
print(t,'is',bool(t))
# zero complex number
```

```
c = 0 + 0j
print(c,'is',bool(c))
num = 99
print(num, 'is', bool(num))
val = None
print(val,'is',bool(val))
val = True
print(val,'is',bool(val))
# empty string
str = ''
print(str,'is',bool(str))
str = 'Hello'
print(str,'is',bool(str))
```

Output:

[] is False

() is False

0j is False

99 is True

None is False

True is True

 is False

Hello is True

4.11 MORE RECURSION, CHECKING TYPES:

* What is recursion?

Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

* In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.
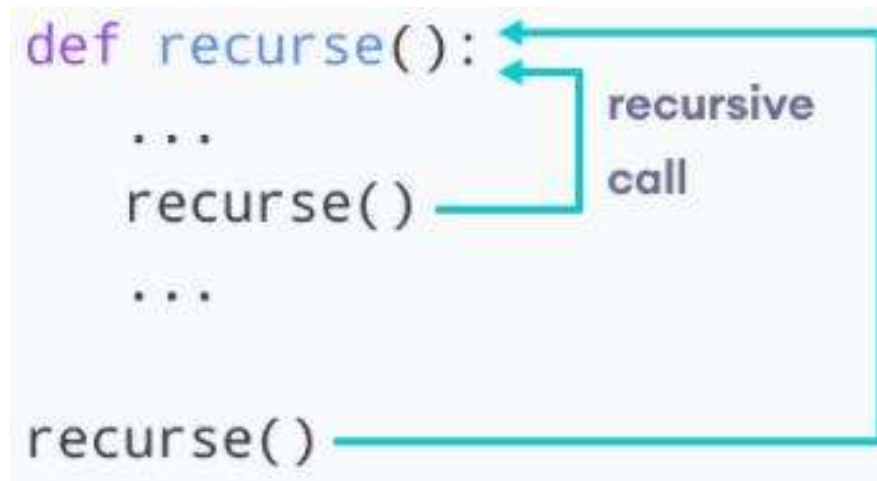
- Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

- Example of a recursive function

def factorial(x):

    """This is a recursive function

    to find the factorial of an integer"""

    if x == 1:

       return 1

    else:

       return (x * factorial(x-1))

num = 3

print("The factorial of", num, "is", factorial(num))

Output:

The factorial of 3 is 6

- In the above example, factorial () is a recursive function as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing the number.

- Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

factorial (3)   # 1st call with 3

3 * factorial (2)   # 2nd call with 2

3 * 2 * factorial (1)   # 3rd call with 1

3 * 2 * 1   # return from 3rd call as number=1

3 * 2   # return from 2nd call

6   # return from 1st call

4.12 SUMMARY

* In this chapter we studied function call, type conversion functions in Python Programming Language.

* In this chapter we are more focused on math function and adding new function in python.

* Elaborating on definitions and uses of function, parameters and arguments in python.

* Also studied fruitful functions and void functions, importing with from, boolean functions and recursion in python.

4.14 UNIT END EXERCISE

1. Python provides a built-in function called len that returns the length of a string, so the value of len('allen') is 5.

   Write a function named right_justify that takes a string named s as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

   >>> right_justify('allen')

           allen

2. Write a Python function to sum all the numbers in a list. Go to the editor
   Sample List : (8, 2, 3, 0, 7)
   Expected Output : 20

3. Write a Python program to reverse a string
   Sample String : "1234abcd"
   Expected Output : "dcba4321"

4. Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.

5. Write a Python program to print the even numbers from a given list.
   Sample List: [1, 2, 3, 4, 5, 6, 7, 8, 9]

   Expected Result: [2, 4, 6, 8]

4.13 REFERENCES

- https://www.tutorialsteacher.com/python/math-module
- https://greenteapress.com/thinkpython/html/thinkpython004.html
- https://beginnersbook.com/
- https://www.programiz.com/python-programming/recursion
- www.journaldev.com
- www.edureka.com
- www.tutorialdeep.com
- www.xspdf.com
- Think Python by Allen Downey 1st edition.

*****

<div align="right">**5**</div>

STRINGS

Unit Structure

5.0 OBJECTIVES

- To learn how to crate string in Python
- To study looping and counting in Python
- To write programs for creating string methods in Python
- To understand various operators and string operation of Python
- To learn the string traversal with a for loop in Python

5.1 A STRING IS A SEQUENCE

There are numerous types of sequences in Python. Strings are a special type of sequence that can only store characters, and they have a special notation. Strings are **sequences** of characters and are immutable.

A string is a sequence of characters. We can access the characters one at a time with the bracket operator:

```
>>>food = 'roti'
>>>letter = food[1]
```

The second statement retrieves the character at index position one from the food variable and assigns it to the letter variable.The expression

in brackets is called an *index*. The index indicates which character in the sequence you required.

Example.
'**I want to read book**'. This is a **string**, It has been surrounded in single quotes.

Declaring Python String
String literals in Python
String literals are surrounded by **single quotes** or **double-quotes**.

'I want to read book'

"I want to read book'"

You can also surround them with **triple quotes (groups of 3 single quotes or double quotes)**.

"""I want to read book"""

"' I want to read book'"

Or using **backslashes**.

>> 'Thank

Good Morning Sir !'
'ThanksGood Morning Sir!'

You cannot **start** a **string** with a **single quote** and **end** it with a **double quote.**.

But if you surround a string with single quotes and also want to use single quotes as part of the string, you have to escape it with a **backslash (\)**.

'Send message to Madam\'s son '

This statement causes a **SyntaxError**:

'Send message to Madam's son'

You can also do this with double-quotes. If you want to ignore **escape sequences**, you can create a **raw string** by using an **'r' or 'R' prefix** with the string.

Common escape sequences:
- \\ Backslash
- \n Linefeed
- \t Horizontal tab
- \' Single quotes
- \" Double quotes

We can assign a string to a **variable**.

```
name='Sunil'
```

You can also create a string with the **str() function**.

```
>>> str(567)
```

Output:
'567'

```
>>> str('Shri')
```

Output:
'Shri'

5.2. TRAVERSAL AND THE FOR LOOP: BY ITEM

A lot of computations involve processing a collection one item at a time. For strings this means that we would like to process one character at a time. Often we start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal.

We have previously seen that the for statement can iterate over the items of a sequence (a list of names in the case below).

```
for aname in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:
    invitation = "Hi " + aname + ".  Please come to my party on Saturday!"
    print(invitation)
```

Recall that the loop variable takes on each value in the sequence of names. The body is performed once for each name. The same was true for the sequence of integers created by the range function.

```
for avalue in range(10):
    print(avalue)
```

Since a string is simply a sequence of characters, the for loop iterates over each character automatically.

```
for achar in "Go Spot Go":
    print(achar)
```

The loop variable achar is automatically reassigned each character in the string "Go Spot Go". We will refer to this type of sequence iteration as iteration by item. Note that it is only possible to process the characters one at a time from left to right.

Check your understanding

strings-10-4: How many times is the word HELLO printed by the following statements?

```
s = "python rocks"
for ch in s:
    print("HELLO")
```

Ans - Yes, there are 12 characters, including the blank.

strings-10-5: How many times is the word HELLO printed by the following statements?

```
s = "python rocks"
for ch in s[3:8]:
    print("HELLO")
```

Ans - Yes, The blank is part of the sequence returned by slice

5.3. STRING SLICES

Python slice string syntax is:

str_object[start_pos:end_pos:step]

The slicing starts with the start_pos index (included) and ends at end_pos index (excluded). The step parameter is used to specify the steps to take from start to end index.

Python String slicing always follows this rule: s[:i] + s[i:] == s for any index 'i'.

All these parameters are optional – start_pos default value is 0, the end_pos default value is the length of string and step default value is 1.

Let's look at some simple examples of string slice function to create substring.

s = 'HelloWorld'

print(s[:])

print(s[::])

Output:

HelloWorld

HelloWorld

Note that since none of the slicing parameters were provided, the substring is equal to the original string.

Let's look at some more examples of slicing a string.

```
s = 'HelloWorld'
first_five_chars = s[:5]
print(first_five_chars)
third_to_fifth_chars = s[2:5]
print(third_to_fifth_chars)
```

Output:

Hello

Llo

Note that index value starts from 0, so start_pos 2 refers to the third character in the string.

Reverse a String using Slicing

We can reverse a string using slicing by providing the step value as -1.

```
s = 'HelloWorld'
reverse_str = s[::-1]
print(reverse_str)
```

Output:

dlroWolleH

Let's look at some other examples of using steps and negative index values.

```
s1 = s[2:8:2]
print(s1)
```

Output:

loo

Here the substring contains characters from indexes 2,4 and 6.

```
s1 = s[8:1:-1]
print(s1)
```

Output:

lroWoll

5.4 STRINGS ARE IMMUTABLE

The standard wisdom is that Python strings are immutable. You can't change a string's value, only the reference to the string. Like so:

x = "hello"

x = "goodbye" # New string!

Which implies that each time you make a change to a string variable, you are actually producing a brand new string. Because of this, tutorials out there warn you to avoid string concatenation inside a loop and advise using join instead for performance reasons. Even the official documentation says so!

This is wrong. Sort of.

There is a common case for when strings in Python are actually mutable. I will show you an example by inspecting the string object's unique ID using the builtin id() function, which is just the memory address. The number is different for each object. (Objects can be shared though, such as with interning.)

An unchanging article alludes to the item which is once made can't change its worth all its lifetime. Attempt to execute the accompanying code:

name_1 = "Aarun"

name_1[0] = 'T'

You will get a mistake message when you need to change the substance of the string.

Traceback (latest call last):

Record   "/home/ca508dc8fa5ad71190ca982b0e3493a8.py",   line   2,   in <module>

name_1[0] = 'T'

TypeError: 'str' object doesn't uphold thing task

Arrangement

One potential arrangement is to make another string object with vital alterations:

name_1 = "Aarun"

name_2 = "T" + name_1[1:]

print("name_1 = ", name_1, "and name_2 = ", name_2)

name_1 = Aarun and name_2 = Tarun

To watch that they are various strings, check with the id() work:

name_1 = "Aarun"

name_2 = "T" + name_1[1:]

print("id of name_1 = ", id(name_1))

print("id of name_2 = ", id(name_2))

Output:

id of name_1 = 2342565667256

id of name_2 = 2342565669888

To see more about the idea of string permanence, think about the accompanying code:

name_1 = "Aarun"

name_2 = "Aarun"

print("id of name_1 = ", id(name_1))

print("id of name_2 = ", id(name_2))

Output:

id of name_1 = 2342565667256

id of name_1 with new value = 2342565668656

5.5 SEARCHING

Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for.This is known as Linear search. It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

Linear Search:

In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structure.

Example

```
def linear_search(values, search_for):
    search_at = 0
    search_res = False
```

```
# Match the value with each data element
    while search_at < len(values) and search_res is False:
        if values[search_at] == search_for:
            search_res = True
        else:
            search_at = search_at + 1
    return search_res
l = [64, 34, 25, 12, 22, 11, 90]
print(linear_search(l, 12))
print(linear_search(l, 91))
```

Output:

When the above code is executed, it produces the following result −

True

False

Interpolation Search:

This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed. Initially, the probe position is the position of the middle most item of the collection. If a match occurs, then the index of the item is returned. If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

Example

There is a specific formula to calculate the middle position which is indicated in the program below −

```
def intpolsearch(values,x ):
    idx0 = 0
    idxn = (len(values) - 1)
    while idx0 <= idxn and x >= values[idx0] and x <= values[idxn]:
# Find the mid point
        mid = idx0 +\
    int((((float(idxn - idx0)/( values[idxn] - values[idx0]))
        * ( x - values[idx0])))
# Compare the value at mid point with search value
    if values[mid] == x:
```

```
        return "Found "+str(x)+" at index "+str(mid)
    if values[mid] < x:
        idx0 = mid + 1
    return "Searched element not in the list"
l = [2, 6, 11, 19, 27, 31, 45, 121]
print(intpolsearch(l, 2))
```

Output:

Found 2 at index 0

5.6 LOOPING AND COUNTING

The following program counts the number of times the letter "r" appears in a string:

```
word = 'raspberry'
        count = 0
        for letter in word:
            if letter == 'r':
                count = count + 1
        print(count)
```

This program demonstrates another pattern of computation called a counter. The variable count is initialized to 0 and then incremented each time an "r" is found. When the loop exits, count contains the result: the total number of r's.

```
s = "peanut butter"
count = 0
for char in s:
    if char == "t":
        count = count + 1
print(count)
```

Output:

The letter t appears 3 times in "peanut butter".

5.7 STRING METHODS

Python has a set of built-in methods that you can use on strings.

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string

swapcase()	Swaps cases, lower case becomes upper case and vice versa

Note: All string methods returns new values. They do not change the original string.

5.8. THE IN OPERATOR

Not let us take an example to get a better understanding of the in operator working.

x in y

Here "x" is the element and "y" is the sequence where membership is being checked.

Let's implement a simple Python code to demonstrate the use of the in operator and how the outputs would look like.

vowels = ['A', 'E', 'I', 'O', 'U']

ch = input('Please Enter a Capital Letter:\n')

if ch in vowels:

    print('You entered a vowel character')

else:

    print('You entered a consonants character')

We can use the "in" operator with Strings and Tuples too because they are sequences.

>>> name='JournalDev'

>>> 'D' in name

True

>>> 'x' in name

False

>>> primes=(2,3,5,7,11)

>>> 3 in primes

True

>>> 6 in primes

False

Can we use Python "in" Operator with a Dictionary?

Let's see what happens when we use "in" operator with a dictionary.

dict1 = {"name": "Pankaj", "id": 1}

print("name" in dict1) # True

print("Pankaj" in dict1) # False

It looks like the Python "in" operator looks for the element in the dictionary keys.

5.9 STRING COMPARISON

The following are the ways to compare two string in Python:

1. By using == (equal to) operator

2. By using != (not equal to) operator

3. By using sorted() method

4. By using is operator

5. By using Comparison operators

1. Comparing two strings using == (equal to) operator

str1 = input("Enter the first String: ")

str2 = input("Enter the second String: ")

if str1 == str2:

    print ("First and second strings are equal and same")

else:

    print ("First and second strings are not same")

Output:

Enter the first String: AA

Enter the second String: AA

First and second strings are equal and same

2. Comparing two strings using != (not equal to) operator

str1 = input("Enter the first String: ")

str2 = input("Enter the second String: ")

if str1 != str2:

    print ("First and second strings are not equal.")

else:

    print ("First and second strings are the same.")

Output:

Enter the first String: ab

Enter the second String: ba

First and second strings are not equal.

3. Comparing two strings using the sorted() method:

If we wish to compare two strings and check for their equality even if the order of characters/words is different, then we first need to use sorted() method and then compare two strings.

```
str1 = input("Enter the first String: ")
str2 = input("Enter the second String: ")
if sorted(str1) == sorted(str2):
    print ("First and second strings are equal.")
else:
    print ("First and second strings are not the same.")
```

Output:

Enter the first String: Engineering Discipline

Enter the second String: Discipline Engineering

First and second strings are equal.

4. Comparing two strings using 'is' operator

Python is Operator returns True if two variables refer to the same object instance.

```
str1 = "DEED"
str2 = "DEED"
str3 = ''.join(['D', 'E', 'E', 'D'])
print(str1 is str2)
print("Comparision result = ", str1 is str3)
```

Output:

True

Comparision result = False

In the above example, str1 is str3 returns False because object str3 was created differently.

5. Comparing two strings using comparison operators

```
input = 'Engineering'
print(input < 'Engineering')
print(input > 'Engineering')
print(input <= 'Engineering')
print(input >= 'Engineering')
```

Output:

False

False

True

True

5.10 STRING OPERATIONS

String is an array of bytes that represent the Unicode characters in Python. Python does not support character datatype. A single character also works as a string.Python supports writing the string within a single quote(') and a double quote("").

Example

"Python" or 'Python'

Code

SingleQuotes ='Python in Single Quotes'

DoubleQuotes ="Python in Double Quotes"

print(SingleQuotes)

print(DoubleQuotes)

Output:

Python in Single Quotes

Python in Double Quotes

A single character is simply a string with a length of 1. The square brackets can be used to access the elements from the string.

print()

This function is used for displaying the output or result on the user screen.

Syntax

 print('Text or Result') or print("Text or Result')

 Indexing in String

 It returns a particular character from the given string.

 Syntax

 getChar = a[index]

message = "Hello, Python!"

If I want to fetch the characters of the 7th index from the given string.

Syntax

print(string[index])

 Code

print(message[7])

Output:

P

5.11 SUMMARY

A **Python String** is an array of bytes demonstrating Unicode characters. Since there is no such data type called character data type in python, A single character is a string of length one. String handling is one of those activities in coding that programmers, use all the time. In Python, we have numerous built-in functions in the standard library to assist you manipulate. If the string has length one, then the indices start from 0 for the preliminary character and go to L-1 for the rightmost character. Negative indices may be used to count from the right end, -1(minus one) for the rightmost character through -L for the leftmost character. Strings are immutable, so specific characters could be read, but not set. A substring of 0 or more successive characters of a string may be referred to by specifying a starting index and the index one *earlier* the last character of the substring. If the starting and/or ending index is left out Python uses 0 and the length of the string correspondingly. Python assumes indices that would be beyond an end of the string essentially say the end of the string. String formatting is the way we form instructions so that Python can recognize how to integrate data in the creation of strings. How strings are formatted determines the presentation of this data. The basis of string formatting is its use of the formatting directives.Logical operators return true or false values. Comparison operators (==, !=, <>, >, >=, <, <=) compare two values.Any value in Python equates to a Boolean true or false. A false can be equal to none, a zero of numeric type (0, 0l, 0.0), an empty sequence ('', (), []), or an empty dictionary ({}). All other values are reflected true. Sequences, tuples, lists, and strings can be added and/or multiplied by a numeric type with the addition and multiplication operators (+, *), correspondingly. Strings should be formatted with tuples and dictionaries using the format directives %i, %d, %f, and %e. Formatting flags should be used with these directives.

5.12 QUESTIONS

1. Find the index of the first occurrence of a substring in a string.

2. How would you check if each word in a string begins with a capital letter?

3. Write a Python program to calculate the length of a string

4. Write a Python program to get a string from a given string where all occurrences of its first char have been changed to '$', except the first char itself.

Sample String given: 'reboot'

1. Write a Python program to change a given string to a new string where the first and last chars have been exchanged.

2. Write a Python program to count the occurrences of each word in a given sentence

3. Write Python program to Check all strings are mutually disjoint

4. Write a program to find the first and the last occurence of the letter 'o' and character ',' in "Good, Morning".

5. Write a program to check if the word 'open' is present in the "This is open source software".

6. Write a program to check if the letter 'e' is present in the word 'Welcome'.

5.13 REFERENCES

1. https://developers.google.com/edu/python/strings
2. https://docs.python.org/3/library/string.html
3. https://www.programiz.com/python-programming/string
4. http://ww2.cs.fsu.edu/~nienaber/teaching/python/lectures/sequence-string.html
5. https://www.tutorialsteacher.com/python/python-string
6. https://techvidvan.com/tutorials/python-strings/
7. http://anh.cs.luc.edu/handsonPythonTutorial/objectsummary.html
8. https://docs.python.org/3/library/stdtypes.html
9. https://www.programiz.com/python-programming/methods/string
10. https://www.w3schools.com/python/python_ref_string.asp
11. https://www.tutorialspoint.com/python_text_processing/python_string_immutability.htm
12. https://web.eecs.utk.edu/~azh/blog/pythonstringsaremutable.html

*****

UNIT III

6

LIST

6.1 OBJECTIVES

1. To learn how python uses lists to store various data values.

2. To study usage of the list index to remove, update and add items from python list.

3. To understand the abstract data types queue, stack and list.

4. To understand the implementations of basic linear data structures.

5. To study the implementation of the abstract data type list as a linked list using the node.

The list is most likely versatile data type available in the Python which is can be written as a list of comma-separated values between square brackets. The Important thing about a list is that items in the list need not be of the same type.

Creating a list is as very simple as putting up different comma-separated by values between square brackets. For example –

list1 = ['jack', 'nick', 1997, 5564];
list2 = [1, 2, 3, 4];
list3 = ["a", "b", "c", "d"];

75

Similar to a string indices, A list indices start from 0, and lists can be sliced, concatenated.

6.2 VALUES AND ACCESSING ELEMENTS

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

list1 =['jack','nick',1997,5564];

list2 =[1,2,3,4,5,6,7];

print"list1[0]: ", list1[0]

print"list2[1:5]: ", list2[1:5]

output –

list1[0]: jack

list2[1:5]: [2, 3, 4, 5]

6.3 LISTS ARE MUTABLE:

lists is mutable. This means we can change a item in a list by accessing it directly as part of the assignment pf statement. Using the indexing operator (square brackets) on the left of side an assignment, we can update one of the list item.

Example:

color = ["red", "white", "black"]
print(color)

color [0] = "orange"
color [-1] = "green"
print(color)

 Output:.

["red", "white", "black"]

["orange ", "white", "green "]

6.4 TRAVERSING A LIST

The mostly common way to traverse the elements of list with for loop. The syntax is the same as for strings:

color = ["red", "white", "blue", "green"]

76

for color in cheeses:

  print(color)

This works well if you only need to read the element of list. But if you want to do write or update the element, you need the indices. A common way to do that is to combine the functions range and len:

for i in range(len(number)):

      number [i] = number[i] * 2

output −
red
white
blue
green

6.5 DELETING ELEMENTS FROM LIST

To delete a list element, you can use either the **del** statement, **del** removes the item at a specific index, if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

list1 = ['red', 'green', 5681, 2000,];

print(list1)

del list[2]

print("After deleting element at index 2 :");

print(list1)

output −.

['red', 'green', 5681, 2000,]

After deleting element at index 2 :

['red', 'green' , 2000,]

Example2:

numbers = [50, 60, 70, 80]
del numbers[1:2]
print(numbers)

Output:
[50, 70, 80]

- One other method from removing elements from a list is to take a slice of the list, which excludes the index or indexes of the item or items you are trying to remove. For instance, to remove the first two items of a list, you can do
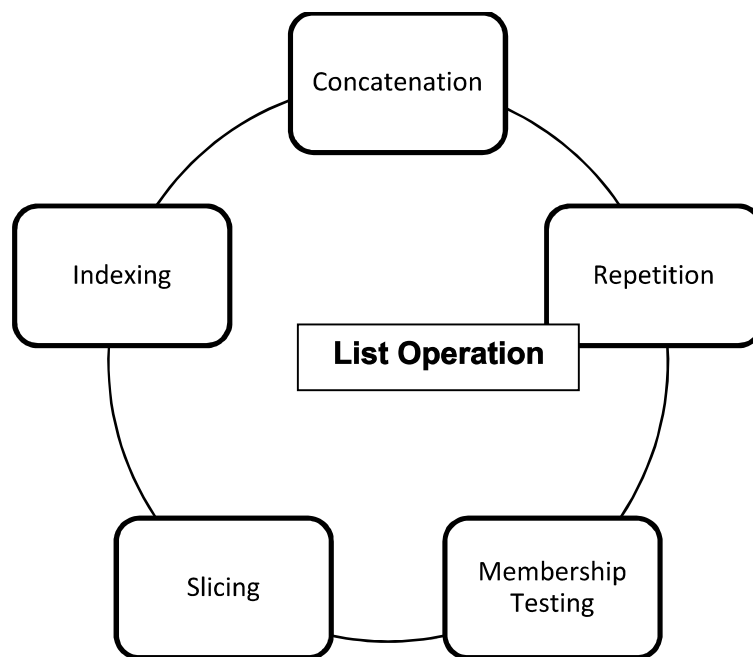
list = list[2:]

6.6 BUILT-IN LIST OPERATORS

In this lesson we will learn about built-in list operators:

1. Concatenation:

Concatenation or joining is a process in which multiple sequence / lists can be combined together. '+' is a symbol concatenation operator.

Example:

```
list1 = [10, 20, 30]

list2 = [40, 50, 60]

print(list1 + list2)
```

Output:

[10, 20, 30, 40, 50, 60]

2. Repetition / Replication / Multiply:

This operator replication the list for a specified number of times and creates a new list. '*' is a symbol of repletion operator.

Example:

list1 = [10, 20, 30]

list2 = [40, 50, 60]

print(list1 * list2

Output:

[10, 20, 30,10, 20, 30,10, 20, 30,])

3. Membership Operator:

This operator used to check or test whether a particular element or item is a member of any list or not. **'in'** And **'not in'** are the operators for membership operator.

Example:

list1 = [10, 20, 30]

list2 = [40, 50, 60]

print(50 **in** list1) #false

print(20 **in** list1) #true

print(50 **not in** list1) #true

print(20 **not in** list1) #false

Output:

False

True

True

False

4. Indexing:

Indexing is nothing but there is an index value for each tem present in the sequence or list.
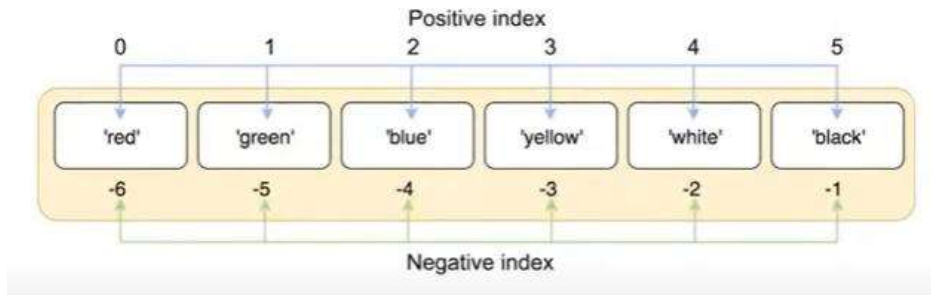
Example:

list1 = [10, 20, 30, 40, 50, 60, 70]

print(list1[4])

Output:

50

5. Slicing operator:

This operator used to slice a particular range of a list or a sequence. Slice is used to retrieve a subset of values.

Syntax: list1[start:stop:step]

Example:

list1 = [10, 20, 30,

print(list1[3:7])

Output:

[40, 50, 60, 70]

6.7 CONCATENATION

In this we will learn different methods to concatenate lists in python. Python list server the purpose of storing homogenous elements and perform manipulations on the same.

In general, Concatenation is the process of joining the elements of a particular data-structure in an end-to-end manner.

The following are the 4 ways to concatenate lists in python.

- **Concatenation (+) operator:**

The '+' operator can be used to concatenate two lists. It appends one list at the end of the other list and results in a new list as output.

Example:

list1 =[10, 11, 12, 13, 14]
list2 =[20, 30, 42]

result =list1 +list2

print(str(result))

Output:

[10, 11, 12, 13, 14, 20, 30, 42]

- **Naive method:**

In the Naive method, a for loop is used to be traverse the second list. After this, the elements from the second list will get appended to the first list. The first list of results out to be the concatenation of the first and the second list.

Example:

list1 = [10, 11, 12, 13, 14]

list2 = [20, 30, 42]

print("Before Concatenation:" + str(list1))

for x in list2 :

　list1.append(x)

 print ("After Concatenation:" + str(list1))

Output:

Before Concatenation:

[10, 11, 12, 13, 14]

After Concatenation:

[10, 11, 12, 13, 14, 20, 30, 42]

List comprehension:

Python list comprehension is an the alternative method to concatenate two lists in python. List comprehension is basically the process of building / generating a list of elements based on an existing list.

It uses the for loop to process and traverses a list in the element-wise fashion. The below inline is for-loop is equivalent to a nested for loop.

Example:

list1 = [10, 11, 12, 13, 14]

list2 = [20, 30, 42]

result = [j for i in [list1, list2] for j in i]

print ("Concatenated List:\n"+ str(result))

Output:

Concatenated list:

 [10, 11, 12, 13, 14, 20, 30, 42]

Extend() method:

Python **extend()** method can be used to concatenate two lists in python. The **extend()** function does iterate over the password parameter and add the item to the list, extending the list in a linear fashion.

Syntax:
                      list.extend(iterable)

Example:

list1 = [10, 11, 12, 13, 14]

list2 = [20, 30, 42]

print("list1 before concatenation:\n" + str(list1))

list1.extend(list2)

print ("Concatenated list i.e ,ist1 after concatenation:\n"+ str(list1))

All the elements of the list2 get appended to list1 and thus the list1 gets updated and results as output.

Output:

list1 before concatenation:

[10, 11, 12, 13, 14]

Concatenated list i.e ,ist1 after concatenation:

[10, 11, 12, 13, 14, 20, 30, 42]

- **'*' operator:**

Python's '*' operator can be used for easily concatenate the two lists in Python.

The '*' operator in Python basically unpacks the collection of items at the index arguments.

For example: Consider a list list = [1, 2, 3, 4].

The statement *list would replace by the list with its elements on the index positions. So, it unpacks the items of the lists.

Example:

```
list1 = [10, 11, 12, 13, 14]

list2 = [20, 30, 42]

res = [*list1, *list2]

print ("Concatenated list:\n " + str(res))
```

Output:

In the above snippet of code, the statement res = [*list1, *list2] replaces the list1 and list2 with the items in the given order i.e. elements of list1 after elements of list2. This performs concatenation and results in the below output.

Output:

Concatenated list:

[10, 11, 12, 13, 14, 20, 30, 42]

- **Itertools.chain() method:**

Python itertools modules' itertools.chain() function can also be used to concatenate lists in Python.

The itertools.chain() function accepts different iterables such as lists, string, tuples, etc as parameters and gives a sequence of them as output.

It results out to be a linear sequence. The data type of the elements doesn't affect the functioning of the chain() method.

For example: The statement itertools.chain([1, 2], ['John', 'Bunny']) would produce the following output: 1 2 John Bunny

Example:

```
import itertools

list1 = [10, 11, 12, 13, 14]
```

```
list2 = [20, 30, 42]

res = list(itertools.chain(list1, list2))

print ("Concatenated list:\n " + str(res))
```

Output:

Concatenated list:

[10, 11, 12, 13, 14, 20, 30, 42]

6.8 REPETITION

Now, we are accustomed to using the '*' symbol to represent the multiplication, but when the operand on the left of side of the '*' is a tuple, it becomes the repetition operator. And The repetition of the operator it will makes the multiple copies of a tuple and joins them all together. Tuples can be created using the repetition operator, *.

Example:

```
number = (0,) * 5  # we use the comma to denote that this is a single valued tuple
and not an '#'expression
```

Output

```
print numbers

(0, 0, 0, 0, 0)
```

[0] is a tuple with one element, 0. Now repetition of the operator it will makes 5 copies of this tuple and joins them all together into a single tuple. Another example using multiple elements in the tuple.

Example:

```
numbers = (0, 1, 2) * 3
```

Output

```
print numbers

(0, 1, 2, 0, 1, 2, 0, 1, 2)
```

6.9 IN OPERATOR

Python's in operator lets you loop through all the members of a collection (such as a list or a tuple) and check if there's a member in the list that's equal to the given item.

Example:

my_list = [5, 1, 8, 3, 7]

print(8 in my_list)

print(0 in my_list)

Output:

True
False

Note: Note that in operator against dictionary checks for the presence of key.

Example:
my_dict = {'name': 'TutorialsPoint', 'time': '15 years', 'location': 'India'}
print('name' in my_dict)

Output:

This will give the output −

True

6.10 BUILT-IN LIST FUNCTIONS AND METHODS

1. Built-in function:

Sr. No.	Function with Description
1	cmp(list1, list2) Compares elements of both lists.
2	len(list) Gives the total length of the list.
3	max(list) Returns item from the list with max value.
4	min(list) Returns item from the list with min value.
5	list(seq) Converts a tuple into list.

2. Built-in methods:

Sr. No.	Methods with Description
1	list.append(obj) Appends object obj to list
2	list.count(obj) Returns count of how many times obj occurs in list
3	list.extend(seq) Appends the contents of seq to list
4	list.index(obj) Returns the lowest index in list that obj appears
5	list.insert(index, obj) Inserts object obj into list at offset index
6	list.pop(obj=list[-1]) Removes and returns last object or obj from list
7	list.remove(obj) Removes object obj from list
8	list.reverse() Reverses objects of list in place
9	list.sort([func]) Sorts objects of list, use compare func if given

6.11 SUMMARY

Python lists are commanding data structures, and list understandings are one of the furthermost convenient and brief ways to create lists. In this chapter, we have given some examples of how you can use list comprehensions to be more easy-to-read and simplify your code. The list is the common multipurpose data type available in Python. A list is an ordered collection of items. When it comes to creating lists, Python list are more compressed and faster than loops and other functions used such as, map(), filter(), and reduce().Every Python list can be rewritten in for loops, but not every complex for loop can be rewritten in Python list understanding. Writing very long list in one line should be avoided so as to keep the code user-friendly. So list looks just like dynamic sized arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain Datatype's like Integers, Strings, and Objects etc. List loads all the elements into memory at one time, when the list is too long, it will reside in too much memory resources, and we usually only need to use a few elements. A list is a data-structure that can be used to store multiple data at once. The list will be ordered and there will be a definite count of it. The elements are indexed allowing to a sequence and the indexing is done with 0 as the first index. Each element will have a discrete place in the sequence and if the same value arises multiple times in the sequence.

6.12 QUESTIONS

1. Explain List parameters with an example.

2. Write a program in Python to delete first and last elements from a list

3. Write a Python program to print the numbers of a specified list after removing even numbers from it.

4. Write a python program using list looping

5. Write a Python program to check a list is empty or not

6. Write a Python program to multiplies all the items in a list

7. Write a Python program to remove duplicates from a list

8. Write a Python program to append a list to the second list

9. Write a Python program to find the second smallest number in a list.

10. Write a Python program to find common items from two lists

6.13 REFERENCES

1. https://python.plainenglish.io/python-list-operation-summary-262f40a863c8?gi=a4f7ce4740e9

2. https://howchoo.com/python/how-to-use-list-comprehension-in-python

3. https://intellipaat.com/blog/tutorial/python-tutorial/python-list-comprehension/

4. https://programmer.ink/think/summary-of-python-list-method.html

5. https://www.geeksforgeeks.org/python-list/

6. https://enricbaltasar.com/python-summary-methods-lists/

7. https://developpaper.com/super-summary-learn-python-lists-just-this-article-is-enough/

8. https://www.programiz.com/python-programming/methods/list

9. https://www.hackerearth.com/practice/python/working-with-data/lists/tutorial/

10. https://data-flair.training/blogs/r-list-tutorial/

*****

<div align="right">

7

</div>

TUPLES AND DICTIONARIES

Unit Structure

7.1 OBJECTIVES

1. To understand when to use a dictionary.
2. To study how a dictionary allows us to characterize attributes with keys and values
3. To learn how to read a value from a dictionary
4. To study how in python to assign a key-value pair to a dictionary
5. To understand how tuples returns values

7.2 TUPLES

A tuple in the Python is similar to the list. The difference between the two is that we cannot change the element of the tuple once it is assigned to whereas we can change the elements of a list

The reasons for having immutable types apply to tuples: copy efficiency: rather than copying an immutable object, you can alias it (bind a variable to a reference) ... interning: you need to store at most of one copy of any immutable value. There's no any need to synchronize access to immutable objects in concurrent code.

Creating a Tuple:

A tuple is created by the placing all the elements inside parentheses '()', separated by commas. The parentheses are the optional and however, it is a good practice to use them.

A tuple can have any number of the items and they may be a different types (integer, float, list, string, etc.).

Example:

```
# Different types of tuples

# Empty tuple

tuple = ()

print(tuple)

# Tuple having integers

tuple = (1, 2, 3)

print(tuple)

# tuple with mixed datatypes

tuple = (1, "code", 3.4)

print(tuple)

# nested tuple

tuple = ("color ", [6, 4, 2], (1, 2, 3))

print(tuple)
```

Output:

()

(1, 2, 3)

(1, 'code', 3.4)

('color', [6, 4, 2], (1, 2, 3))

A tuple can also be created without using parentheses. This is known as tuple packing.

```
tuple = 3, 2.6, "c

print(tuple)


# tuple unpackir

a, b, c = tuple


print(a)     # 3

print(b)     # 4.6

print(c)     # dog
```

Output:

(3, 2.6, 'color')

3

2.6

color

7.3 ACCESSING VALUES IN TUPLES

There are various ways in which we can access the elements of a tuple.

1. Indexing:

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
# Accessing tuple elements using indexing

tuple = ('a','b','c','d','e','f')

print(tuple[0])   # 'a'

print(tuple[5])   # 'f'

# IndexError: list index out of range

# print(tuple[6])

# Index must be an integer

# TypeError: list indices must be integers, not float

# tuple[2.0]

# nested tuple

tuple = ("color", [6, 4, 2], (1, 2, 3))

# nested index

print(tuple[0][3])      # 'o'

print(tuple[1][1])      # 4
```

Output:
```
a

f

o

4
```

2. Negative Indexing:

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

Example:

```
# Negative indexing for accessing tuple elements

tuple = ('a','b','c','d','e','f')
```

Output: 'f'

print(tuple[-1])

Output: 'a'

print(tuple[-6])

Output:

f

a

3. Slicing:

We can access the range of items from the tuple by using the slicing operator colon:

Example:

Accessing tuple elements using slicing

tuple = ('a','b','c','d','e','f','g','h','i')

elements 2nd to 4th

Output: ('b', 'c', 'd')

print(tuple[1:4])

elements beginning to 2nd

Output: ('a', 'b')

print(tuple[:-7])

elements 8th to end

Output: ('h', 'i')

print(tuple[7:])

elements beginning to end

Output: ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')

print(tuple[:])

Output:
('b', 'c', 'd')

('a', 'b')

('h', 'i')

('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')

7.4 TUPLE ASSIGNMENT

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

Example:

tuple = ['red', 'blue']

x, y = tuple

Output:
x

'red'

y

'blue'

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:

m = ['red', 'blue']

x = m[0]

y = m[1]

Output:
x

'red'

y

'blue'

7.5 TUPLES AS RETURN VALUES

Functions can return tuples as return values. Now we often want to know some batsman's highest and lowest score or we want to know to

find the mean and the standard deviation, or we want to know the year, the month, and the day, or if we're doing some ecological modeling we may want to know the number of rabbits and the number of wolves on an island at a given time. In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.

For example, we could write a function that returns both the area and the circumference of a circle of radius.

Example:

```
def cirlce_info(r):

    #Return (circumference, area) of a circle of radius r

    c = 2 * 3.14159 * r

    a = 3.14159 * r * r

    return (c, a)

print(cirlce_info(10))
```

Output:

(62.8318, 314.159)

7.6 VARIABLE-LENGTH ARGUMENT TUPLES

Functions can take a variable number of arguments. The parameter name that begins with the * gather the argument into the tuple. For example, the print all take any number of the arguments and print them:

```
def printall(*args):

    print args
```

The gather parameter can have any name you like, but args is conventional. Here's how the function works:

The complement of gather is scatter. If you have the sequence of values and you want to pass it to the function as multiple as arguments, you can use the * operator. For example, divmod take exactly two arguments it doesn't work with the tuple:

```
t = (7, 3)

divmod(t)

TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
divmod(*t)
```

(2, 1)

Example -Many of the built-in functions are use variable-length argument tuples. For example, max and min it can take any of number arguments:

max(1,2sum(1,2,3)

TypeError: sum expected at most 2 arguments, got 3,3)

But sum does not.

7.7 BASIC TUPLES OPERATIONS

Now, we will learn the operations that we can perform on tuples in Python.

1. Membership:

We can apply the 'in' and 'not in' operator on the items. This tells us whether they belong to the tuple.

'a' in tuple("string")

Output:

False

'x' not in tuple("string")

Output:
True

2. Concatenation:

Like we've previously discussed on several occasions, concatenation is the act of joining. We can join two tuples using the concatenation operator '+'.

(1,2,3)+(4,5,6)

Output:

(1, 2, 3, 4, 5, 6)

Note: Other arithmetic operations do not apply on a tuple.

3. Logical:

All the logical operators (like >,>=,..) can be applied on a tuple.

(1,2,3)>(4,5,6)

Output:

False

(1,2)==('1','2')

Output:

False

4. Identity:

Remember the 'is' and 'is not' operators we discussed about in our tutorial on Python Operators? Let's try that on tuples.

a=(1,2)

(1,2) is a

Output:

That did not make sense, did it? So what really happened? Now, in Python, two tuples or lists don't have the same identity. In other words, they are two different tuples or lists. As a result, it returns False.

7.8 CONCATENATION

Now, we will learn 2 different ways to concatenate or join tuples in the python language with code example. We will use '+" operator and a built-in sum() function to concatenate or join tuples in python.

- **How to concatenate tuples into single/nested Tuples using sum():**

2.1 Sum() to concatenate tuples into a single tuple:

In our first example, we will use the sum() function to concatenate two tuples and result in a single tuple.Let us jump to example:

tupleint= (1,2,3,4,5,6)

langtuple = ('C#','C++','Python','Go')

#concatenate the tuple

tuples_concatenate = sum((tupleint, langtuple), ())

print('concatenate of tuples \n =',tuples_concatenate)

Output:

concatenate of tuples

= (1, 2, 3, 4, 5, 6, 'C#', 'C++', 'Python', 'Go')

Sum() to concatenate tuple into a nested tuple:

Now let us understand how we can sum tuples to make a nested tuple.In this program we will use two tuples which are nested tuples, please note the ',' at the end of each tuples tupleint and langtuple.

let us understand example:

tupleint= (1,2,3,4,5,6),

langtuple = ('C#','C++','Python','Go'),

#concatenate the tuple

tuples_concatenate = sum((tupleint, langtuple), ())

print('concatenate of tuples \n =',tuples_concatenate)

Output:

concatenate of tuples

= ((1, 2, 3, 4, 5, 6), ('C#', 'C++', 'Python', 'Go'))

Concatenate tuple Using '+' operator:

2.1 '+' operator to concatenate two tuples into a single tuple:

In our first example, we will use the "+" operator to concatenate two tuples and result in a single tuple.In this example we have two tuple tupleint and langtuple,We are the concatenating these tuples into the single tuple as we can see in output.

tupleint= (1,2,3,4,5,6)

langtuple = ('C#','C++','Python','Go')

#concatenate the tuple

tuples_concatenate = tupleint+langtuple

print('concatenate of tuples \n =',tuples_concatenate)

Output:

2.2 '+' operator with a comma(,) to concatenate tuples into nested Tuples:

This example, we have the two tuple tuple int and lang tuple. Now We are using the comma(,) end of the each tuple to the concatenate them

into a nested tuple. We are concatenating these tuples into a nested tuple as we can see in the resulting output.

comma(,) after tuple to concatenate nested tuple

tupleint= (1,2,3,4,5,6),

langtuple = ('C#','C++','Python','Go'),

#concatenate the tuple into nested tuple

tuples_concatenate = tupleint+langtuple

print('concatenate of tuples \n =',tuples_concatenate)

Output:
concatenate of tuples

 = ((1, 2, 3, 4, 5, 6), ('C#', 'C++', 'Python', 'Go'))

7.9 REPETITION

Now, we are going to explain how to use Python tuple repetition operator with basic syntax and many examples for better understanding.

Python tuple repetition operator (*) is used to the repeat a tuple, number of times which is given by the integer value and create a new tuple values.

Syntax:

<tuple_variable_name1> * N
N * <tuple_variable_name1>

Input Parameters:

- tuple_variable_name1 : The tuples that we want to be repeated.
- N : where is the number of times that we want that tuple to be repeated ex: 1,2,3,……..n

Example:

data=(1,2,3,'a','b')

tuple after repetition

print('New tuple:', data* 2)

Output:

New tuple: [1, 2, 3, 'a', 'b', 1, 2, 3, 'a', 'b']

In the above Example, using repetition operator (*), we have repeated 'data' tuple variable 2 times by 'data* 2' in print statement and created new tuple as [1, 2, 3, 'a', 'b', 1, 2, 3, 'a', 'b'].

7.10 IN OPERATOR

The Python in operator lets you loop through all to the members of the collection and check if there's a member in the tuple that's equal to the given item.

Example:
my_tuple = (5, 1, 8, 3, 7)
print(8 in my_tuple)

print(0 in my_tuple)

Output:

True

False

Note that in operator against dictionary checks for the presence of key.

Example:

Output:

True

It can also be used to check the presence of a sequence or substring against string.

Example:

my_str = "This is a sample string"

print("sample" in string)

Output:

True

It can be uses in the many of other places and how it works in the those scenarios of varies a lot. This is the how in works in tuples. It start

99

the comparing references of the objects from the first one till it either finds that the object in the tuple or reaches in the end of the tuple.

7.11 ITERATION

There are many ways to iterate through the tuple object. For statement in Python has a variant which traverses a tuple till it is exhausted. It is equivalent to for each statement in Java. Its syntax is −

for var in tuple:

stmt1

stmt2

Example:

T = (10,20,30,40,50)

for var in T:

print (T.index(var),var)

Output:

0 10

1 20

2 30

3 40

4 50

7.12 BUILT-IN TUPLE FUNCTIONS

Tuples support the following build-in functions:

Comparison:

If the elements are of the same type, python performs the comparison and returns the result. If elements are different types, it checks whether they are numbers.

If numbers, perform comparison.

If either the element is an number, then the other element is a returned.

Otherwise, types are sorted alphabetically.

If we reached to the end of one of the lists, the longer list is a "larger." If both are list are same it returns 0.

tuple1 = ('a', 'b', 'c','d', 'e')

tuple2 = ('1','2','3')

tuple3 = ('a', 'b', 'c', 'd', 'e')

cmp(tuple1, tuple2)

Out: 1

cmp(tuple2, tuple1)

Out: -1

cmp(tuple1, tuple3)

Out: 0

Tuple Length:

len(tuple1)

Out: 5

Max of a tuple:

The function min returns the item from the tuple with the min value:

min(tuple1)

Out: 'a'

min(tuple2)

Out: '1'

Convert a list into tuple:

The built-in function tuple converts a list into a tuple:

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

tuple(list)

Out: (1, 2, 3, 4, 5)

Tuple concatenation:

Use + to concatenate two tuples:

tuple1 + tuple2

Out: ('a', 'b', 'c','d', 'e', '1', '2', '3')

7.13 CREATING A DICTIONARY

Creating a Dictionary:

To create the Python dictionary, we need to pass the sequence of the items inside curly braces {}, and to separate them using a comma (,). Each item has a key and a value expressed as an "key:value" pair.

The values can belong to the any of data type and they can repeat, but the keys are must remain the unique.

The following examples are demonstrate how to create the Python dictionaries:

Creating an empty dictionary:

dict_sample = {}

Creating a dictionary with integer keys:

dict_sample = {1: 'mango', 2: 'pawpaw'}

Creating a dictionary with mixed keys:

dict_sample = {'fruit': 'mango', 1: [4, 6, 8]}

We can also create a dictionary by explicitly calling the Python's dict() method:

dict_sample = dict({1:'mango', 2:'pawpaw'})

A dictionary can also be created from a sequence as shown below:
Dictionaries can also be nested, which means that we can create a dictionary inside another dictionary. For example:

dict_sample = {1: {'student1' : 'Nicholas', 'student2' : 'John', 'student3' : 'Mercy'},
        2: {'course1' : 'Computer Science', 'course2' : 'Mathematics',
        'course3' : 'Accounting'}}

To print the dictionary contents, we can use the Python's print() function and pass the dictionary name as the argument to the function. For example:

```
dict_sample = {
  "Company": "Toyota",
  "model": "Premio",
  "year": 2012
}
print(dict_sample)
```

Output:

{'Company': 'Toyota', 'model': 'Premio', 'year': 2012}

7.14 ACCESSING VALUES IN A DICTIONARY

To access the dictionary items, we need to pass the key inside square brackets []. For example:

```
dict_sample = {

  "Company": "Toyota",

  "model": "Premio",

  "year": 2012

}

x = dict_sample["model"]

print(x)
```

Output:

```
Premio
```

We created a dictionary named dict_sample. A variable named x is then created and its value is set to be the value for the key "model" in the dictionary.

7.15 UPDATING DICTIONARY

After adding a value to a dictionary we can then modify the existing dictionary element. You use the key of the element to change the corresponding value. For example:

```
dict_sample = {
  "Company": "Toyota",
  "model": "Premio",
  "year": 2012
}

dict_sample["year"] = 2014

print(dict_sample)
```

Output:

```
{'year': 2014, 'model': 'Premio', 'Company': 'Toyota'}
```

In this example you can see that we have updated the value for the key "year" from the old value of 2012 to a new value of 2014.

7.16 DELETING ELEMENTS FROM DICTIONARY

The removal of an element from a dictionary can be done in several ways, which we'll discuss one-by-one in this section:

The del keyword can be used to remove the element with the specified key. For example:

```
dict_sample = {

  "Company": "Toyota",

  "model": "Premio",

  "year": 2012

}

del dict_sample["year"]

print(dict_sample)
```

Output:

```
{'Company': 'Toyota', 'model': 'Premio'}
```

We called the del keyword followed by the dictionary name. Inside the square brackets that follow the dictionary name, we passed the key of the element we need to delete from the dictionary, which in this example was "year". The entry for "year" in the dictionary was then deleted.

Another type to delete a key-value pair is to use the pop() method and pass the key of the entry to be deleted as the argument to the function. For example:

Output:

```
dict_sample = {

  "Company": "Toyota",

  "model": "Premio",

  "year": 2012

}

dict_sample.pop("year")

print(dict_sample)
```

We invoked that pop() method by appending it with the dictionary name. And, in this example the entry for "year" in the dictionary will be deleted.

The popitem() method removes the last item of inserted into the dictionary, without needing to specify the key. Take a look at the following example:

```
dict_sample = {

  "Company": "Toyota",

  "model": "Premio",

  "year": 2012

}

dict_sample.popitem()

print(dict_sample)
```

Output:

```
{'Company': 'Toyota', 'model': 'Premio'}
```

The last entry into the dictionary was "year". It has been removed after calling the popitem() function.

But what if you want to delete the entire dictionary? It would be difficult and cumbersome to use one of these methods on every single key. Instead, you can use the del keyword to delete the entire dictionary. For example:

```
dict_sample = {

  "Company": "Toyota",

  "model": "Premio",

  "year": 2012

}

del dict_sample

print(dict_sample)
```

Output:

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

The code returns an error. The reason is we are trying to access the an dictionary which is doesn't exist since it is has been deleted.

However, your use-case may require you to just remove all dictionary elements and be left with an empty dictionary. This can be achieved by calling the clear() function on the dictionary:

105

```
dict_sample = {

  "Company": "Toyota",

  "model": "Premio",

  "year": 2012

}
dict_sample.clear()

print(dict_sample)
```

Output:

{}

The code is returns an empty dictionary since all the dictionary elements have been removed.

7.17 PROPERTIES OF DICTIONARY KEYS

Dictionary values have no restrictions. These can be any of erratically Python object, either they standard objects or user-defined objects. However, similar is not true for the keys.

There are two important points to be remember about the dictionary keys:

(a) More than one of the entry per key not allowed. Which means that no duplicate key isallowed. When the duplicate keys are encountered during the assignment, And, the last assignment wins.

For example:

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

print "dict['Name']: ", dict['Name'];

Output:

dict['Name']: Manni

dict = {['Name']: 'Zara', 'Age': 7};

print "dict['Name']: ", dict['Name'];

(b) Keys must be immutable. Which mean by you can use strings, And the numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple:

106

Output:

Traceback (most recent call last):

File "test.py", line 3, in<module>

dict = {['Name']: 'Zara', 'Age': 7};

TypeError: list objects are unhashable

7.18 OPERATIONS IN DICTIONARY

Below is a list of common dictionary operations:

create an empty dictionary

x = {}

create a three items dictionary

x = {"one":1, "two":2, "three":3}

access an element

x['two']

get a list of all the keys

x.keys()

get a list of all the value

      x.values()

add an entry

      x["four"]=4

change an entry

      x["one"] = "uno"

delete an entry

      del x["four"

make a copy
      y = x.copy()

remove all items

```
        x.clear()
```

number of items

```
        z = len(x)
```

test if has key

```
        z = x.has_key("one")
```

looping over keys

```
        for item in x.keys(): print item
```

looping over values

```
        for item in x.values(): print item
```

using the if statement to get the values

```
        if "one" in x:
            print x['one']

        if "two" not in x:
            print "Two not found"

        if "three" in x:
            del x['three']
```

7.18 BUILT-IN DICTIONARY FUNCTIONS

A function is a procedure that can be applied on a construct to get a value. Furthermore, it doesn't modify the construct. Python gives us a few functions that we can apply on a Python dictionary. Take a look.

1. len():

The len() function returns the length of the dictionary in Python. Every key-value pair adds 1 to the length.

```
    len(dict4)
```

Output
```
    3
```

```
    len({})
```

```
    any({False:False,":"})
```

An empty Python dictionary has a length of 0.

2. any():

Like it is with lists an tuples, the any() function returns True if even one key in a dictionary has a Boolean value of True.

Output:

False

Output:

any({True:False,"":""})

True

3

. all():

Unlike the any() function, all() returns True only if all the keys in the dictionary have a Boolean value of True.

Output:

all({1:2,2:",""":3})

False

4. sorted():

Like it is with lists and tuples, the sorted() function returns a sorted sequence of the keys in the dictionary. The sorting is in ascending order, and doesn't modify the original Python dictionary.

dict4={3:3,1:1,4:4}

But to see its effect, let's first modify dict4.
Now, let's apply the sorted() function on it.

Output:

[1, 3, 4]

As you can see, the original Python dictionary wasn't modified.

dict4

This function returns the keys in a sorted list. To prove this, let's see what the type() function returns.

Output: <class 'list'>

109

This proves that sorted() returns a list.

{3: 3, 1: 1, 4: 4}

dict4

type(sorted(dict4))

7.19 BUILT-IN DICTIONARY METHODS

A method is a set of the instructions to execute on the construct, and it may be modify the construct. To do this, the method must be called on the construct, let's look at the available the methods for dictionaries.

dict4.keys()

Let's use dict4 for this example.

1. keys():

dict_keys([3, 1, 4])

The keys() method returns a list of keys in a Python dictionary.
dict4.values()

Output:

2. values():
Likewise, the values() method returns a list of values in the dictionary.

Output:

dict_values([3, 1, 4])

3. items()
This method returns a list of key-value pairs.

Output:

dict_items([(3, 3), (1, 1), (4, 4)])

7.20 SUMMARY

Tuples: In Python, tuples are structured and accessed based on position. A Tuple is a collection of Python objects separated by commas. In some ways a tuple is similar to a list in terms of indexing, nested objects and repetition but a tuple is absolute unlike lists that are variable. In Python it is an unordered collection of data values, used to store data values like a map, which unlike other data types that hold only single value as an element. Tuples are *absolute* lists. Elements of a list can be modified, but elements in a tuple can only be accessed, not modified. The

name *tuple* does not mean that only two values can be stored in this data structure.

Dictionaries: Dictionaries in Python are structured and accessed using keys and values. Dictionaries are defined in Python with curly braces { } . Commas separate the key-value pairs that make up the dictionary. Dictionaries are made up of key and/or value pairs. In Python, tuples are organized and accessed based on position. The location of a pair of keys and values stored in a Python dictionary is unrelated. Key value is provided in the dictionary to make it more optimized. A Python dictionary is basically a **hash** table. In some languages, they might be mentioned to an associative **arrays**. They are indexed with keys, which can be any absolute type.

7.21 QUESTIONS

1. Let list = ['a', 'b', 'c','d', 'e', 'f']. Find a) list[1:3] b) t[:4] c) t[3:]
2. State the difference between lists and dictionary
3. What is the benefit of using tuple assignment in Python?
4. Define dictionary with an example
5. Write a Python program to swap two variables
6. Define Tuple and show it is immutable with an example
7. Create tuple with single element
8. How can you access elements from the dictionary
9. Write a Python program to create a tuple with different data types.
10. Write a Python program to unpack a tuple in several variables

7.22 REFERENCES

1. https://www.geeksforgeeks.org/differences-and-applications-of-list-tuple-set-and-dictionary-in-python/
2. https://problemsolvingwithpython.com/04-Data-Types-and-Variables/04.05-Dictionaries-and-Tuples/
3. https://problemsolvingwithpython.com/04-Data-Types-and-Variables/04.05-Dictionaries-and-Tuples/
4. https://ncert.nic.in/textbook/pdf/kecs110.pdf
5. https://python101.pythonlibrary.org/chapter3_lists_dicts.html
6. https://www.programmersought.com/article/26815189338/
7. https://www.javatpoint.com/python-tuples
8. https://cloudxlab.com/assessment/displayslide/873/python-dictionaries-and-tuples
9. https://medium.com/@aitarurachel/data-structures-with-lists-tuples-dictionaries-and-sets-in-python-612245a712af
10. https://www.w3schools.com/python/python_tuples.asp

*****

<div align="right">**8**</div>

FILES AND EXCEPTIONS

Unit Structure

8.1 OBJECTIVE

1. To understand how python will raise an exception.
2. To create program to catch an exception using a try/except block.
3. To study the Python errors and exceptions.
4. To study creation and use of read and write commands for files.in python
5. To understand how to open, write and close files in python

8.2 TEXT FILES

Now we will learn about various ways to read text files in Python.

The following shows how to read all texts from the readme.txt file into a string:

```
with open('readme.txt') as f:
    lines = f.readlines()
```

Steps for reading a text file in Python:

To read the text file in the Python, you have to follow these steps:

Firstly, you have to open the text file for reading by using the open() method.

Second, you have to read the text from the text file using the file read(), readline(), or readlines() method of the file object.

Third, you have to close the file using the file close() method.

1) open() function
The open() function has many parameters but you'll be focusing on the first two.

open(path_to_file, mode)

The path to the file parameter is specifies the path to the text file.

If the file is in the same folder as is program, you have just need to specify the file name. Otherwise, you have need to specify the path to the file.

Specify the path to the file, you have to use the forward-slash ('/') even if you are working in Windows.

Example, if the file is in the readme.txt stored in the sample folder as the program, you have need to specify the path to the file as c:/sample/readme.txt

A mode is in the optional parameter. This is the string that is specifies the mode in which you want to open the file.

The following table shows available modes for opening a text file:

Mode	Description
'r'	Open for text file for reading text
'w'	Open a text file for writing text
'a'	Open a text file for appending text

For example, to open a file whose name is the-zen-of-python.txt stored in the same folder as the program, you use the following code:

```
f = open('the-zen-of-python.txt','r')
```

The open() function returns a file object which you will use to read text from a text file.

2) Reading text methods:

The file object provides you with three methods for reading text from a text file:

read() – read all text from a file into a string. This method is useful if you have a small file and you want to manipulate the whole text of that file.

readline() – read the text file line by line and return all the lines as strings.

readlines() – read all the lines of the text file and return them as a list of strings.

3) close() method:

The file that you open will remain open until you close it using the close() method.

It's important to close the file that is no longer in use. If you don't close the file, the program may crash or the file would be corrupted.

The following shows how to call the close() method to close the file:

```
f.close()
```

To close the file automatically without calling the close() method, you use the with statement like this:

```
with open(path_to_file) as f:
   contents = f.readlines()
```

In practice, you'll use the with statement to close the file automatically.

Reading a text file examples:

We'll use the-zen-of-python.txt file for the demonstration.

The following example illustrates how to use the read() method to read all the contents of the the-zen-of-python.txt file into a string:

```
with open('the-zen-of-python.txt') as f:
   contents = f.read()
   print(contents)
```

Output:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

...

The following example uses the readlines() method to read the text file and returns the file contents as a list of strings:

lines = []

with open('the-zen-of-python.txt') as f:

    lines = f.readlines()

count = 0

for line in lines:

    count += 1

    print(f'line {count}: {line}')

Output:

line 1: Beautiful is better than ugly.

line 2: Explicit is better than implicit.

line 3: Simple is better than complex.

...

8.3 THE FILE OBJECT ATTRIBUTES

Once a file is opened and you have one file object, you can get various information related to that file. Here is a list of all attributes related to file object:

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Example

Open a file

fo = open("foo.txt", "wb")

print "Name of the file: ", fo.name

print "Closed or not : ", fo.closed

print "Opening mode : ", fo.mode

print "Softspace flag : ", fo.softspace

115

This produces the following result:

Name of the file: foo.txt

Closed or not : False

Opening mode : wb

Softspace flag : 0

8.4 DIRECTORIES

In this Python Directory tutorial, we will import the OS module to be able to access the methods we will apply.

import   os

How to Get Current Python Directory?

To find out which directory in python you are currently in, use the getcwd() method.

os.getcwd()

Output:

'C:\\Users\\lifei\\AppData\\Local\\Programs\\Python\\Python36-32'

Cwd is for current working directory in python. This returns the path of the current python directory as a string in Python.

To get it as a bytes object, we use the method getcwdb().

os.getcwdb()

Output:

b'C:\\Users\\lifei\\AppData\\Local\\Programs\\Python\\Python36-32'

Here, we get two backslashes instead of one. This is because the first one is to escape the second one since this is a string object.

type(os.getcwd())

<class 'str'>

To render it properly, use the Python method with the print statement.

print(os.getcwd())

Output:

Changing Current Python Directory
To change our current working directories in python, we use the chdir() method.

This takes one argument- the path to the directory to which to change.

Output:

'unicodeescape' code can't decode bytes in position 2-3: truncated \UXXXXXXXX escape

But remember that when using backward slashes, it is recommended to escape the backward slashes to avoid a problem.

Output:

How to Create Python Directory?

We can also create new python directories with the mkdir() method. It takes one argument, that is, the path of the new python directory to create.

os.mkdir('Christmas Photos')

 os.listdir()

Output:

['Adobe Photoshop CS2.lnk', 'Atom.lnk', 'Burn Book.txt', 'Christmas Photos', 'desktop.ini', 'Documents', 'Eclipse Cpp Oxygen.lnk', 'Eclipse Java Oxygen.lnk', 'Eclipse Jee Oxygen.lnk', 'For the book.txt', 'Items for trip.txt', 'Papers', 'Remember to remember.txt', 'Sweet anticipation.png', 'Today.txt', 'topics.txt', 'unnamed.jpg']

How to Rename Python Directory?:

To rename directories in python, we use the rename() method. It takes two arguments- the python directory to rename, and the new name for it.

os.rename('Christmas Photos','Christmas 2017')

 os.listdir()

Output:
['Adobe Photoshop CS2.lnk', 'Atom.lnk', 'Burn Book.txt', 'Christmas 2017', 'desktop.ini', 'Documents', 'Eclipse Cpp Oxygen.lnk', 'Eclipse

117

Java Oxygen.lnk', 'Eclipse Jee Oxygen.lnk', 'For the book.txt', 'Items for trip.txt', 'Papers', 'Remember to remember.txt', 'Sweet anticipation.png', 'Today.txt', 'topics.txt', 'unnamed.jpg']

How to Remove Python Directory?

We made a file named 'Readme.txt' inside our folder Christmas 2017. To delete this file, we use the method remove().

os.chdir('C:\\Users\\lifei\\Desktop\\Christmas 2017')

os.listdir()

Output:

['Readme.txt']

8.5 BUILT-IN EXCEPTIONS

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur. We can view all the built-in exceptions using the built-in local() function as

print(dir(locals()['__builtins__']))

 follows:

locals()['__builtins__'] will return a module of built-in exceptions, functions, and attributes. dir allows us to list these attributes as strings. Some of the common built-in exceptions in Python programming along with the error that cause them are listed below:

Exception	Cause of Error
AssertionError fails.	Raised when an assert statement
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() function hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or Delete).

MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets an argument of correct type but improper value
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.

8.6 HANDLING EXCEPTIONS

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax:
Here is simple syntax of try....except...else blocks:

try:

  You do your operations here;

  ....

except ExceptionI:

  If there is ExceptionI, then execute this block.

except ExceptionII:

  If there is ExceptionII, then execute this block.

  ....

else:

  If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax −

    A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

    You can also provide a generic except clause, which handles any exception.

    After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

    The else-block is a good place for code that does not need the try: block's protection.

Example:
This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all:

try:

  fh = open("testfile", "w")

  fh.write("This is my test file for exception handling!!")

except IOError:

  print "Error: can\'t find file or read data"

else:

```
  print "Written  content  in  the  file  successfully"

  fh.close()
```

This produces the following result:

Written content in the file successfully

8.7 EXCEPTION WITH ARGUMENTS

Why use Argument in Exceptions?

Using arguments for Exceptions in Python is useful for the following reasons:

It can be used to gain additional information about the error encountered.

As contents of an Argument can vary depending upon different types of Exceptions in Python, Variables can be supplied to the Exceptions to capture the essence of the encountered errors. Same error can occur of different causes, Arguments helps us identify the specific cause for an error using the except clause.

It can also be used to trap multiple exceptions, by using a variable to follow the tuple of Exceptions.

Arguments in Buil-in Exceptions:

The below codes demonstrates use of Argument with Built-in Exceptions:

Example 1:

```
try:

    b = float(100 + 50 / 0)

except Exception as Argument:

    print( 'This is the Argument\n', Argument)
```

Output:

This is the Argument

 division by zero

Arguments in User-defined Exceptions:

The below codes demonstrates use of Argument with User-defined Exceptions:

Example 1:

```
# create user-defined exception

# derived from super class Exception

class MyError(Exception):

        # Constructor or Initializer

        def __init__(self, value):

                self.value = value

        # __str__ is to print() the value

        def __str__(self):

                return(repr(self.value))

try:

        raise(MyError("Some Error Data"))

# Value of Exception is stored in error

except MyError as Argument:

        print('This is the Argument\n', Argument)
```

Output:

'This is the Argument
'Some Error data'

8.8 USER-DEFINED EXCEPTIONS

Creating User-defined Exception

Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in "Error" similar to naming of the standard exceptions in python. For example:

```
# A python program to create user-defined exception

# class MyError is derived from super class Exception

class MyError(Exception):

        # Constructor or Initializer

        def __init__(self, value):
```

```
            self.value = value

        # __str__ is to print() the value

        def __str__(self):

            return(repr(self.value))

try:

    raise(MyError(3*2))

# Value of Exception is stored in error

except MyError as error:

    print('A New Exception occured: ',error.value)
```

Ouput:

('A New Exception occured: ', 6)

8.9 SUMMARY

Files: Python supports file handling and allows users to handle files for example, to read and write files, along with many other file handling options, to operate on files. The concept of file handling has justified by various other languages, but the implementation is either difficult. Python delights file differently as text or binary and this is significant. Each line of code includes a sequence of characters and they form text file. Each line of a file is ended with a special character like comma {,}. It ends the current line and expresses the interpreter a new one has started. In Python a file is a contiguous set of bytes used to store data. This data is organized in a precise format and can be anything as simple as a text file. In the end, these byte files are then translated into binary 1 and 0 for simple for processing. In Python, a file operation takes place in the order like Open a file then Read or write and finally close the file.

Exceptions: Python provides two important features to handle any unexpected error in Python programs and to add debugging capabilities in them .In Python, all exceptions must be occurrences of a class that arises from BaseException. In a try statement with an except clause that references a particular class, that clause further handles any exception classes derived from that class .Two exception classes that are not connected via sub classing are never equal, even if they have the same name. User code can advance built-in exceptions. This can be used to test an exception handler and also to report an error condition.

123

8.10 QUESTIONS

1. Write a Python program to read an entire text file

2. Write a Python program to append text to a file and display the text

3. Write a Python program to read a file line by line store it into a variable

4. Write a Python program to count the number of lines in a text file

5. Write a Python program to write a list to a file

6. Write a Python program to extract characters from various text files and puts them into a list.

7. What are exceptions in Python?

8. When would you not use try-except?

9. When will the else part of try-except-else be executed?

10. How can one block of except statements handle multiple exception? ...

8.11 REFERENCES

1. https://www.learnpython.org/

2. https://www.packtpub.com/tech/python

3. https://www.softcover.io/read/e4cd0fd9/conversational-python/ch6_files_excepts

4. https://docs.python.org/3/library/exceptions.html

5. https://www.tutorialspoint.com/python/python_exceptions.htm

6. https://www.w3schools.com/python/python_try_except.asp

7. https://www.geeksforgeeks.org/python-exception-handling/

8. https://www.analyticsvidhya.com/blog/2020/04/exception-handling-python/

9. https://www.programiz.com/python-programming/file-operation

10. https://www.geeksforgeeks.org/file-handling-python/

11. https://realpython.com/read-write-files-python/

12. https://www.guru99.com/reading-and-writing-files-in-python.html

*****

UNIT IV

9

REGULAR EXPRESSION

Unit Structure

9.0 OBJECTIVES

- **Regular expressions** are particularly useful for defining filters.
- **Regular expressions** contain a series of characters that define a pattern of text to be matched—to make a filter more specialized, or general.
- The **regular expression** ^AL[.]* searches for all items beginning with AL.

9.1 INTRODUCTION

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as "Does this string match the pattern?", or "Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn't covered in this document, because it requires that you have a good understanding of the matching engine's internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that *can* be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

9.2 CONCEPT OF REGULAR EXPRESSION

You may be familiar with searching for text by pressing **ctrl-F and typing in the words** you're looking for.

Regular expressions go one step further: They allow you to specify a **pattern of text** to search for.

Regular expressions are helpful, but not many non-programmers know about them even though most **modern text editors and word processors**, such as **Microsoft Word** or **OpenOffice**, have find and **find-and-replace** features that can search based on regular expressions.

Regular expressions are huge **time-savers**, not just for software users but also for programmers.

Finding Patterns of Text Without Regular Expressions

Say you want to find a phone number in a string.

You know the pattern:
three numbers, a hyphen, three numbers, a hyphen, and four numbers.
example: 415-555-4242.

Regular expressions, called regexes for short, are descriptions for a pattern of text.

For example, a \d in a regex stands for a digit character— that is, any single numeral 0 to 9.

The regex \d\d\d-\d\d\d-\d\d\d\d is used by Python to match the same text the
a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers.

Any other string would not match the \d\d\d-\d\d\d-\d\d \d\d regex.

But regular expressions can be much more sophisticated.

Example:

adding a 3 in curly brackets ({3}) after a pattern is like saying, "Match this pattern three times."

So the slightly shorter regex \d{3}-\d{3}-\d{4} also matches the correct phone number format.

Symbol and it's Meaning:

Symbol	matching
.	Any character except newline
\d	Digit(0-9)
\D	Not a digit(0-9)
\w	Word character (a-z, A-Z , 0-9, _)
\W	Not a word character
\s	Whitespace (Space , Tab, newline)
\S	Not whitespace (Space, Tab, Newline)
\b	Word boundary
^	Beginning of string
$	End of a string
[ ]	Matches character in brackets
[^]	Matches character not in brackets

Quantifiers :

Symbol	matching
*	0 Or more
+	1 or more
?	0 or one
{3}	Exact number
{3,4}	Range of numbers (minimum , maximum)

9.3 VARIOUS TYPES OF REGULAR EXPRESSIONS

The "re" package provides several methods to actually perform queries on an input string. We will see the methods of re in Python:

Note: Based on the regular expressions, Python offers two different primitive operations. The match method checks for a match only at the beginning of the string while search checks for a match anywhere in the string.

9.3.1 re.search(): Finding Pattern in Text:

re.search() function will search the regular expression pattern and return the first occurrence. Unlike Python re.match(), it will check all lines of the input string. The Python re.search() function returns a match object when the pattern is found and "null" if the pattern is not found

In order to use search() function, you need to import Python re module first and then execute the code. The Python re.search() function takes the "pattern" and "text" to scan from our main string.

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example:
Search for the first white-space character in the string:

import re

txt = "The rain in Spain"

x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())

output:

The first white-space character is located in position: 3

9.3.2 The split() Function:

The split() function returns a list where the string has been split at each match:

Example:

Split at each white-space character:
import re

txt = "The rain in Spain"

x = re.split("\s", txt)

print(x)
output:
['The','rain','in','Spain']

9.3.3 re.findall():

  findall() module is used to search for "all" occurrences that match a given pattern. In contrast, search () module will only return the first occurrence that matches the specified pattern. findall () will iterate over all the lines of the file and will return all non-overlapping matches of pattern in a single step.

The findall () function returns a list containing all matches

Example:

Print a list of all matches:
import re

txt = "The rain in Spain"

x = re.findall("ai", txt)

print(x)
output:
['ai' , 'ai']

  For example, here we have a list of e-mail addresses, and we want all the e-mail addresses to be fetched out from the list, we use the method re.findall() in Python. It will find all the e-mail addresses from the list.

9.3.4 The Sub () Function:

The sub() function replaces the matches with the text of your choice:
Replace every white-space character with the number 9:

import re

txt = "The rain in Spain"

x = re.sub("\s", "9", txt)

print(x)

output:

The9rain9in9Spain
You can control the number of replacements by specifying the count parameter:

Example:
Replace the first 2 occurrences:

```
import re

txt = "The rain in Spain"

x = re.sub("\s", "9", txt, 2)

print(x)
```

output:
The9rain9inSpain

9.4 USING MATCH FUNCTION

re.match() function of re in Python will search the regular expression pattern and return the first occurrence. The Python RegEx Match method checks for a match only at the beginning of the string. So, if a match is found in the first line, it returns the match object. But if a match is found in some other line, the Python RegEx Match function returns null.

Example
Do a search that will return a Match Object:
```
import re

txt = "The rain in Spain"

x = re.search("ai", txt)

print(x) #this will print an object
```

output:

`<_sre.SRE_Match object; span=(5, 7), match='ai'>`

The Match object has properties and methods used to retrieve information about the search, and the result:

.span() returns a tuple containing the start-, and end positions of the match.

.string returns the string passed into the function

.group() returns the part of the string where there was a match

Example:
Print the string passed into the function:
```
import re

txt = "The rain in Spain"

x = re.search(r"\bS\w+", txt)
```

print(**x.string**)

output:
The rain in Spain

Example:

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case

"S":

import re

txt = "The rain in Spain"

x = re.search(r"\bS\w+", txt)

print(**x.group()**)

output:
Spain

Email validation example

validate the Email from file as well from string by using Regular Expression:

importing the module re

```
# importing the module re
import re
emails='''
CoreyMSchafer@gmail.com
corey.schafer@university.edu
corey-321-schafer@my-work.net
'''
```

`C:\Users\Rahul\Desktop\ss.png`

```
# extract email from string emails
pattern=re.compile(r'([A-Z a-z 0-9 . -]+@[a-z -]+\.(com|edu|net))')
matches =pattern.findall(emails)
for match in matches:
    print(match)

# extract email from file data.txt
pattern=re.compile(r'([A-Z a-z 0-9 . -]+@[a-z -]+\.(com|edu|net))')
with open("data.txt","r") as f:
    contents=f.read()
matches =pattern.findall(contents)
for match in matches:
    print(match)
```

Mobile number validation

```python
#importing module re
import re
text_to_search='''
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
MetaCharacters (Need to be escaped):
. ^ $ * + ? { } [ ] \ | ( )
coreyms.com
321-555-4321
123.555.1234
236*234*1879
091-9732520247
+91-8615714913
9732520247
'''
#extracting mobile number from string text_to_search
pattern=re.compile(r'[0+]?9?1?[-]?\d{10}')
matches=pattern.findall (text_to_search)
    for match in matches:
        print (match)

#extracting the mobile from data.txt file
pattern=re.compile(r'[0+]?9?1?[-]?\d{10}')
with open("data.txt",'r')as f:
    contents=f.read()
    matches=pattern.findall (contents)
    for match in matches:
        print (match)
```

Url validation

```python
import re
urls='''
https://www.google.com
http://sinhgad.edu
https://youtube.com
'''

pattern=re.compile(r'(https?://(www\.)?[a-z]+\.(com|edu))')
matches =pattern.search(urls)
print (natches)
```

9.7 SUMMARY

A regular expression in a programming language is a special text string used for describing a search pattern. It includes digits and punctuation and all special characters like $#@!%, etc. Expression can include literal

- Text matching
- Repetition
- Branching
- Pattern-composition etc.

In Python, a regular expression is denoted as RE (REs, regexes or regex pattern) are embedded through Python re module.

132

9.8 BIBLIOGRAPHY

1. Python for Beginners by Shroff Publishers
2. https://www.w3schools.com/python/python_regex.asp
3. https://www.tutorialspoint.com/python/python_reg_expressions.htm
4. https://www.guru99.com/python-regular-expressions-complete-tutorial.html
5. https://docs.python.org/3/howto/regex.html

9.9 UNIT END EXERCISES

1) Explain the Regular Expression and Pattern Matching in details
2) Write a code to Validate mobile number by using regular expressions.
3) Write a code to Validate URL by using regular expressions.
4) Write a code to Validate Email by using regular expressions.

*****

<div align="right">

10

</div>

CLASSES AND OBJECTS

Unit Structure

10.0 OBJECTIVES

- Classes provide an easy way of keeping the data members and methods together in one
- Place which helps in keeping the program more organized.
- Using classes also provides another functionality of this object-oriented programming
- Paradigm, that is, inheritance.
- Classes also help in overriding any standard operator

10.1 OVERVIEW OF OOP

Python is an object-oriented programming language. It allows us to develop applications using Object Oriented approach. In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below
Object
Class
Method
Inheritance

Polymorphism
Data Abstraction
Encapsulation

Object:
Object is an entity that has state and behavior. It may be anything. It may be physical and logical. For example: mouse, keyboard, chair, table, pen etc.

Class:
Class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods.

Inheritance:
Inheritance is a feature of object-oriented programming. It specifies that one object acquires all the properties and behaviors of parent object. By using inheritance you can define a new class with a little or no changes to the existing class. The new class is known as derived class or child class and from which it inherits the properties is alled base class or parent class. It provides re-usability of the code.

Polymorphism:
Polymorphism is made by two words "poly" and "morphs". Poly means many and Morphs means form, shape. It defines that one task can be performed in different ways.

For example:

You have a class animal and all animals talk. But they talk differently. Here, the "talk" behavior is polymorphic in the sense and totally depends on the animal. So, the abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".

Encapsulation:
Encapsulation is also the feature of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Data Abstraction:

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things, so that the name captures the core of what a function or a whole program does.

135

10.2 CLASS DEFINITION, CREATING OBJECTS

Class:

Class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class then it should contain an attribute and method i.e. an email id, name, age, salary etc.

For example: if you have an employee class then it should contain an attribute and method i.e. an email id, name, age, salary etc.

Syntax:
class ClassName:
<statement-1>
.
.
.
<statement-N>

Method:

Method is a function that is associated with an object. In Python, method is not unique to class instances. Any object type can have methods.

Object:

Object is an entity that has state and behavior. It may be anything. It may be physical and logical. For example: mouse, keyboard, chair, table, pen etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the doc string defined in the function source code.

Syntax
class ClassName:
self.instance_variable = value #value specific to instance
class_variable = value #value shared across all class instances
#accessing instance variable
class_instance = ClassName()
class_instance.instance_variable
#accessing class variable
ClassName.class_variable

Example
class Car:
wheels = 4 # class variable
def __init__(self, make):

```python
self.make = make #instance variable
newCar = Car("Honda")
print ("My new car is a {}".format(newCar.make))
print ("My car, like all cars, has {%d} wheels".format(Car.wheels))
```

10.3 INSTANCES AS ARGUMENTS AND INSTANCES AS RETURN VALUES

Functions and methods can return objects. This is actually nothing new since everything in Python is an object and we have been returning values for quite some time. The difference here is that we want to have the method create an object using the constructor and then return it as the value of the method.

Suppose you have a point object and wish to find the midpoint halfway between it and some other target point. We would like to write a method, call it halfway that takes another Point as a parameter and returns the Point that is halfway between the point and the target.

Example:
```python
class Point:
    def __init__(self, initX, initY):
        """ Create a new point at the given coordinates. """
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x=" + str(self.x) + ", y=" + str(self.y)

    def halfway(self, target):
        mx = (self.x + target.x) / 2
        my = (self.y + target.y) / 2
        return Point(mx, my)
p = Point(3, 4)
q = Point(5, 12)
mid = p.halfway(q)

print(mid)
print(mid.getX())
print(mid.getY())
```

The resulting Point, mid, has an x value of 4 and a y value of 8. We can also use any other methods since mid is a Point object.

In the definition of the method halfway see how the requirement to always use dot notation with attributes disambiguates the meaning of the attributes x and y: We can always see whether the coordinates of Point self or target are being referred to.

Instances as return values:

When you call an instance method (e.g. func) from an instance object (e.g. inst), Python automatically passes that instance object as the first argument, in addition to any other arguments that were passed in by the user.

In the example there are two classes Vehicle and Truck, object of class Truck is passed as parameter to the method of class Vehicle. In method main() object of Vehicle is created.

Then the add_truck() method of class Vehicle is called and object of Truck class is passed as parameter.

Example:

```
class Vehicle:
    def __init__(self):
        self.trucks = []

    def add_truck(self, truck):
        self.trucks.append(truck)

class Truck:
    def __init__(self, color):
        self.color = color

    def __repr__(self):
        return "{}".format(self.color)

def main():
    v = Vehicle()
    for t in 'Red Blue Black'.split():
        t = Truck(t)
        v.add_truck(t)
    print(v.trucks)

if __name__ == "__main__":
    main()
```

Sample output of above program:

[Red, Blue, Black]

10.4 BUILT-IN CLASS ATTRIBUTES

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

__dict__ − Dictionary containing the class's namespace.

__doc__ − Class documentation string or none, if undefined.

__name__ − Class name.

__module__ − Module name in which the class is defined. This attribute is "__main__" in interactive mode.

__bases__ − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Example:

For the above class let us try to access all these attributes −

```
classEmployee:
   'Common base class for all employees'
   empCount =0
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount +=1
   def displayCount(self):
      print"Total Employee %d"%Employee.empCount
   def displayEmployee(self):
      print"Name : ",self.name,", Salary: ",self.salary
print"Employee.__doc__:",Employee.__doc__
print"Employee.__name__:",Employee.__name__
print"Employee.__module__:",Employee.__module__
print"Employee.__bases__:",Employee.__bases__
print"Employee.__dict__:",Employee.__dict__
```

Output:

When the above code is executed, it produces the following result −

Employee.__doc__: Common base class for all employees

Employee.__name__: Employee

Employee.__module__: __main__

Employee.__bases__: ()

Employee.__dict__: {'__module__': '__main__', 'displayCount':

139

<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}

10.5 INHERITANCE

What is Inheritance?

Inheritance is a feature of Object Oriented Programming. It is used to specify that one class will get most or all of its features from its parent class. It is a very powerful feature which facilitates users to create a new class with a few or more modification to an existing class.

The new class is called child class or derived class and the main class from which it inherits the properties is called base class or parent class.

The child class or derived class inherits the features from the parent class, adding new features to it. It facilitates re-usability of code.

Python Multilevel Inheritance:

Multilevel inheritance is also possible in Python like other Object Oriented programming languages. We can inherit a derived class from another derived class, this process is known as multilevel inheritance. In Python, multilevel inheritance can be done at any depth.

```python
# Multilevel Inheritance Example
class Employee: # Base class
    raise_amt=1.04
    def __init__(self,first,last,pay):
        self.first=first
        self.last=last
        self.email=first+'.'+last+'@bollywood.com'
        self.pay=pay

class Developer (Employee): # Derived class 1
    def __init__(self,prog_lang):
        Employee.__init__(self,first,last,pay)
        self.prog_lang=prog_lang

class Manager(Developer): # Manager class  is derived class 2
    def __init__(self,first,last,pay,prog_lang,time):
        Developer.__init__(self,first,last,pay,prog_lang)
        self.time=time

mgr1=Manager('akib','Bagwan',25000,'python',10)
print(mgr1.pay)
print(mgr1.time)
print (mgr1.prog_lang)
```
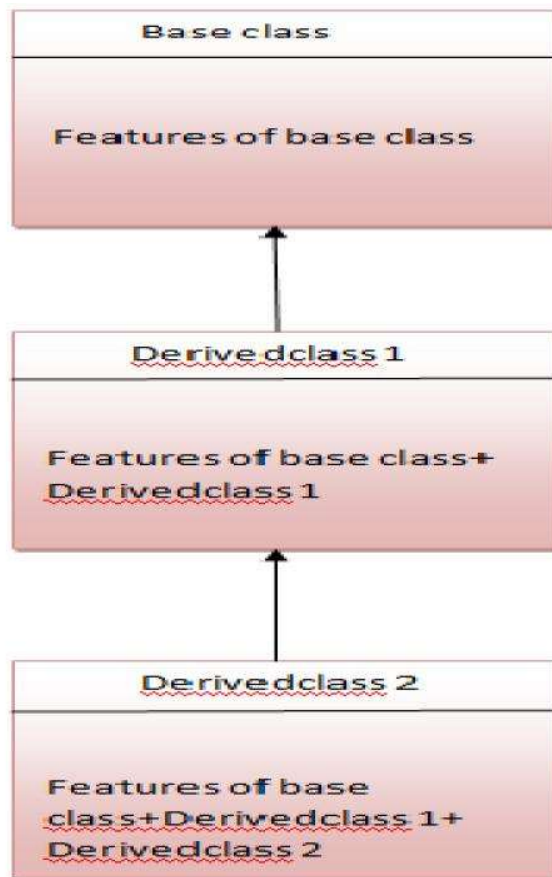
Python Multiple Inheritances:

Python supports multiple inheritance too. It allows us to inherit multiple parent classes. We can derive a child class from more than one base (parent) classes.

141
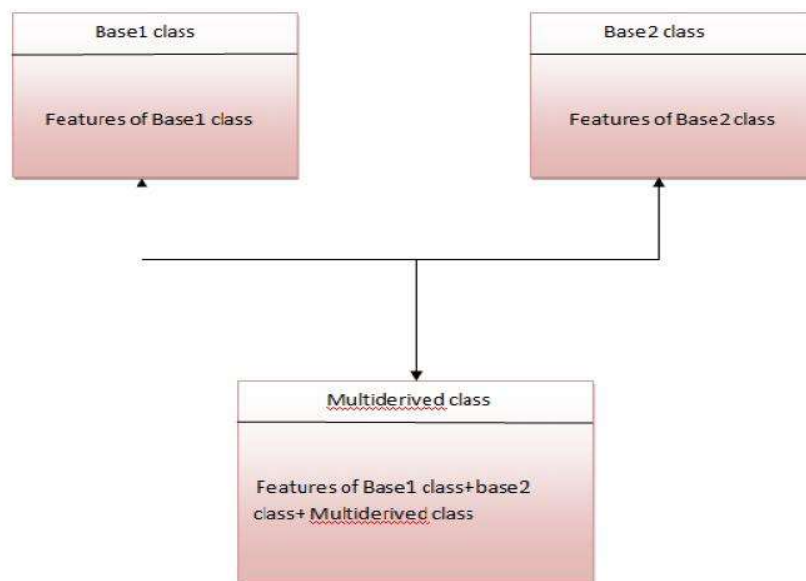
```python
# Multiple Inheritance Example
class Employee: # Base class 1
    raise_amt=1.04
    def __init__(self,first,last,pay):
        self.first=first
        self.last=last
        self.email=first+'.'+last+'@bollywood.com'
        self.pay=pay

class Developer (): # Base class 2
    def __init__(self,prog_lang):
        self.prog_lang=prog_lang

class Manager(Developer,Employee): # Manager class  is derived class
    def __init__(self,first,last,pay,prog_lang,time):
        Employee.__init__(self,first,last,pay)
        Developer.__init__(self,prog_lang)
        self.time=time

mgr1=Manager('akib','Bagwan',25000,'python',10)
print(mgr1.pay)
print(mgr1.time)
print (mgr1.prog_lang)
```

10.6 METHOD OVERRIDING

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.
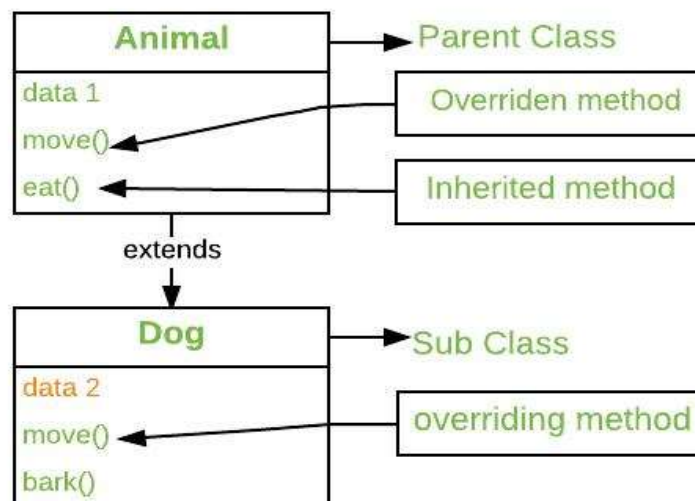
The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to

invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

Example:

```python
class Parent():

    # Constructor
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)

        # Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

Output:
Inside Parent
Inside Child

10.7 DATA ENCAPSULATION

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable

can only be changed by an object's method. Those types of variables are known as **private variable**.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is.

```python
# Creating a base class
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ")
        print(self._a)

obj1 = Derived()

obj2 = Base()

# Calling protected member
# Outside class will  result in
# AttributeError
print(obj2.a)
```

Output:

Calling protected member of base class:

2

Traceback (most recent call last):

  File "/home/6fb1b95dfba0e198298f9dd02469eb4a.py", line 25, in

    print(obj1.a)

AttributeError: 'Base' object has no attribute 'a'

10.8 DATA HIDING

What is Data Hiding?

Data hiding is a part of object-oriented programming, which is generally used to hide the data information from the user. It includes internal object details such as data members, internal working. It maintained the data integrity and restricted access to the class member. The main working of data hiding is that it combines the data and functions into a single unit to conceal data within a class. We cannot directly access the data from outside the class.

This process is also known as the **data encapsulation.** It is done by hiding the working information to user. In the process, we declare class members as private so that no other class can access these data members. It is accessible only within the class.

Data Hiding in Python:

Python is the most popular programming language as it applies in every technical domain and has a straightforward syntax and vast libraries. In the official Python documentation, Data hiding isolates the client from a part of program implementation. Some of the essential members must be hidden from the user. Programs or modules only reflected how we could use them, but users cannot be familiar with how the application works.

Thus it provides security and avoiding dependency as well.

We can perform data hiding in Python using the __ double underscore before prefix. This makes the class members private and inaccessible to the other classes.

Example -
class CounterClass:
    __privateCount = 0

145

```
    def count(self):
        self.__privateCount += 1
        print(self.__privateCount)
counter = CounterClass()
counter.count()
counter.count()
print(counter.__privateCount)
```

Output:
1
2
Traceback (most recent call last):
  File "<string>", line 17, in <module>
AttributeError: 'CounterClass' object has no attribute '__privateCount

10.9 SUMMARY

It allows us to develop applications using an Object-Oriented approach. In Python, we can easily create and use classes and objects. An object-oriented paradigm is to design the program using classes and objects. The oops concept focuses on writing the reusable code.

10.10 UNIT END EXERCISE

1) Explain the object oriented concept in details.

2) Explain the instance return values in details

3) Explain the Data hiding and Data encapsulation with examples.

4) Write a python code to create animal class and in it create one instance variable , and access it   through object .

5) Explain multiple and multilevel inheritance with examples

10.11 BIBLIOGRAPHY

1.https://runestone.academy/runestone/books/published/thinkcspy/Classes Basics/InstancesasrreturnValues.html

2. https://www.tutorialspoint.com/built-in-class-attributes-in-python

3.https://www.pythonlikeyoumeanit.com/Module4_OOP/Methods.html#:~:text=When%20y u%20call%20an%20instance,passed%20in%20by%20the%20user.

4. https://www.geeksforgeeks.org/method-overriding-in-python/

5. https://www.javatpoint.com/data-hiding-in-python.
*****

11

MULTITHREADED PROGRAMMING

Unit Structure

11.0 OBJECTIVES

- To use Multithreading
- To achieve Multithreading
- To use the threading module to create threads
- Address issues or challenges for threads

11.1 INTRODUCTION

What is a Thread in Computer Science?

In software programming, a thread is the smallest unit of execution with the independent set of instructions. It is a part of the process and operates in the same context sharing program's runnable resources like memory. A thread has a starting point, an execution sequence, and a result. It has an instruction pointer that holds the current state of the thread and controls what executes next in what order.

What is multithreading in Computer Science?

The ability of a process to execute multiple threads parallelly is called multithreading. Ideally, multithreading can significantly improve the performance of any program. And Python multithreading mechanism is pretty user-friendly, which you can learn quickly.

11.2 THREAD MODULE

It is started with Python 3, designated as obsolete, and can only be accessed with **_thread** that supports backward compatibility.

How to find Nth Highest Salary in SQL

Syntax:
thread.start_new_thread (function_name, args[, kwargs])

To implement the thread module in Python, we need to import a **thread** module and then define a function that performs some action by setting the target with a variable.

Example:Thread.py

```
import thread # import the thread module
import time # import time module

def cal_sqre(num): # define the cal_sqre function
    print(" Calculate the square root of the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(' Square is : ', n * n)

def cal_cube(num): # define the cal_cube() function
    print(" Calculate the cube of  the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(" Cube is : ", n * n *n)

arr = [4, 5, 6, 7, 2] # given array

t1 = time.time() # get total time to execute the functions
cal_sqre(arr) # call cal_sqre() function
cal_cube(arr) # call cal_cube() function

print(" Total time taken by threads is :", time.time() -
t1) # print the total time
```

Output:
```
Calculate the square root of the given number
 Square is:  16
 Square is:  25
 Square is:  36
 Square is:  49
```

Square is: 4
Calculate the cube of the given number
Cube is: 64
Cube is: 125
Cube is: 216
Cube is: 343
Cube is: 8
Total time taken by threads is: 3.005793809890747

11.3 CREATING A THREAD

Threads in python are an entity within a process that can be scheduled for execution. In simpler words, a thread is a computation process that is to be performed by a computer. It is a sequence of such instructions within a program that can be executed independently of other codes.

In python, there are two ways to create a new Thread. In this article, we will also be making use of the **threading module** in Python.

Below is a detailed list of those processes:

1. Creating python threads using class:

Below has a coding example followed by the code explanation for creating new threads using

Classinpython.

```
# imprt the threading module
import threading

class thread(threading.Thread):
    def __init__(self, thread_name, thread_ID):
        threading.Thread.__init__(self)
        self.thread_name = thread_name
        self.thread_ID = thread_ID

        # helper function to execute the threads
    def run(self):
        print(str(self.thread_name) +" "+ str(self.thread_ID));

thread1 = thread("GFG", 1000)
thread2 = thread("IDOL", 2000);

thread1.start()
thread2.start()
```

print("Exit")

Output:

GFG 1000
IDOL 2000
Exit

2. Creating python threads using function:

The below code shows the creation of new thread using a function:

Example:
```
from threading import Thread
from time import sleep

# function to create threads
def threaded_function(arg):
    for i in range(arg):
        print("running")

        # wait 1 sec in between each thread
        sleep(1)

 if __name__ == "__main__":
    thread = Thread(target = threaded_function, args = (10, ))
    thread.start()
    thread.join()
    print("thread finished...exiting")
```
Output:
```
running
running
running
running
running
running
running
running
running
running
thread finished...exiting
```

So what we did in the above code,
We defined a function to create a thread.

Then we used the threading module to create a thread that invoked the function as its target.

Then we used start() method to start the Python thread.

11.4 SYNCHRONIZING THREADS

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the Lock() method, which returns the new lock.

The acquire(blocking) method of the new lock object is used to force threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock.

If blocking is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The release() method of the new lock object is used to release the lock when it is no longer
required.

```
import threading
import time
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
    # Get lock to synchronize threads
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # Free lock to release next thread
        threadLock.release()
def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1
```

```
threadLock = threading.Lock()
threads = []
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
# Add threads to thread list
threads.append(thread1)
threads.append(thread2)
# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
```

When the above code is executed, it produces the following result −
Starting Thread-1
Starting Thread-2
Thread-1: Thu Mar 21 09:11:28 2013
Thread-1: Thu Mar 21 09:11:29 2013
Thread-1: Thu Mar 21 09:11:30 2013
Thread-2: Thu Mar 21 09:11:32 2013
Thread-2: Thu Mar 21 09:11:34 2013
Thread-2: Thu Mar 21 09:11:36 2013
Exiting Main Thread

11.5 MULTITHREADED PRIORITY QUEUE

The Queue module is primarily used to manage to process large amounts of data on multiple threads. It supports the creation of a new queue object that can take a distinct number of items.

The get() and put() methods are used to add or remove items from a queue respectively.

Below is the list of operations that are used to manage Queue:

get()   : It is used to add an item to a queue.

put()   : It is used to remove an item from a queue.

qsize()   : It is used to find the number of items in a queue.

empty() : It returns a boolean value depending upon whether the queue is

empty or not.

152

full() : It returns a boolean value depending upon whether the queue is full or not.

A Priority Queue is an extension of the queue with the following properties:

An element with high priority is dequeued before an element with low priority.

If two elements have the same priority, they are served according to their order in the queue.

Below is a code example explaining the process of creating multi-threaded priority queue:

Example:

```
import queue
import threading
import time

thread_exit_Flag = 0

class sample_Thread (threading.Thread):
  def __init__(self, threadID, name, q):
    threading.Thread.__init__(self)
    self.threadID = threadID
    self.name = name
    self.q = q
  def run(self):
    print ("initializing " + self.name)
    process_data(self.name, self.q)
    print ("Exiting " + self.name)

# helper function to process data
def process_data(threadName, q):
  while not thread_exit_Flag:
    queueLock.acquire()
    if not workQueue.empty():
      data = q.get()
      queueLock.release()
      print ("% s processing % s" % (threadName, data))
    else:
      queueLock.release()
      time.sleep(1)

thread_list = ["Thread-1", "Thread-2", "Thread-3"]
name_list = ["A", "B", "C", "D", "E"]
```

```python
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# Create new threads
for thread_name in thread_list:
    thread = sample_Thread(threadID, thread_name, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# Fill the queue
queueLock.acquire()
for items in name_list:
    workQueue.put(items)

queueLock.release()

# Wait for the queue to empty
while not workQueue.empty():
    pass

# Notify threads it's time to exit
thread_exit_Flag = 1

# Wait for all threads to complete
for t in threads:
    t.join()
print ("Exit Main Thread")
```

Output:

initializing Thread-1

initializing Thread-2initializing Thread-3

Thread-2 processing AThread-3 processing B

Thread-3 processing C

Thread-3 processing D

Thread-2 processing E

Exiting Thread-2

Exiting Thread-1

Exiting Thread-3

Exit Main Thread

Advantages of Multithreading:

Multithreading can significantly improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.

Multithreading allows a program to remain responsive while one thread waits for input, and another runs a GUI at the same time. This statement holds true for both multiprocessor or single processor systems.

All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables.

Disadvantages of Multithreading:

On a single processor system, multithreading won't hit the speed of computation. The performance may downgrade due to the overhead of managing threads.

Synchronization is needed to prevent mutual exclusion while accessing shared resources. It directly leads to more memory and CPU utilization.

Multithreading increases the complexity of the program, thus also making it difficult to debug.

It raises the possibility of potential deadlocks.

It may cause starvation when a thread doesn't get regular access to shared resources. The application would then fail to resume its work.

11.6 SUMMARY

In this Python multithreading tutorial, you'll get to see different methods to create threads and learn to implement synchronization for thread-safe operations. Each section of this post includes an example and the sample code to explain the concept step by step.

By the way, multithreading is a core concept of software programming that almost all the high-level programming languages support.

11.7 BIBLIOGRAPHY

1.   https://www.javatpoint.com/multithreading-in-python-3

2.   https://www.geeksforgeeks.org/how-to-create-a-new-thread-in-python/

3. https://www.tutorialspoint.com/multithreaded-priority-queue-in-python

4. https://www.techbeamers.com/python-multithreading-concepts/

11.8 UNIT END EXERCISE

1. Explain the differences between multithreading and multiprocessing?
2. Explain different types of multithreading?
3. Explain different types of thread states?
4. Explain the wait () and sleep () methods?
5. Explain different methods for threads?

*****

<div align="right">**12**</div>

MODULE

Unit Structure

12.0 OBJECTIVES

- Modules are simply a 'program logic' or a 'python script' that can be used for variety of applications or functions.

- We can declare functions, classes etc in a module.

- The focus is to break down the code into different modules so that there will be no or minimum dependencies on one another

12.1 INTRODUCTION

If you quit from the Python interpreter and enter it again, the definitions you have made(functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script.

As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program. To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

12.2 IMPORTING MODULE

Import in python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import.

The import statement is the most common way of invoking the import machinery, but it is not the only way.

12.2.1 import module_name:

When the import is used, it searches for the module initially in the local scope by calling __import__() function.

The value returned by the function is then reflected in the output of the initial code.

Example:
import math
print(math.pi)

Output:
3.141592653589793

12.2.2 import module_name.member_name:

In the above code module, math is imported, and its variables can be accessed by considering it to be a class and pi as its object.

The value of pi is returned by __import__().

pi as a whole can be imported into our initial code, rather than importing the whole module.

Example:
from math import pi
# Note that in the above example,
# we used math.pi. Here we have used
# pi directly.
print(pi)

Output:
3.141592653589793

12.2.3 from module_name import * :

In the above code module, math is not imported, rather just pi has been imported as a variable.

All the functions and constants can be imported using *.

Example:
from math import *
print(pi)
print(factorial(6))

Output:
3.141592653589793
720

As said above import uses __import__() to search for the module, and if not found, it would raise ImportError

Example:
import mathematics
print(mathematics.pi)

Output:
Traceback (most recent call last):
  File "C:/Users/GFG/Tuples/xxx.py", line 1, in
    import mathematics
ImportError: No module named 'mathematics'

12.3 CREATING AND EXPLORING MODULES

12.3.1 What are modules in Python?

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for example: **example.py**, is called a module, and its module name would be **example**.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.

```
# Python Module example
def add(a, b):
   """This program adds two
   numbers and return the result"""

   result = a + b
   return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

12.3.2 Importing modules:

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.
>>> import example

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.

Using the module name we can access the function using the dot . operator. For example:
>>> example.add(4,5.5)
9.5

Python has tons of standard modules. You can check out the full list of Python standard modules and their use cases. These files are in the Lib directory inside the location where you installed Python.

Standard modules can be imported the same way as we import our user-defined modules.

12.3.3 Executing a Module as a Script:

Any .py file that contains a module is essentially also a Python script, and there isn't any reason it can't be executed like one.

Here again is mod.py as it was defined above:
mod.py
```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]
def foo(arg):
    print(f'arg = {arg}')
class Foo:
    pass
```

This can be run as a script:
```
C:\Users\john\Documents>python mod.py
C:\Users\john\Documents>
```

There are no errors, so it apparently worked. Granted, it's not very interesting. As it is written, it only defines objects. It doesn't do anything with them, and it doesn't generate any output.

159

Let's modify the above Python module so it does generate some output when run as a script:

mod.py
```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]
def foo(arg):
    print(f'arg = {arg}')
class Foo:
    pass
print(s)
print(a)
foo('quux')
x = Foo()
print(x)
```

Now it should be a little more interesting:

```
C:\Users\john\Documents>python mod.py
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<__main__.Foo object at 0x02F101D0>
```

Unfortunately, now it also generates output when imported as a module:

```
>>>
>>> import mod
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<mod.Foo object at 0x0169AD50>
```

This is probably not what you want. It isn't usual for a module to generate output when it is imported.

Wouldn't it be nice if you could distinguish between when the file is loaded as a module and when it is run as a standalone script? Ask and ye shall receive.

When a .py file is imported as a module, Python sets the special dunder variable __name__ to the name of the module. However, if a file is run as a standalone script, __name__ is (creatively) set to the string '__main__'. Using this fact, you can discern which is the case at run-time and alter behavior accordingly:

mod.py
```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]
def foo(arg):
    print(f'arg = {arg}')
```

```
class Foo:
    pass
if (__name__ == '__main__'):
    print('Executing as standalone script')
    print(s)
    print(a)
    foo('quux')
    x = Foo()
    print(x)
```

Now, if you run as a script, you get output:

C:\Users\john\Documents>python mod.py

Executing as standalone script

If Comrade Napoleon says it, it must be right.

[100, 200, 300]

arg = quux

<__main__.Foo object at 0x03450690>

12.4 MATH MODULE

Python math module is defined as the most famous mathematical functions, which includes trigonometric functions, representation functions, logarithmic functions, etc. Furthermore, it also defines two mathematical constants, i.e., Pie and Euler number, etc.

Pie (n): It is a well-known mathematical constant and defined as the ratio of circumstance to the diameter of a circle. Its value is 3.141592653589793.

Euler's number(e): It is defined as the base of the natural logarithmic, and its value is 2.718281828459045.

There are different math modules which are given below:

math.log ()

This method returns the natural logarithm of a given number. It is calculated to the base e.

Example

HTML Tutorial

import math

number = 2e-7 # small value of of x

print('log(fabs(x), base) is :', math.log(math.fabs(number), 10))

Output:

log(fabs(x), base) is : -6.698970004336019

<

math.log10()

161

This method returns base 10 logarithm of the given number and called the standard logarithm.

Example

import math

x=13 # small value of of x

print('log10(x) is :', math.log10(x))

Output:

log10(x) is : 1.1139433523068367

math.exp()

This method returns a floating-point number after raising e to the given number.

Example

import math

number = 5e-2 # small value of of x

print('The given number (x) is :', number)

print('e^x (using exp() function) is :', math.exp(number)-1)

Output:

The given number (x) is : 0.05

e^x (using exp() function) is : 0.05127109637602412

math.pow(x,y)

This method returns the power of the x corresponding to the value of y. If value of x is negative or y is not integer value than it raises a ValueError.

Example

import math

number = math.pow(10,2)

print("The power of number:",number)

Output:

The power of number: 100.0

math.floor(x)

This method returns the floor value of the x. It returns the less than or equal value to x.

Example:

import math

number = math.floor(10.25201)

print("The floor value is:",number)

Output:
The floor value is: 10
math.ceil(x)
This method returns the ceil value of the x. It returns the greater than or equal value to x.
import math
number = math.ceil(10.25201)
print("The floor value is:",number)

Output:
The floor value is: 11
math.fabs(x)
This method returns the absolute value of x.
import math
number = math.fabs(10.001)
print("The floor absolute is:",number)

Output:
The absolute value is: 10.001
math.factorial()

This method returns the factorial of the given number x. If x is not integral, it raises a **ValueError**.

Example
import math
number = math.factorial(7)
print("The factorial of number:",number)

Output:
The factorial of number: 5040
math.modf(x)

This method returns the fractional and integer parts of x. It carries the sign of x is float.

Example
import math
number = math.modf(44.5)
print("The modf of number:",number)

Output:
The modf of number: (0.5, 44.0)

Python provides the several math modules which can perform the complex task in single-line of code. Here, we have discussed a few important math modules.

12.5 RANDOM MODULE

The Python random module functions depend on a pseudo-random number generator function random(), which generates the float number between 0.0 and 1.0.

There are different types of functions used in a random module which is given below:

random.random()

This function generates a random float number between 0.0 and 1.0.

random.randint()

This function returns a random integer between the specified integers.

random.choice()

This function returns a randomly selected element from a non-empty sequence.

Example

Hello Java Program for Beginners

importing "random" module.

import random

We are using the choice() function to generate a random number from

the given list of numbers.

print ("The random number from list is : ",end="")

print (random.choice([50, 41, 84, 40, 31]))

Output:

The random number from list is : 84

random.shuffle()

This function randomly reorders the elements in the list.

random.randrange(beg,end,step)

This function is used to generate a number within the range specified in its argument. It accepts three arguments, beginning number, last number, and step, which is used to skip a number in the range.

Consider the following example.

We are using **randrange() function** to generate in range from 100

to 500. The last parameter 10 is step size to skip

ten numbers when selecting.

import random

print ("A random number from range is : ",end="")

print (random.randrange(100, 500, 10))

Output:

A random number from range is : 290

random.seed()

This function is used to apply on the particular random number with the seed argument. It returns the mapper value. Consider the following example.

importing "random" module.

import random

using random() to generate a random number

between 0 and 1

print("The random number between 0 and 1 is : ", end="")

print(random.random())

 # using seed() to seed a random number

random.seed(4)

Output:

The random number between 0 and 1 is : 0.4405576668981033

12.6 TIME MODULE

Python has defined a module, "time" which allows us to handle various operations regarding time, its conversions and representations, which find its use in various applications in life. The beginning of time is started measuring from **1 January, 12:00 am, 1970** and this very time is termed as "**epoch**" in Python.

Operations on Time:

1. **time ():** - This function is used to count the number of **seconds elapsed since the epoch**.

2. **gmtime(sec)** :- This function returns a **structure with 9 values** each representing a time attribute in sequence. It converts **seconds into time attributes(days, years, months etc.)** till specified seconds from epoch. If no seconds are mentioned, time is calculated till present. The structure attribute table is given below.

Index	Attributes	Values
0	tm_year	2008
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6	tm_wday	0 to 6
7	tm_yday	1 to 366
8	tm_isdst	-1, 0, 1 where -1 means Library determines DST

165

```
# Python code to demonstrate the working of
# time() and gmtime()
 #  importing "time" module for time operations
import time
 # using time() to display time since epoch
print ("Seconds elapsed since the epoch are : ",end="")
print (time.time())
 # using gmtime() to return the time attribute structure
print ("Time calculated acc. to given seconds is : ")
print (time.gmtime())
```

Output:
Seconds elapsed since the epoch are : 1470121951.9536893
Time calculated acc. to given seconds is :
time.struct_time(tm_year=2016, tm_mon=8, tm_mday=2,
tm_hour=7, tm_min=12, tm_sec=31, tm_wday=1,
tm_yday=215, tm_isdst=0)

3. **asctime("time")** :- This function takes a time attributed string produced by gmtime() and returns a **24 character string denoting time**.

4. **ctime(sec)** :- This function returns a **24 character time string** but takes seconds as argument and **computes time till mentioned seconds**. If no argument is passed, time is calculated till present.

```
# Python code to demonstrate the working of
# asctime() and ctime()
 #  importing "time" module for time operations
import time
 # initializing time using gmtime()
ti = time.gmtime()
 # using asctime() to display time acc. to time mentioned
print ("Time calculated using asctime() is : ",end="")
print (time.asctime(ti))
 # using ctime() to display time string using seconds
print ("Time calculated using ctime() is : ", end="")
print (time.ctime())
```

Output:
Time calculated using asctime() is : Tue Aug 2 07:47:02 2016
Time calculated using ctime() is : Tue Aug 2 07:47:02 2016

5. **sleep(sec)** :- This method is used to **halt the program execution** for the time specified in the arguments.

Python

```
# Python code to demonstrate the working of
# sleep()

#  importing "time" module for time operations
import time

# using ctime() to show present time
print ("Start Execution : ",end="")
print (time.ctime())

# using sleep() to hault execution
time.sleep(4)

# using ctime() to show present time
print ("Stop Execution : ",end="")
print (time.ctime())
```

Python
```
# Python code to demonstrate the working of
# sleep()

#  importing "time" module for time operations
import time

# using ctime() to show present time
print ("Start Execution : ",end="")
print (time.ctime())

# using sleep() to hault execution
time.sleep(4)

# using ctime() to show present time
print ("Stop Execution
```

Output:
Start Execution : Tue Aug 2 07:59:03 2016
Stop Execution : Tue Aug 2 07:59:07 2016

12.7 SUMMARY

A module is a Python object with arbitrarily named attributes that you can bind and reference. Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. Again, we have seen various in-built module like math, time, random modules.

167

12.8 UNIT END EXERCISE

1. Explain the concept of Module in detail with examples.
2. Write a python code to execute module as script.
3. Explain the dir ( ) Function in details.
4. What is package? Write a python code to create package of FRUIT and create two modules APPLE and ORANGE in it and it contains apple and orange classes respectively. Create test script.py file access both the module in it.
5. Write short notes:
   a) Standard module
   b) Intra package references
   c) Module search path

12.9 BIBLIOGRAPHY

1. https://docs.python.org/3/tutorial/modules.html
2. https://www.geeksforgeeks.org/import-module-python/
3. https://realpython.com/python-modules-packages/#executing-a-module-as-a-script
4. https://www.programiz.com/python-programming/modules
5. https://www.javatpoint.com/python-math-module

*****

UNIT V

13

CREATING THE GUI FORM
AND ADDING WIDGETS

Unit Structure

13.1 OBJECTIVES

At the end of this unit, the student will able to

* Design GUI form using any widgets like button, label, checkbutton
* Demonstrate the properties of widget learned in chapter

13.2 INTRODUCTION

1. In python GUI recipes are build using standard built in library of python known as Tkinter.

2. Tkinter is used for creating desktop application.

3. Steps to install python and environment for Tkinter

169

3.1 Prequisite to have installed python in your machine, but check version must be above 3.0 to run tkinter properly

3.2 Next download the python IDE known as pycharm from the link given-https://www.jetbrains.com/pycharm/
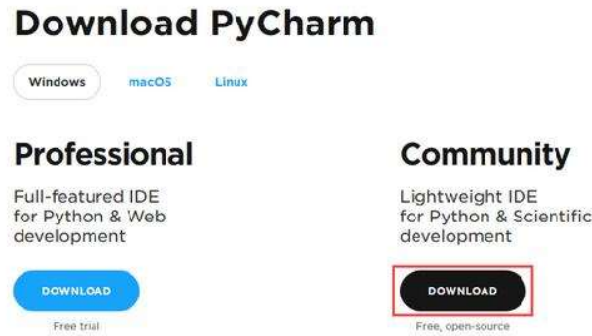
3.3   Download Pycharm Community version for trial of 30 days.

3.4   Once the download is complete, run the exe for install pycharm.

The setup wizard should have started. Click Next as shown in figure below
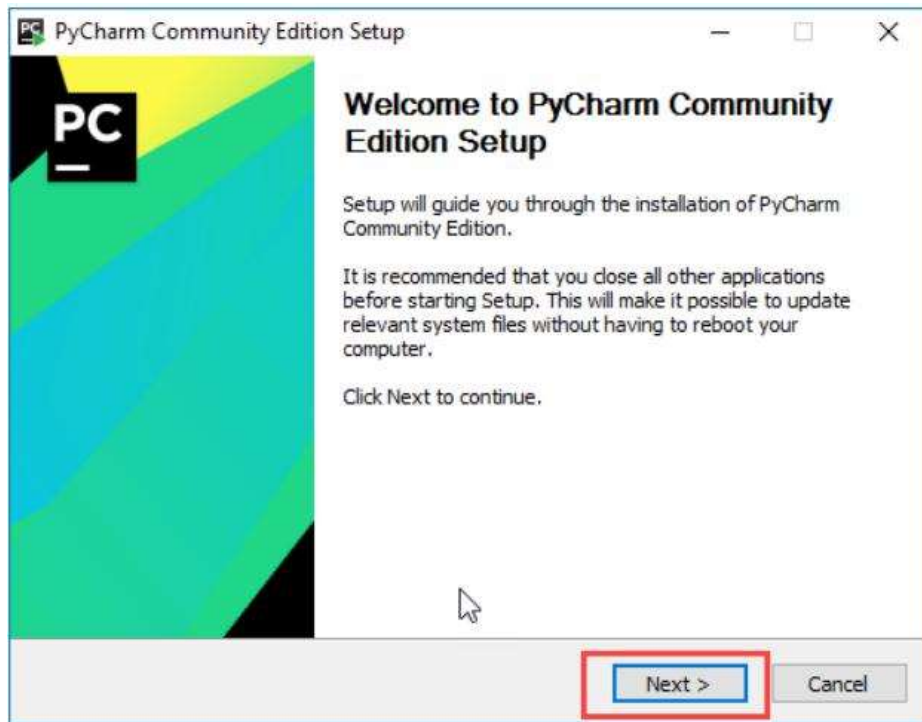
3.5 On the next screen, change the installation path if required. Click on Next as shown in figure below
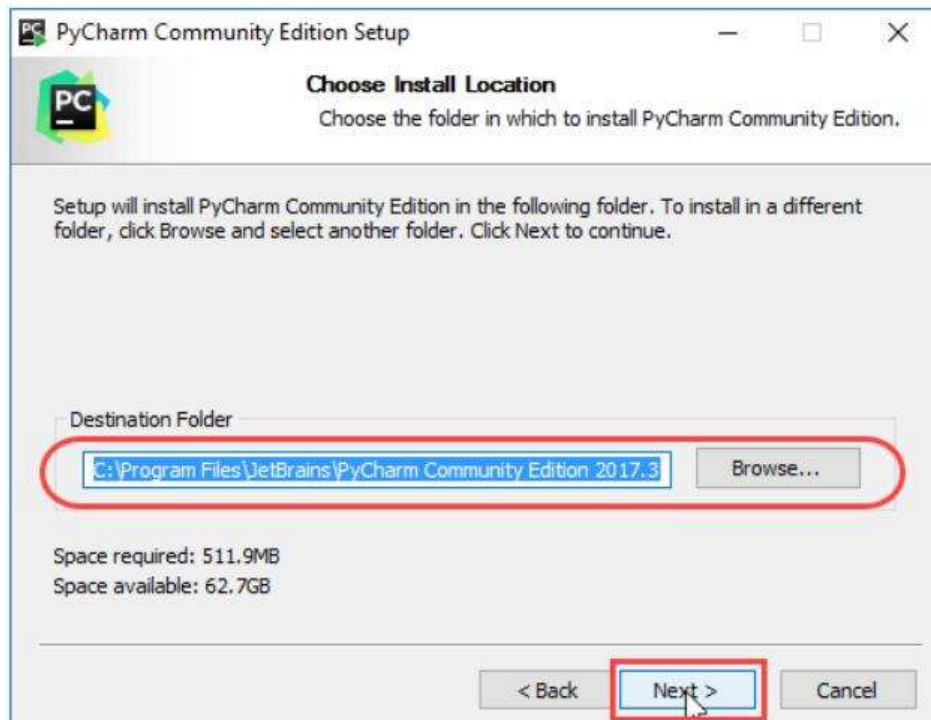
3.6 On the Next Screen, you can create a desktop shortcut if you want and click on "Next"
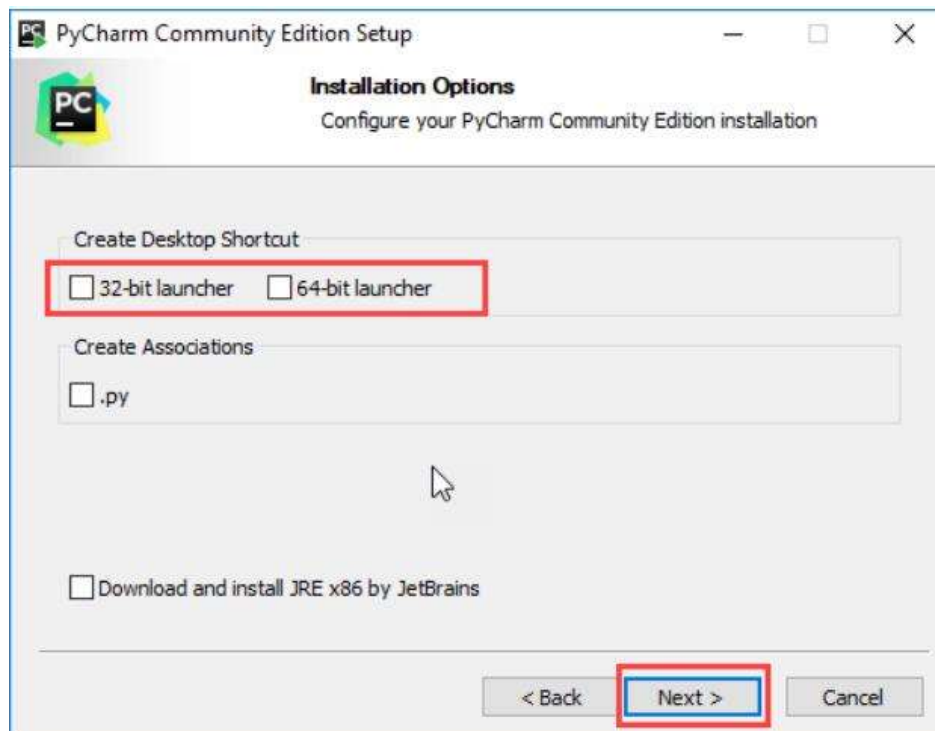
3.7 Choose the start menu folder. Keep selected Jetbrains and click on "Install"
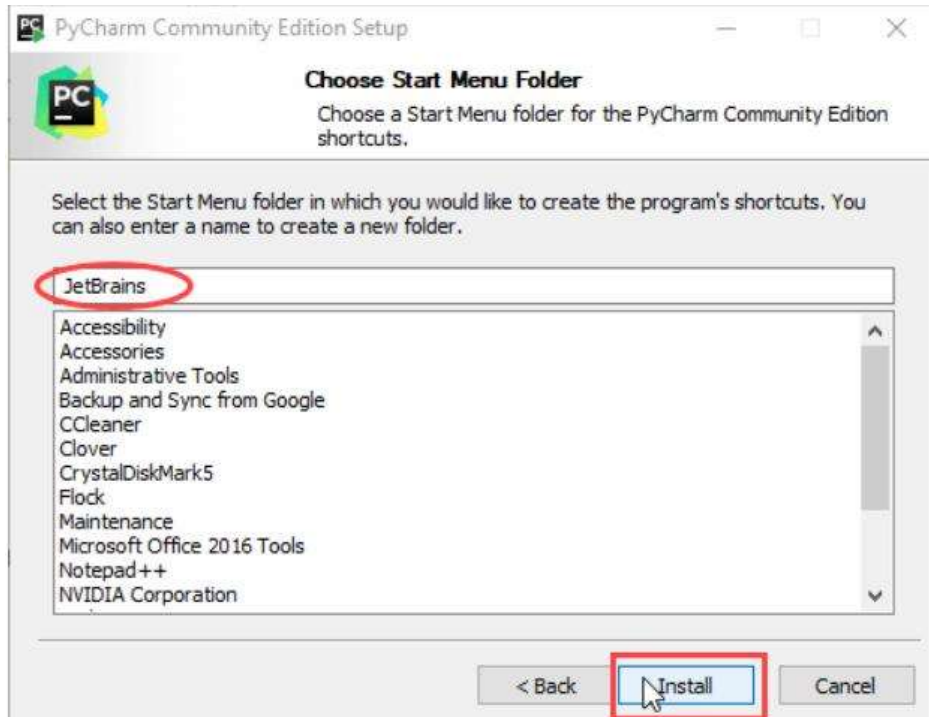
3.8 Wait for the Installation to finish
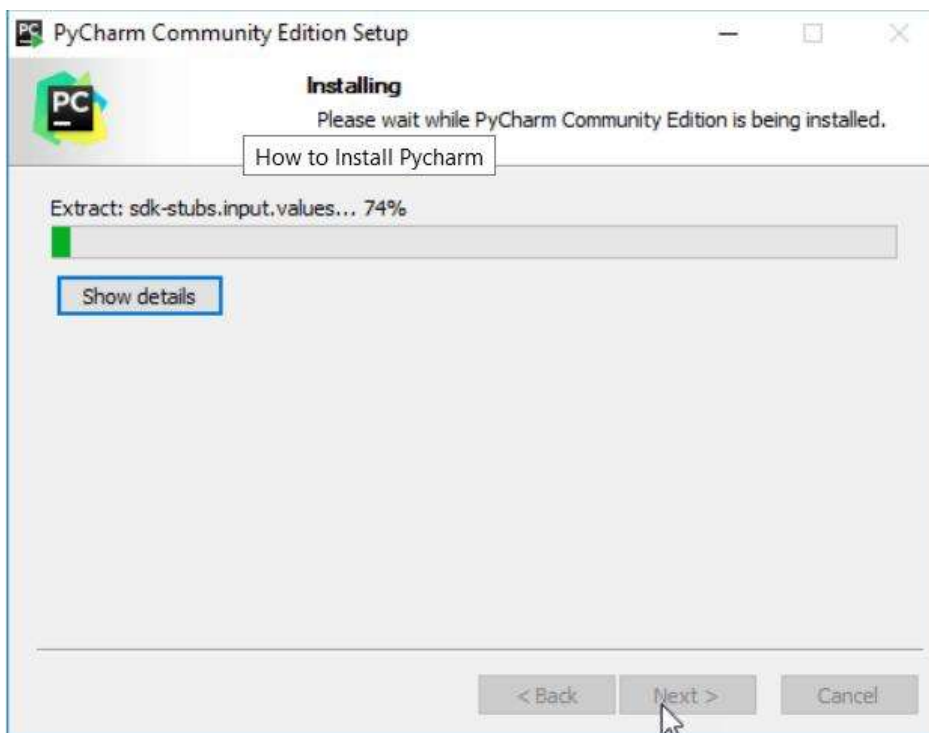
3.9 Once installation finished, you should receive a message screen that pycharm is installed. If you want to go ahead and run it, Click the "Run Pycharm Community edition" box first and click finish.

172
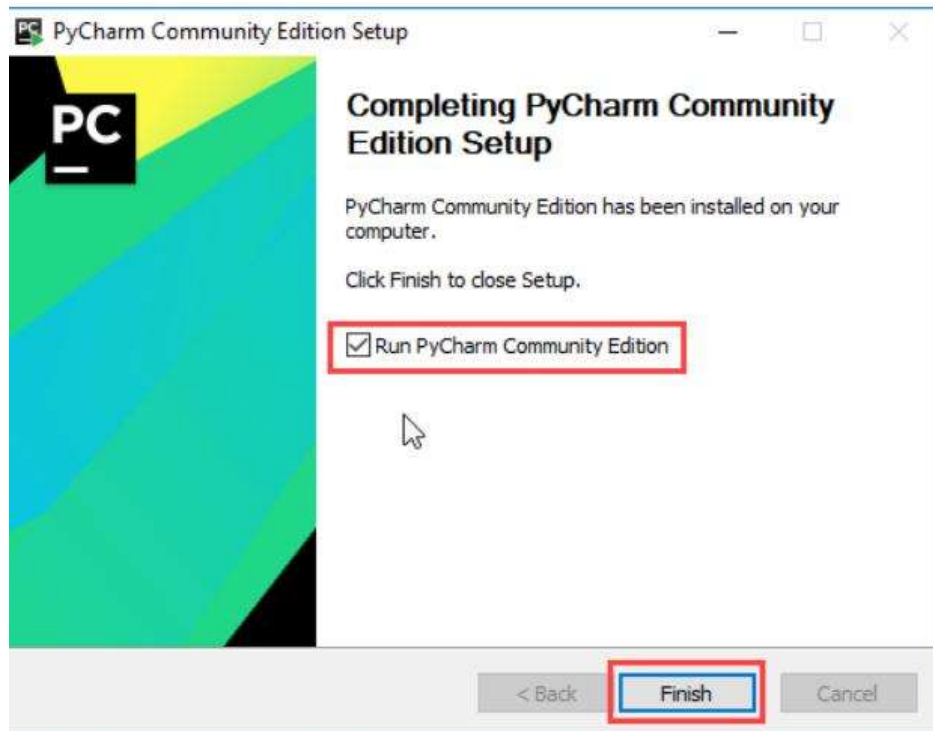
3.10 After you click on "Finish", the Following screen will appear
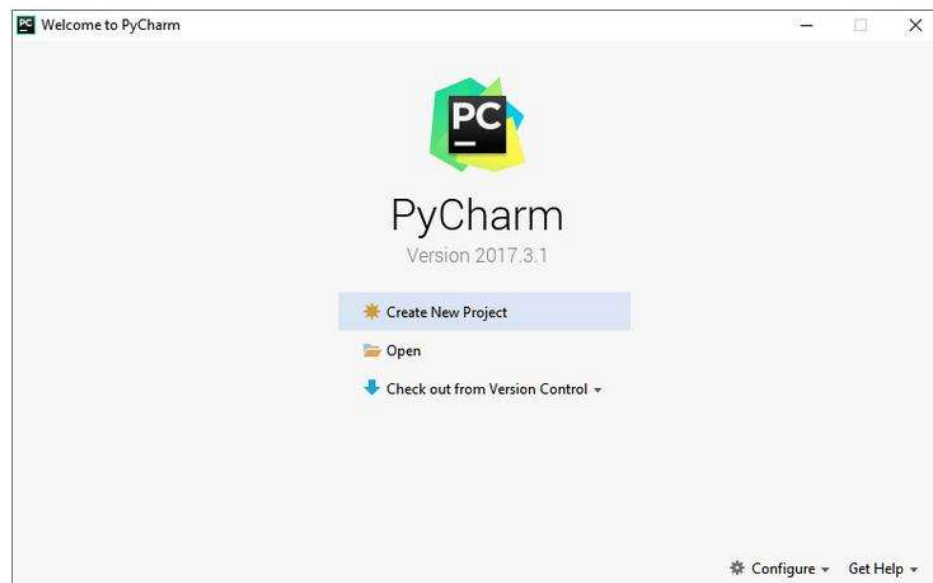
4   An empty Tkinter top-level window can be created by using the following steps

4.1   import the Tkinter Module

4.2   Create the main application window

4.3   Add the widgets like labels, buttons, frames etc to the window

4.4   Call the main event loop so that the actions can take place on the users computer screen

13.3 WIDGETS

There are various widgets like button, canvas, check button, entry etc. that are used to build the python GUI application

1   Label

1   A label is a text used to display some message or information about the other widgets

2   Syntax w= Label(master,options)

3   A list of option are as follows

Sr. No	Option	Description
1	Anchor	It specifies the exact position of the text within the size provided to the widget. The default value is CENTER, which is used to center the text within the specified space
2	bg	The background color displayed behind the widget
3	bitmap	It is used to set the bitmap to the graphical object specified so that, the label can represent the graphics instead of text.
4	bd	It represents the width of the border. The default is 2 pixels.
5	cursor	The mouse pointer will be changed to the type of the cursor specified, i.e., arrow, dot, etc.
6	font	The font type of the text written inside the widget
7	fg	The foreground color of the text written inside the widget.
8	height	The height of the widget
9	image	The image that is to be shown as the label
10	justify	It is used to represent the orientation of the text if the text contains multiple lines. It can be set to LEFT for left justification, RIGHT for right justification, and CENTER for center justification
11	Padx	The horizontal padding of the text. The default value is 1
12	Pady	The vertical padding of the text. The default value is 1.
13	Releif	The type of the border. The default value is FLAT.
14	Text	This is set to the string variable which may contain one or more line of text.
15	textvariable	The text written inside the widget is set to the control variable StringVar so that it can be accessed and changed accordingly.

16	underline	We can display a line under the specified letter of the text. Set this option to the number of the letter under which the line will be displayed.
17	width	The width of the widget. It is specified as the number of characters.
18	Wraplength	Instead of having only one line as the label text, we can break it to the number of lines where each line has the number of characters specified to this option.

3 Code:

```
import               tkinter              as               tk
from             tkinter            import            ttk
#create                                          Instance
win=tk.Tk()
#Add                      a                      title
win.title("Label                                       GUI")
#Adding                  a                      label
ttk.Label(win,      text="A      label").grid(column=0,row=0)
#start                                            GUI
win.mainloop()
```

After executing this program on pycharm using run command, the output of above code as shown below

Fig 1 Label Widget

2 Button:

1. The button widget is used to add various types of buttons to the python application. Python allows the look of the button according to our requirements.

2. Various options can be set or reset depending upon the requirements.

3. We can also associate a method or function with a button which is called when the button is pressed.

4. Syntax

W=Button(parent,options)

5. A list of possible options is illustrated in table below

Sr.No	Option	Description
1.	activebackground	It represents the background of the button when the mouse hover the button.
2.	activeforeground	It represents the font color of the button when the mouse hover the button.
3.	Bd	It represents the border width in pixels.
4.	Bg	It represents the background color of the button.
5.	Command	It is set to the function call which is scheduled when the function is called.
6.	Fg	Foreground color of the button.
7.	Font	The font of the button text
8.	Height	The height of the button. The height is represented in the number of text lines for the textual lines or the number of pixels for the images.
9.	HighlightColor	The color of the highlight when the button has the focus.
10.	Image	It is set to the image displayed on the button.
11.	justify	It illustrates the way by which the multiple text lines are represented. It is set to LEFT for left justification, RIGHT for the right justification, and CENTER for the center.
12.	Padx	Additional padding to the button in the horizontal direction.
13.	Pady	Additional padding to the button in the vertical direction.
14.	Relief	It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE.
15.	State	This option is set to DISABLED to make the button unresponsive. The ACTIVE represents the active state of the button.
16.	Underline	Set this option to make the button text underlined.
17.	Width	The width of the button. It exists as a number of letters for textual buttons or pixels for image buttons.
18.	Wraplength	If the value is set to a positive number, the text lines will be wrapped to fit within this length.

6. Code:

```python
import                  tkinter                  as                  tk
from              tkinter              import              ttk
#create                                              Instance
win=tk.Tk()
#Adding      a      label      that      will      get      modified
a_label=ttk.Label(win,text="A                          Label")
a_label.grid(column=0,row=0)
#Button          click          Event          Function
def                                          click_me():
    action.configure(text="**  I  have  been  clicked!  **")
    a_label.configure(foreground='red')
    a_label.configure(text='A          Red          Label')
#Adding                      a                      Button
action=ttk.Button(win,  text="Click  Me!",command=click_me)
action.grid(column=1,row=0)
#start                                              Gui
win.mainloop()
```
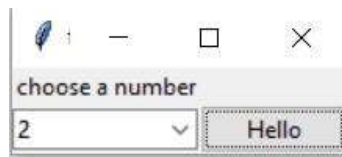
Here Win is the parent of Button. The output of above code is shown below
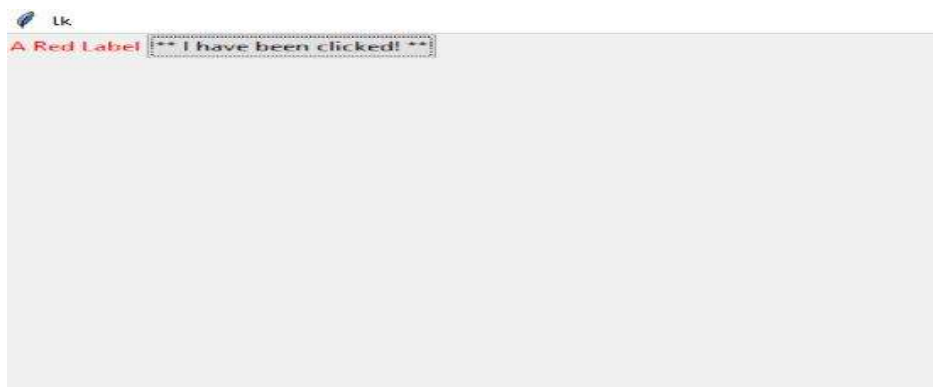
Fig 2 Button Widget

3. Entry TextBox:

1. The Entry widget is used to provide the single line text-box to the user to accept a value from the user.

2. We can use this widget to accept the text strings from the user. It can only be used for one line of text from the user.

3. For multiple lines of text, we must use the text widget.

4. Syntax

   W=Entry(parent,options)

5. A list of possible options is given below

Sr.No	Option	Description
1.	bg	The background color of the widget
2.	bd	The border width of the widget in pixels

177

3.	cursor	The mouse pointer will be changed to the cursor type set to the arrow, dot etc.
4.	exportselection	The text written inside the entry box will be automatically copied to the clipboard by default. We can set the exportselection to 0 to not copy this.
5.	fg	It represents the color of the text
6.	font	It represents the font type of the text
7.	highlightbackground	It represents the color to display in the traversal highlight region when the widget does not have the input focus.
8.	highlightcolor	It represents the color to display in the traversal highlight region when the widget does not have the input focus.
9.	highlightthickness	It represents a non-negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus.
10.	Insertbackground	It represents the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget.
11.	insertbackground	It represents the color to use as background in the are covered by the insertion cursor. This color will normally override either the normal background for the widget.
12.	justify	It specifies how the text is orgranized if the text contains multiple lines.
13.	relief	It specifies the type of the border. Its default value is flat.
14.	selectbackground	The background color of the selected text.
15.	show	It is used to show the entry text of some other type instead of the string. For example the password is typed using stars(*).
16.	textvariable	It is set to the instance of the Stringvar to retrieve the text from the entry.
17.	Width	The width of the displayed text or image.
18.	xscrollcommand	The entry widget can be linked to the horizontal scrollbar if we want the user to enter more text then the actual width of the widget.

6 Code Snippet:

```python
import                      tkinter                    as                      tk
from                    tkinter                    import                    ttk
#create                                                                 Instance
win=tk.Tk()
#Modified              Button              Click              Function
def                                                        click_me():
    action.configure(text='Hello'                          +name.get())
#changing                             our                             label
ttk.Label(win,text="Enter        a         name:").grid(column=0,row=0)

#Adding        a        text        box        Entry        Widget
name=tk.StringVar()
name_entered=ttk.Entry(win,width=12,textvariable=name)
name_entered.grid(column=0,row=1)
#adding                           a                           Button
action=ttk.Button(win,text="Click              Me",command=click_me)
action.grid(column=1,row=0)

#Start                                                                  GUI
win.mainloop()
Output
```


Fig 3 Text box Widget

13.4 COMBOBOX

---

1 code snippet:

179

Ouput

Fig 4 output of Combobox

5. Checkbutton:

1. The checkbutton is used to display the checkbutton on the window.

2. It is used to track the user's choices provided to the application. In other words, we can say that checkbutton is used to implement the on/off selections.

3. The checkbutton can obtain the text or images. The checkbutton is mostly used to provide many choices to the user among which, the user needs to choose the one. It generally implements many of many selections.

4. syntax
   W=checkbutton(master,options)

5. A list of possible options is given below

Sr.No	Option	Description
1.	bitmap	It displays an image (monochrome) on the button.
2.	command	It is associated with a function to be called when the state of the checkbutton is changed.
3.	highlightcolor	The color of the focus highlight when the checkbutton is under focus.
4.	justify	This specifies the justification of the text if the text contains multiple lines.
5.	offvalue	The associated control variable is set to 0 by default if the button is unchecked. We can change the state of an unchecked variable to some other one.
6.	onvalue	The associated control variable is set to 1 by default if the button is checked. We can change the state of the checked variable to some other one.
7.	Variable	It represents the associated variable that tracks the state of the checkbutton
8.	Width	It represents the width of the checkbutton. It is represented in the number of characters that are represented in the form of texts.

180

9.	Wraplength	If this option is set to an integer number, the text will be broken in to the number of pieces.

6. Code Snippet:

```
import tkinter as tk
from tkinter import ttk
win=tk.Tk()
#Creating Three Checkbuttons
chvardis=tk.IntVar()
check1=tk.Checkbutton(win,text="C",variable=chvardis,state='disabled')
check1.select()
check1.grid(column=0,row=4,sticky=tk.W)
```

```
chvardis1=tk.IntVar()
check2=tk.Checkbutton(win,text="Python",variable=chvardis1)
check2.select()
check2.grid(column=1,row=4,sticky=tk.W)
win.mainloop()
```

Output:
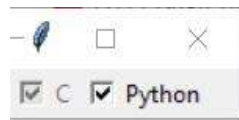
Fig 5 Output of checkbutton

6. Radio Button:

1. The Radiobutton is different from a checkbutton. Here the user is provided with various options and the user can select only one option among them.

2. It is used to implement one-of-many selection in the python application. It shows multiple choices to the user out of which, the user can select only one out of them.

3. We can associate different methods with each of the radiobutton.

4. We can display the multiple line text or images on the radiobuttons. Each button displays a single value for that particular variable.

5. Syntax

181

W=Radiobutton(top,options)

6. The list of possible options given below

Sr.No	Option	Description
1.	command	This option is set to the procedure which must be called every-time when the state of the radiobutton is changed
2.	cursor	The mouse pointer is changed to the specified cursor type. It can be set to the arrow, dot, etc.
3.	font	It represents the font type of the widget text.
4.	fg	The normal foreground color of the widget text
5.	height	The vertical dimension of the widget. It is specified as the number of lines (not pixel).
6.	highlightcolor	It represents the color of the focus highlight when the widget has the focus.
7.	state	It represents the state of the radio button. The default state of the Radiobutton is NORMAL. However, we can set this to DISABLED to make the radiobutton unresponsive.
8.	text	The text to be displayed on the radiobutton.
9.	Textvariable	It is of String type that represents the text displayed by the widget.
10.	Value	The value of each radiobutton is assigned to the control variable when it is turned on by the user.

7. Code snippet:

1.   from tkinter import *
2.
3.   def selection():
4.   selection = "You selected the option " + str(radio.get())
5.   label.config(text = selection)
6.
7.   top = Tk()
8.   top.geometry("300x150")
9.   radio = IntVar()
10.   lbl = Label(text = "Favourite programming language:")

11.   lbl.pack()
12.   R1 = Radiobutton(top, text="C", variable=radio, value=1,
13.   command=selection)
14.   R1.pack(anchor = W)
15.
16.   R2 = Radiobutton(top, text="C++", variable=radio, value=2,
17.   command=selection)
18.   R2.pack(anchor = W)
19.
20.   R3 = Radiobutton(top, text="Java", variable=radio, value=3,
21.   command=selection)
22.   R3.pack(anchor = W)
23.
24.   label = Label(top)
25.   label.pack()
26.   top.mainloop()

Output:
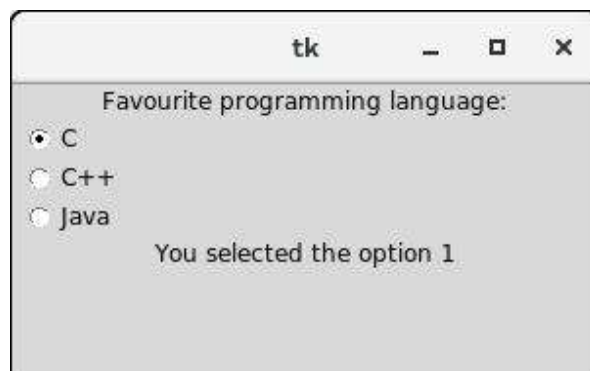
Fig 6 Output of radiobutton

13.3.7 Scrollbar:

1 It provides the scrollbar to the user so that the user can scroll the window up and down.

2 This widget is used to scroll down the content of the other widgets like listbox, text and canvas. However, we can also create the horizontal scrollbars to the entry widget.

3 The syntax to use the scrollbar widget is give below
   W=Scrollbar(top, options)

4 A list of possible options is given below

Sr. No.	option	Description
1	orient	It can be set to HORIZONTAL or VERTICAL depending upon the orientation of the scrollbar.
2	jump	It is used to control the behavior of the scroll jump. If it set to 1, then the callback is called when the user releases the mouse button.
3	repeatdelay	This option tells the duration up to which the button is to be pressed before the slider starts moving in that direction repeatedly. The default is 300 ms.
4	takefocus	We can tab the focus through this widget by default. We can set this option to 0 if we don't want this behavior.
5	troughcolor	It represents the color of the trough.
6	width	It represents the width of the scrollbar

5. Code snippet:

```
from tkinter import *

top = Tk()
sb = Scrollbar(top)
sb.pack(side = RIGHT, fill = Y)

mylist = Listbox(top, yscrollcommand = sb.set )

for line in range(30):
    mylist.insert(END, "Number " + str(line))

mylist.pack( side = LEFT )
sb.config( command = mylist.yview )

mainloop()
```

Output:
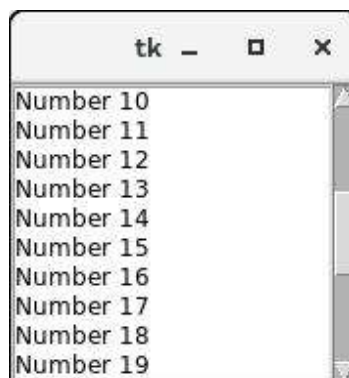
Fig 7 Output of scrollbar

184

8. Listbox:

1. The listbox widget is used to display a list of options to the user.

2. It is used to display the list items to the user. We can place only text items in the ListBox and all text items contain the same font and color.

3. The user can choose one or more items from the list depending upon the configuration.

4. The syntax to use the listbox is given below

   W=ListBox(parent,options)

5. A list of possible options is given below

Sr.No	Options	Description
1	selectbackground	The background color that is used to display the selected text
2	selectmode	It is used to determine the number of items that can be selected from the list. It can set to BROWSE, SINGLE, MULTIPLE, EXTENDED.
3	width	It represents the width of the widget in characters.
4	XscrollCommand	It is used to let the user scroll the Listbox horizontally.
5	Yscrollcommand	It is used to let the user scroll the Listbox vertically.

6. Code Snippet:

```
from tkinter import *
top = Tk()

top.geometry("200x250")

lbl = Label(top,text = "A list of favourite countries...")

listbox = Listbox(top)

listbox.insert(1,"India")

listbox.insert(2, "USA")

listbox.insert(3, "Japan")

listbox.insert(4, "Austrelia")

#this button will delete the selected item from the list

btn = Button(top, text = "delete", command = lambda listbox=listbox: listbox.delete(ANCHOR))
```

lbl.pack()

listbox.pack()

btn.pack()
top.mainloop()

output

Fig 8 Listbox button output

Fig 9 After pressing delete button-output

13.3.9 Menubutton:

1. The menubutton is used to display the menu items to the user.

186

2. It can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

3. The Menubutton is used to implement various types of menus in the python application. A Menu is associated with the menubutton that can display the choices of the menubutton when clicked by the user.

4. The syntax to use the python tkinter menubutton is given below
   W=Menubutton(Top,options)

5. code snippet
   from tkinter import *

top = Tk()

top.geometry("200x250")

menubutton = Menubutton(top, text = "Language", relief = FLAT)

menubutton.grid()

menubutton.menu = Menu(menubutton)

menubutton["menu"]=menubutton.menu

menubutton.menu.add_checkbutton(label = "Hindi", variable=IntVar())

menubutton.menu.add_checkbutton(label = "English", variable = IntVar())

menubutton.pack()

top.mainloop()

output

Fig 10 Output -Menubutton

187

13.3.10 Spinbox:

1 It is an entry widget used to select from options of values.

2. The Spinbox widget is an alternative to the Entry widget. It provides the range of values to the user, out of which, the user can select the one.

3. It is used in the case where a user is given some fixed number of values to choose from.

4. We can use various options with the Spinbox to decorate the widget. The syntax to use the Spinbox is given below

5. Syntax

    W=spinbox(top, options)

6. Code snippet

```
from tkinter import *
top = Tk()
top.geometry("200x200")
spin = Spinbox(top, from_= 0, to = 25)
spin.pack()
top.mainloop()
```
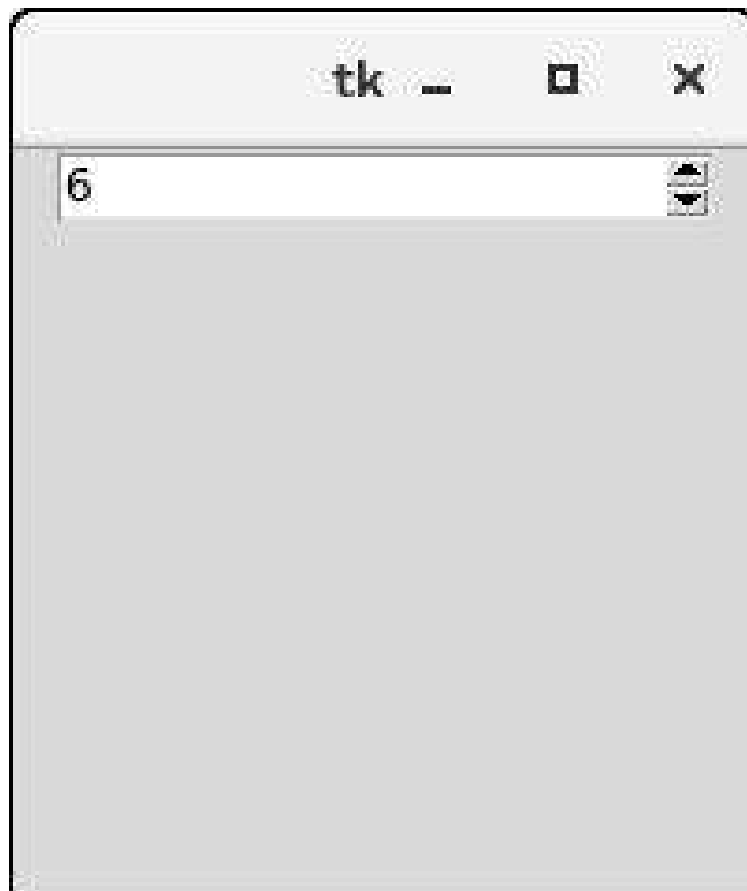
Fig 11 Menubutton

188

13.3.11 PanedWindow:

1 It is like a container widget that contains horizontal or vertical panes.

2 The PanedWindow widget acts like a Container widget which contains one or more child widgets (panes) arranged horizontally or vertically. The child panes can be resized by the user, by moving the separator lines known as sashes by using the mouse.

3 Each pane contains only one widget. The PanedWindow is used to implement the different layouts in the python applications.

4 The syntax to use the PanedWindow is given below
   W=PanedWindow(master,options)

5 Code Snippet

```
from tkinter import *

def add():
    a = int(e1.get())
    b = int(e2.get())
    leftdata = str(a+b)
    left.insert(1,leftdata)

w1 = PanedWindow()
w1.pack(fill = BOTH, expand = 1)

left = Entry(w1, bd = 5)
w1.add(left)

w2 = PanedWindow(w1, orient = VERTICAL)
w1.add(w2)
e1 = Entry(w2)
e2 = Entry(w2)
w2.add(e1)
w2.add(e2)
bottom = Button(w2, text = "Add", command = add)
w2.add(bottom)
mainloop()
```

Output:

Fig 12 PanedWindow
189

13.3.13. TkMessagebox:

1. This module is used to display the message-box in the desktop based applications

2. The messagebox module is used to display the message boxes in the python applications. There are the various functions which are used to display the relevant messages depending upon the application requirements.

3. The syntax to use the messagebox is given below.

    Messagebox.function_name(title,message,[,options])

4. Parameter explanation

    4.1 function_name-It represents an appropriate message box functions

    4.2 title-It is a string which is shown as a title of a messagebox

    4.3 message-It is the string to be displayed as a message on the massagebox

    4.4 Options- There are various options which can be used to configure the message dialog box.

5. **Code Snippet**

    from tkinter import *
     from tkinter import messagebox
     top = Tk()
     top.geometry("100x100")
     messagebox.askquestion("Confirm","Are you sure?")
     top.mainloop()

Output:

Fig 12 Output-Tkmessagebox

190

13.4 SUMMARY

1   In this chapter we discussed about various widgets used in python for handling GUI in python programming.

2   This chapter also revising the concept of each widget with its syntax, options, methods, code and output

3   In this chapter, one section is briefed about installation of pycharm required to handle the python tkinter for GUI purpose

13.6 QUESTIONS

Q1   Design a calculator using widget of python

Q2   Design a pendulum clock

Q3   Design a PingPong game in tkinter

Q4   List down the various options of button widget

Q5   Compare and contrast between the listbox and combobox

13.5 REFERENCES

1.   Python GUI programming Cookbook-Burkahard A Meier, Packt Publication, 2nd Edition

Useful Links

1.   https://www.javatpoint.com/python-tkinter

*****

<div align="right"># 14</div>

LAYOUT MANAGEMENT & LOOK
& FEEL CUSTOMIZATION

Unit structure

14.1 Objectives

14.2 Introduction

14.3 Layout Management-Designing GUI Application with proper layout Management features

14.4 Look & Feel customization- Enhancing look & feel of GUI using different appearances of widgets.

14.5 Summary

14.6 Questions

14.7 References

14.1 OBJECTIVES

At the end of this unit, the student will be able to

- Demonstrate the appearance of Label widget
- Illustrate how widgets dynamically expand the GUI
- Use the grid layout manager
- Describe about the message box, progress bar, canvas widget etc.

14.2 INTRODUCTION

1. In this chapter we will explore how to manage widgets within widgets to create our python GUI.

2. Learning the fundamentals of GUI layout design will enable us to create great-looking GUI's

3. There are certain techniques that will help us in achieving this layout design better.

4. The grid layout manager is one of the most important layout tools built in to tkinter that will be used in this chapter.

5. In this chapter we also going to learn how to customize some of the widgets in our GUI by changing some of their properties.

14.3 LAYOUT MANAGEMENT-DESIGNING GUI APPLICATION WITH PROPER LAYOUT MANAGEMENT FEATURES

1 Arranging several labels within a label frame widget

1.1 The LabelFrame widget allows us to design our GUI in an organized fashion. We are still using the grid layout manager as our main layout design tool, but by using LabelFrame widgets, will get much more control over our GUI design.

1.2 Pseudocode for LabelFrame is

```
# Create a container to hold labels
buttons_frame = ttk.LabelFrame(win, text=' Labels in a Frame ')
buttons_frame.grid(column=0, row=7)
# buttons_frame.grid(column=1, row=7)              # now in col 1

# Place labels into the container element
ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0, sticky=tk.W)
ttk.Label(buttons_frame, text="Label2").grid(column=1, row=0, sticky=tk.W)
ttk.Label(buttons_frame, text="Label3").grid(column=2, row=0, sticky=tk.W)

name_entered.focus()        # Place cursor into name Entry
#=======================
# Start GUI
#=======================
win.mainloop()
```

If we run the above pseudo code the output will simulated as

Fig 1 Output -Label Frame

1.3 We can easily align the labels vertically by changing our code, as shown below in pseudocode

```
# Place labels into the container element
ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0)
ttk.Label(buttons_frame, text="Label2").grid(column=0, row=1)
ttk.Label(buttons_frame, text="Label3").grid(column=0, row=2)

for child in buttons_frame.winfo_children():
```
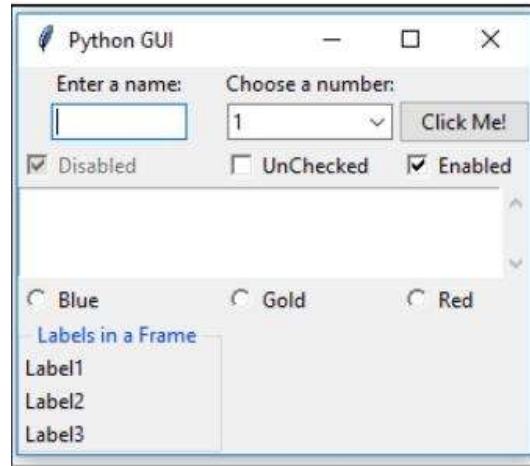
Fig 2 Output of GUI Label Frame

2. Using padding to add space around widgets:

1.4 The procedural way of adding space around widgets is shown here and then use of loop is done to achieve the same thing in a much better way.

1.5 buttons_frame.grid(column=0,row=7,padx=20,pady=40), with this statement labelFrame gets some breathing space.

Fig 3 LabelFrame output

1.6 In tkinter, adding space horizontally and vertically is done by using built-in properties named padx and pady. These can be used to add space around many widgets, improving horizontal and vertical alignments, respectively. We hardcoded 20 pixels of space to the left and right of LabelFrame, and we added 40 pixels to the top and bottom of the frame. Now our LabelFrame stands out better than it did before.

1.7 We can use a loop to add space around the labels contained within LabelFrame

194

```
# Place labels into the container element
ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0)
ttk.Label(buttons_frame, text="Label2").grid(column=0, row=1)
ttk.Label(buttons_frame, text="Label3").grid(column=0, row=2)

for child in buttons_frame.winfo_children():
    child.grid_configure(padx=8, pady=4)

name_entered.focus()        # Place cursor into name Entry
#========================
# Start GUI
#========================
win.mainloop()
```

Pseudocode for adding space around the labels
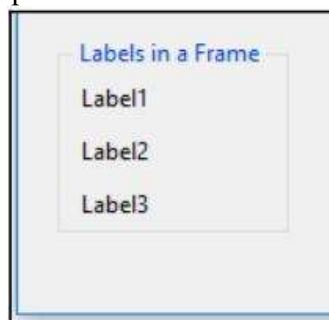
Fig 4 Label Frame widget with Space

1.8 The grid_configure() function enables us to modify the UI elements before the main loop displays them. So, instead of hardcoding values when we first create a widget, we can work on our layout and then arrange spacing towards the end of our file, just before the GUI is created.

1.9 The winfo_children() function returns a list of all the children belonging to the buttons_frame variable. This enables us to loop through them and assign the padding to each label.

3. How widgets dynamically expand the GUI:

1.10 Java introduced the concept of dynamic GUI layout management. In comparison, visual development IDEs, such as VS.NET, layout the GUI in a visual manner and basically hardcode the x and y coordinates of the UI elements.

1.11 Using tkinter, this dynamic capability creates both an advantage and a little bit of a challenge because sometimes our GUI dynamically expands when we would rather it not be so dynamic.

1.12 We are using the grid layout manager widget and it lays out our widgets in a zero based grid. This is very similar to an excel spreadsheet on a data base table.

1.13 The following is an example of a grid layout manager with two rows and three columns

195

Row0,Col0	Row 0, Col 1	Row 0, Col2
Row 1, Col 0	Row 1, Col 1	Row 1, Col2

1.14 Using the grid layout manager, what happens is that the width of any given column is determined by the longest name or widget in that column. This affects all the rows.

1.15 By adding our LabelFrame widget and giving it a title that is longer than some hardcoded size widget, such as the top-left label and the text entry below it, we dynamically move those widgets to the center of column 0, adding space on the left- and right-hand side of those widgets.

1.16 The following code can be added to Label frame code shown above and then placed labels in to his frame

```
# Create a container to hold labels
buttons_frame = ttk.LabelFrame(win, text=' Labels in a Frame ')
buttons_frame.grid(column=0, row=7)
```

4. Aligning the GUI widgets by embedding frames within frames:

1.17 The dynamic behavior of Python and its GUI modules can create a little bit of a challenge to really get our GUI looking the way we want. Here, we will embed frames within frames to get more control of our layout. This will establish a stronger hierarchy among the different UI elements, making the visual appearance easier to achieve.

1.18 Here, we will create a top-level frame that will contain other frames and widgets. This will help us get our GUI layout just the way we want.

1.19 In order to do so, we will have to embed our current controls within a central frame called ttk.LabelFrame. This frame ttk.LabelFrame is the child of the main parent window and all controls will be the children of this ttk.LabelFrame.

1.20 We will only assign LabelFrame to our main window and after that, we will make this LabelFrame the parent container for all the widgets.
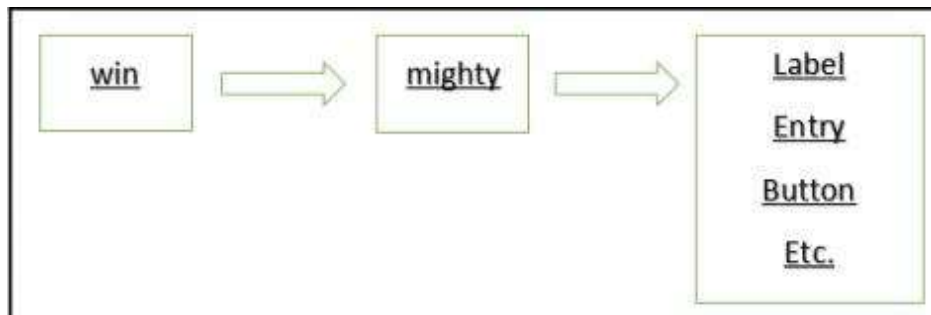
Fig 5 Hierarchy Layout in GUI

In the diagram shown above, win is the variable that holds a reference to our main GUI tkinter window frame, mighty is the variable that holds a reference to our LabelFrame and is a child of the main window frame (win), and Label and all other widgets are now placed into the LabelFrame container (mighty).

```python
#=========================
# imports
#=========================
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext

# Create instance
win = tk.Tk()

# Add a title
win.title("Python GUI")

# We are creating a container frame to hold all other widgets
mighty = ttk.LabelFrame(win, text=' Mighty Python ')
mighty.grid(column=0, row=0, padx=8, pady=4)
```

Next, we will modify the following controls to use mighty as the parent, replacing win. Here is an example of how to do this

```python
# Modify adding a Label using mighty as the parent instead of win
a_label = ttk.Label(mighty, text="Enter a name:")
a_label.grid(column=0, row=0)
```
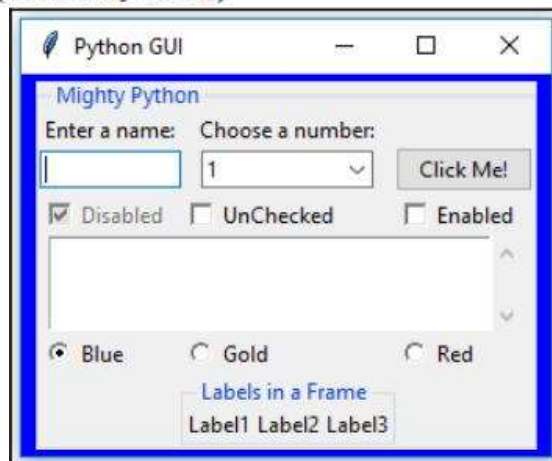

Fig 6 Output of above code

5. Creating Menu bars:

1   We will add a menu bar to our main window, add menus to the menu bar, and then add menu items to the menus.

2   We are creating a Menuitem for functionalities like File, Exit and Help.

197

3   We have to import the Menu class from tkinter. Add the following line of code to the top of the python module, where the import statement live as shown below in pseudocode

```
#=========================
# imports
#=========================
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu

# Create instance
win = tk.Tk()
```

Next, we will create the menu bar, Add the following code towards the bottom of the module, just above where we create the main event loop

```
# Creating a Menu Bar
menu_bar = Menu(win)
win.config(menu=menu_bar)

# Create menu and add menu items
file_menu = Menu(menu_bar)                              # create File menu
file_menu.add_command(label="New")                      # add File menu item
```

In order to make above code in workable condition, we also have to add the menu bar and give it a label.

```
# Create menu and add menu items
file_menu = Menu(menu_bar)                              # create File menu
file_menu.add_command(label="New")                      # add File menu item
menu_bar.add_cascade(label="File", menu=file_menu)      # add File menu to menu bar and give it a label

name_entered.focus()       # Place cursor into name Entry
#=========================
# Start GUI
#=========================
win.mainloop()
```
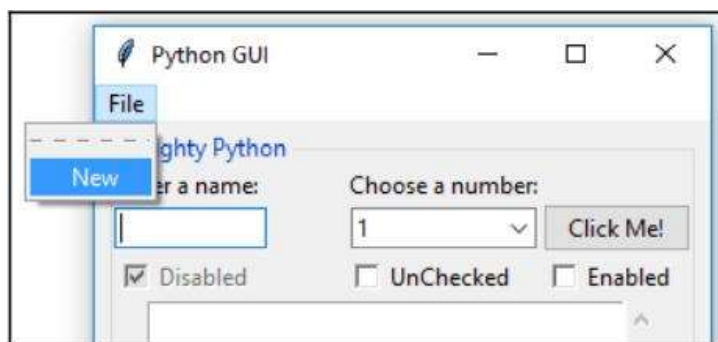

Fig 7 Menu item with File option

Next, we will add a second menu item to the first menu that we added to the menu bar

198

```
# Add menu items
file_menu = Menu(menu_bar)
file_menu.add_command(label="New")
file_menu.add_command(label="Exit")
menu_bar.add_cascade(label="File", menu=file_menu)

name_entered.focus()        # Place cursor into name Entry
#=======================
# Start GUI
#=======================
win.mainloop()
```
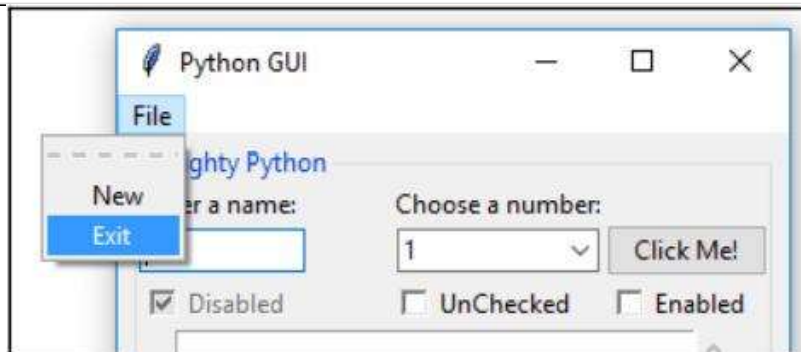

Fig 8 Menu item with Exit option

Next, we will add a help functionalities to our existing menu

```
# Creating a Menu Bar
menu_bar = Menu(win)
win.config(menu=menu_bar)

# Add menu items
file_menu = Menu(menu_bar, tearoff=0)
file_menu.add_command(label="New")
file_menu.add_separator()
file_menu.add_command(label="Exit")
menu_bar.add_cascade(label="File", menu=file_menu)

# Add another Menu to the Menu Bar and an item
help_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="Help", menu=help_menu)
help_menu.add_command(label="About")

name_entered.focus()        # Place cursor into name Entry
#=======================
# Start GUI
#=======================
win.mainloop()
```

199

Fig 9 Menu item with Help option

Next, we will add menu bar exit functionalities

```
# Exit GUI cleanly
def _quit():
    win.quit()
    win.destroy()
    exit()

# Creating a Menu Bar
menu_bar = Menu(win)
win.config(menu=menu_bar)

# Add menu items
file_menu = Menu(menu_bar, tearoff=0)
file_menu.add_command(label="New")
file_menu.add_separator()
file_menu.add_command(label="Exit", command=_quit)
menu_bar.add_cascade(label="File", menu=file_menu)
```

6. Creating tabbed widgets:

1.21 We will create tabbed widgets to further organize our expanding GUI written in tkinter

1.22 Pseudocode for Same

```
#========================
# imports
#========================
import tkinter as tk
from tkinter import ttk

win = tk.Tk()                                  # Create instance
win.title("Python GUI")                        # Add a title
tabControl = ttk.Notebook(win)                 # Create Tab Control
tab1 = ttk.Frame(tabControl)                   # Create a tab
tabControl.add(tab1, text='Tab 1')             # Add the tab
tabControl.pack(expand=1, fill="both")         # Pack to make visible

#========================
# Start GUI
#========================
win.mainloop()
```

Fig 10 Tabbed GUI

7. Using the grid layout manager

1.23 The grid layout manager is one of the most useful layout tools.

1.24 Pseudocode to add grid layout in any python GUI code

```
# Using a scrolled Text control
scrol_w  = 30
scrol_h  =  3
scr = scrolledtext.ScrolledText(mighty, width=scrol_w, height=scrol_h, wrap=tk.WORD)
# scr.grid(column=0, row=2, sticky='WE', columnspan=3)
scr.grid(column=0, sticky='WE', columnspan=3)                    # row not specified
```

Tkinter automatically adds the missing row where we did not specify any particular row.

14.4 Look & Feel customization- Enhancing look & feel of GUI using different appearances of widgets

1. Creating message boxes-information warning and error

1.1 A message box is a pop-up window that gives feedback to the user. It can be informational, hinting at potential problems as well as catastrophic errors.

1.2 Using python to create message boxes is very easy.

1.3 Add the following line of code to the top of the module where the import statement live

```
#=======================
# imports
#=======================
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
from tkinter import messagebox as msg
```

Next, create a callback function that will display a message box. We have to locate the code of the call back above the code where we attach the callback to the menu item, because this is still procedural and not OOP code.

201

Add the following code just above the lines where we create the help menu

```
# Display a Message Box
def _msgBox():
    msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2017.')

# Add another Menu to the Menu Bar and an item
help_menu = Menu(menu_bar, tearoff=0)
help_menu.add_command(label="About", command=_msgBox)    # display messagebox when clicked
menu_bar.add_cascade(label="Help", menu=help_menu)
```


Fig 11 A Help Message box

Next, transform the above code in to a warning message box pop-up window, instead.

```
# Display a Message Box
def _msgBox():
#    msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:'
#                 '\nThe year is 2017.')
    msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:'
                    '\nWarning: There might be a bug in this code.')
```
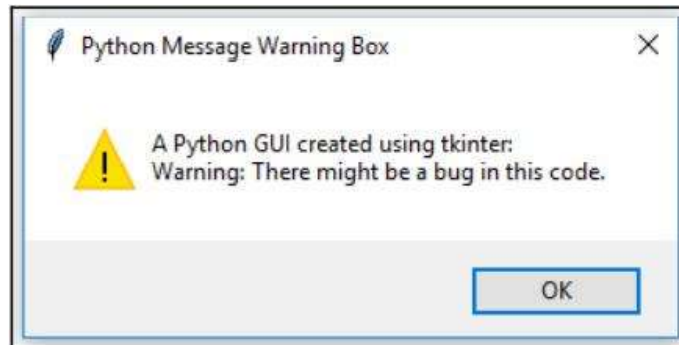

Fig 12 A warning Message

Next we will add error message code to show error message box

```
# Display a Message Box
def _msgBox():
#    msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2017.')
#    msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:'
#                    '\nWarning: There might be a bug in this code.')
    msg.showerror('Python Message Error Box', 'A Python GUI created using tkinter:'
                  '\nError: Houston ~ we DO have a serious PROBLEM!')
```
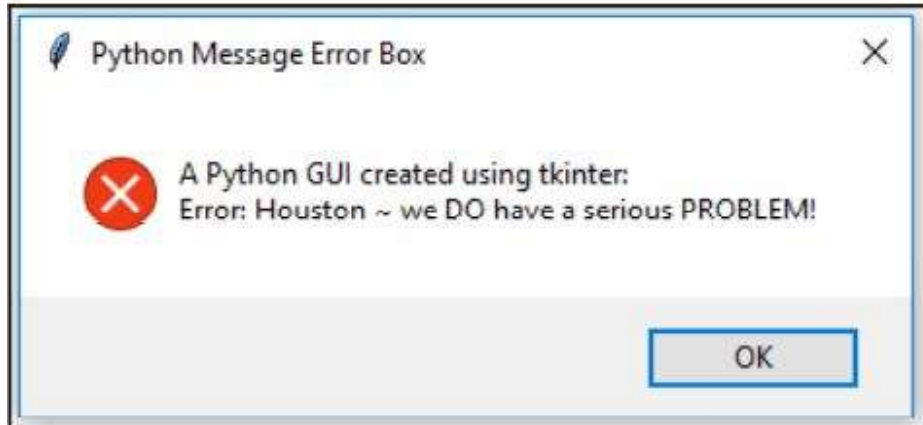
Fig 13 A error message box

2. How to create independent message boxes:

1.4 We will create out tkinter message boxes as stand-alone top-level GUI windows.

1.5 So, why would we wish to create an independent message box? One reason is that we might customize our message boxes and reuse them in several of our GUIs. Instead of having to copy and paste the same code in to every python GUI we design.

```
from tkinter import messagebox as msg
msg.showinfo('Python GUI created using tkinter:\nThe year is 2017')
```
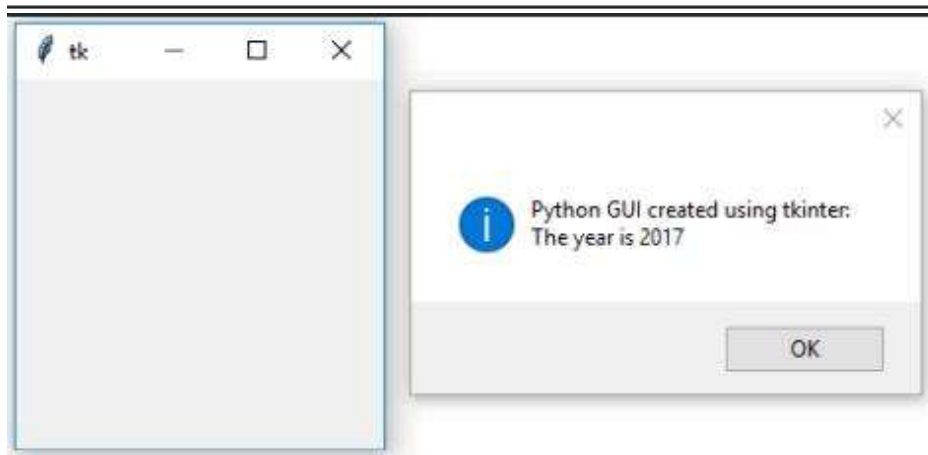

Fig 14 Undesired Output of Message box

1.6 We still need a title and we definitely want to get rid of this unnecessary second window. The second window is caused by a windows event loop. We can get rid of it by suppressing it.

```
from tkinter import messagebox as msg
from tkinter import Tk
root = Tk()
root.withdraw()
msg.showinfo('', 'Python GUI created using tkinter:\nThe year is 2017')
```

203

Fig 15 Independent Message window got by adding withdraw()
in code

2. How to create the title of a tkinter window form:

1.6 The principle of changing the title of a tkinter main root window is the same as what discussed in topic presented above.

1.7 Here we create the main root window and give it a title

```
import tkinter as tk
win = tk.Tk()              # Create instance
win.title("Python GUI")    # Add a title
```
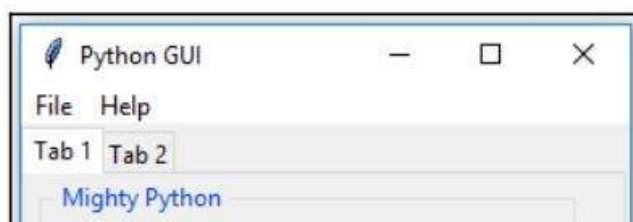

Fig 16 GUI title

3. Changing the icon of the main root window:

1.8 We will use an icon that ships with python but you can use any icon you find useful.

1.9 Place the following code somewher above the main event loop

```
# Change the main windows icon
win.iconbitmap('pyc.ico')
```


Fig 17 Icon added to the main root window

204

4. Using a Spin box control:

1.10 We will use a spinbox widget and we will also bind the Enter key on the keyboard to one of our widget.

1.11 We will use tabbed GUI code and will add further a spinbox widget above the scrolledtext control. This simply requires us to increment the ScrolledText row value by one and to insert our new spinbox control in the row above the entry widget.

1.12 First, we add the Spinbox control. Place the following code above the ScrolledText widget

```
# Adding a Spinbox widget
spin = Spinbox(mighty, from_=0, to=10)
spin.grid(column=0, row=2)
```
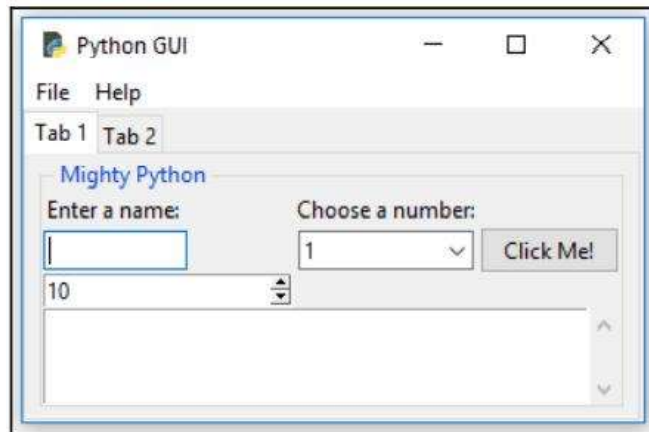

Fig 18 SpinBox Control

Next, we will reduce the sixe of the spinbox widget, by adding following code snippet

spin=spinbox(mighty, from=0,to=10,width=5)
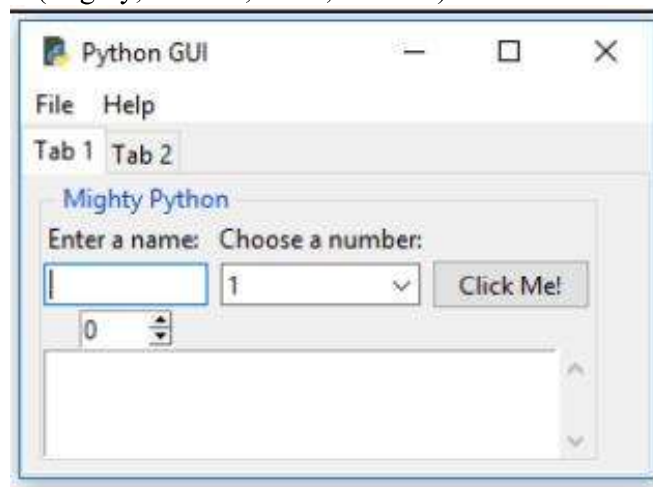
Fig 19 Spin box control with reduce size

205

Next, we add another property to customize our widget further, bd is short-hand notation for the borderwidth property

spin=Spinbox(mighty, from=0, to=0,width-5, bd=8)
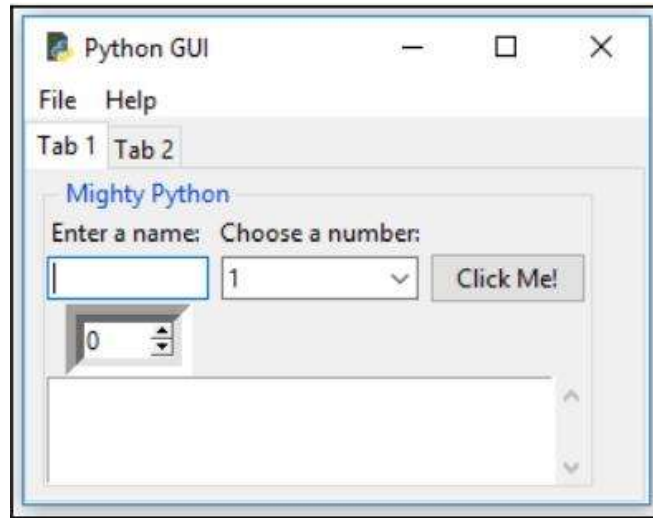
Fig 20 Spin box with border

Here, we add functionality to the widget by creating a callback and linking it to the control

```
# Spinbox callback
def _spin():
    value = spin.get()
    print(value)
    scrol.insert(tk.INSERT, value + 'n')

spin = Spinbox(mighty, from_=0, to=10, width=5, bd=8,
               command=_spin)
```
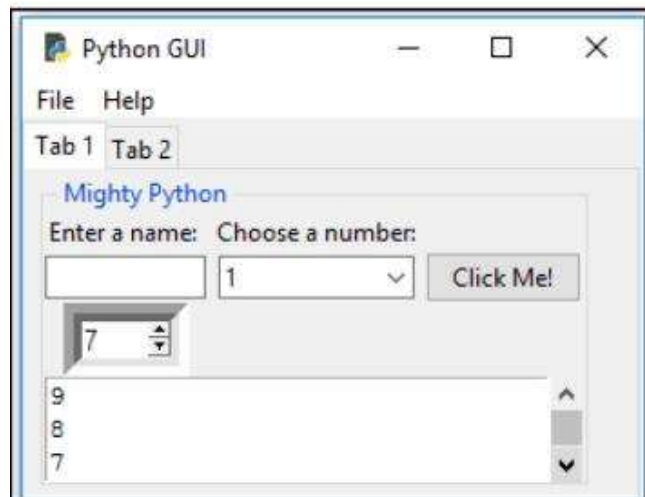

Fig 21 spinbox with small borderwidth

Instead of using a range, we can also specify a set of values

206

```
# Adding a Spinbox widget using a set of values
spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5, bd=8,
               command=_spin)
spin.grid(column=0, row=2)
```
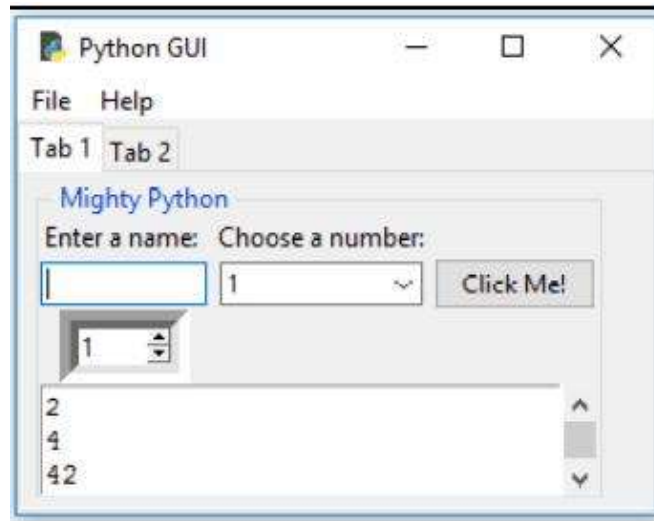

Fig 22 Spinbox with small border width

5. Relief, Sunken and raised appearance of widgets

1.10 We can control the appearance of our spinbox widgets by using a property that makes them appear in different sunken or raised formats.

1.11 We will add one more spinbox control to demonstrate the available appearance of widgets using the relief property of the spinbox control

```
# Adding a second Spinbox widget
spin = Spinbox(mighty, values=(0, 50, 100), width=5, bd=20,
               command=_spin)
spin.grid(column=1, row=2)
```
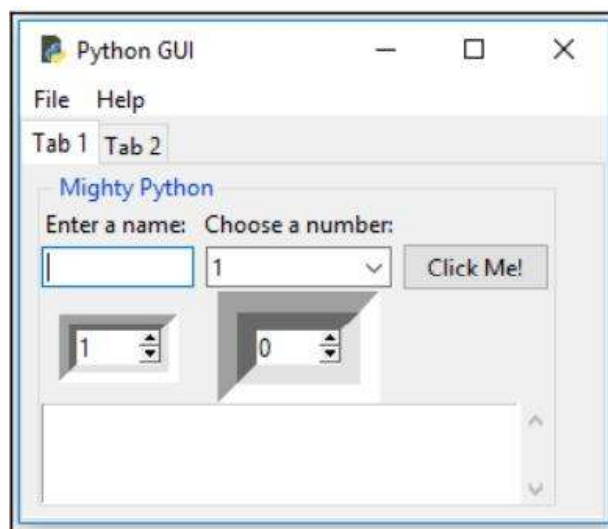

Fig 23 Two Sunken Spinbox

207

3.8 Here are the available relief property options that can be set

tk.SUNKEN	tk.RAISED	tk.FLAT	tk.GROOVE	tk.RIDGE

By assigning the different available options to the relief property, we can create different appearances for this widget. Assigning the tk.RIDGE relief and reducing the border width to the same value as our first spinbox widget results in the following GUI
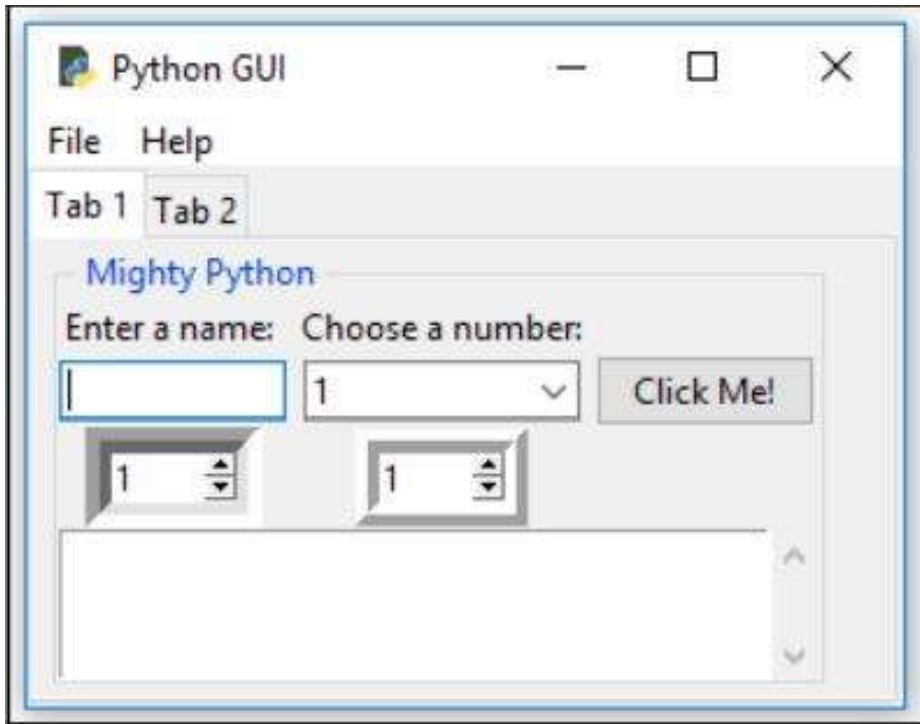
Fig 24 spinbox with two ridge

8. Creating tooltips using python:

1.12   We will be adding more useful functionality to our GUI. Surprisingly, adding a tooltip to our controls should be simple, but it is not as simple as we would wish it to be .

1.13   In order to achieve this desired functionality, we will place our tooltip code in to its own OOP class

```
#==========================================================
class ToolTip(object):
    def __init__(self, widget):
        self.widget = widget
        self.tip_window = None

    def show_tip(self, tip_text):
        "Display text in a tooltip window"
        if self.tip_window or not tip_text:
            return
        x, y, _cx, cy = self.widget.bbox("insert")        # get size of widget
        x = x + self.widget.winfo_rootx() + 25            # calculate to display tooltip
        y = y + cy + self.widget.winfo_rooty() + 25       # below and to the right
        self.tip_window = tw = tk.Toplevel(self.widget)   # create new tooltip window
        tw.wm_overrideredirect(True)                      # remove all Window Manager (wm) decorations
#         tw.wm_overrideredirect(False)                   # uncomment to see the effect
        tw.wm_geometry("+%d+%d" % (x, y))                 # create window size

        label = tk.Label(tw, text=tip_text, justify=tk.LEFT,
                         background="#ffffe0", relief=tk.SOLID, borderwidth=1,
                         font=("tahoma", "8", "normal"))
        label.pack(ipadx=1)

    def hide_tip(self):
        tw = self.tip_window
        self.tip_window = None
        if tw:
            tw.destroy()


#==========================================================
def create_ToolTip(widget, text):
    toolTip = ToolTip(widget)         # create instance of class
    def enter(event):
        toolTip.show_tip(text)
    def leave(event):
        toolTip.hide_tip()
    widget.bind('<Enter>', enter)     # bind mouse events
    widget.bind('<Leave>', leave)
```

In an object oriented programming (OOP) approach, we create a new class in our python module. Python allows us to place more than one class in to same pyhton module and it also enables us to mix-and-match classes and regular functions in the same module.

In our tooltip code, we declare a Python class and explicitly make it inherit from object, which is the foundation of all Python classes. We can also leave it out, as we did in the AClass code example, because it is the default for all Python classes.

After all the necessary tooltip creation code that occirs within the Tooltip class, we switch over to non-OOP python programming by creating a function just below it

We can add a tooltip for our Spinbox widget, as follows
#Add a Tooltip
Create_ToolTip(spin, 'This is a spin control')

We could do the same for all of our other GUI widgets in the very same manner. We just have to pass in a reference to the widget we wish to have a tooltip, displaying some extra information. For our ScrolledText widget, we made the scrol variable point to it, so this is what we pass into the constructor of our tooltip creation function:

209

```
# Using a scrolled Text control
scrol_w  = 30
scrol_h  =  3
scrol = scrolledtext.ScrolledText(mighty, width=scrol_w, height=scrol_h, wrap=tk.WORD)
scrol.grid(column=0, row=3, sticky='WE', columnspan=3)

# Add a Tooltip to the ScrolledText widget
create_ToolTip(scrol, 'This is a ScrolledText widget')
```


Fig 25 ToolTip Output

9. Adding a Progressbar to the GUI

7.1 Progressbar is typically used to show the current status of a long-running process.

7.2 Add four buttons in to Label frame and set the label frame text property to progressbar.

7.3 We connect each of our four new buttons to a new callback function, which we assign to their command property

```
# Add Buttons for Progressbar commands
ttk.Button(buttons_frame, text=" Run Progressbar    ", command=run_progressbar).grid(column=0, row=0, sticky='W')
ttk.Button(buttons_frame, text=" Start Progressbar  ", command=start_progressbar).grid(column=0, row=1, sticky='W')
ttk.Button(buttons_frame, text=" Stop immediately ", command=stop_progressbar).grid(column=0, row=2, sticky='W')
ttk.Button(buttons_frame, text=" Stop after second ", command=progressbar_stop_after).grid(column=0, row=3, sticky='W')
```

```
# Add a Progressbar to Tab 2
progress_bar = ttk.Progressbar(tab2, orient='horizontal', length=286, mode='determinate')
progress_bar.grid(column=0, row=3, pady=2)

# update progressbar in callback loop
def run_progressbar():
    progress_bar["maximum"] = 100
    for i in range(101):
        sleep(0.05)
        progress_bar["value"] = i    # increment progressbar
        progress_bar.update()        # have to call update() in loop
    progress_bar["value"] = 0        # reset/clear progressbar
```

210
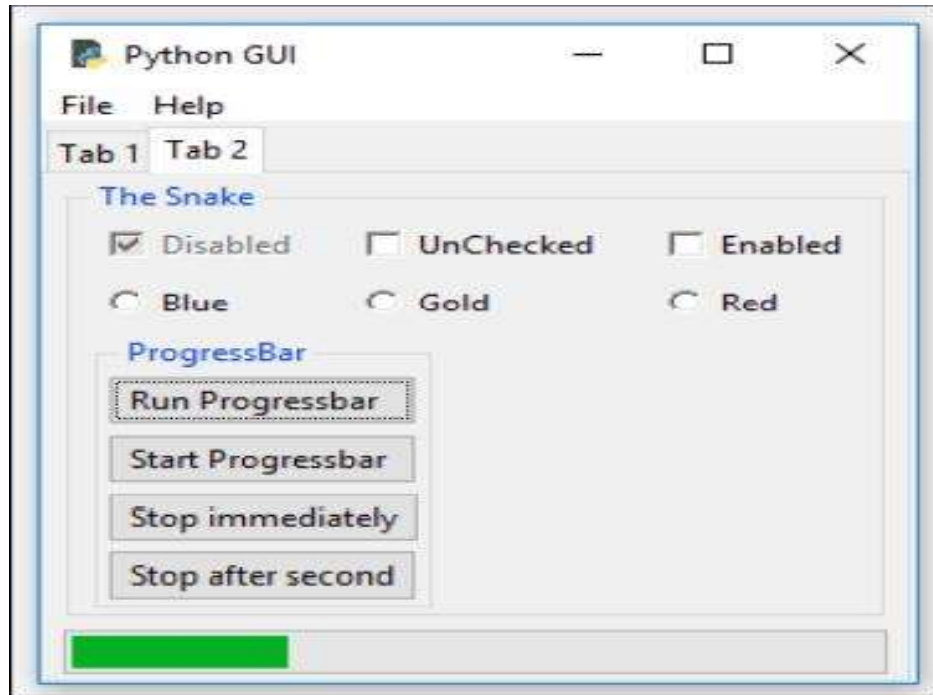
Fig 26 Progress Bar

10. How to use the Canvas Widget:

1.11.1 First, we will create a third tab in our GUI in order to isolate our new code

1.12 Here is the code to create the new third tab

Here is the code to create the new third tab:

```
tabControl = ttk.Notebook(win)          # Create Tab Control

tab1 = ttk.Frame(tabControl)            # Create a tab
tabControl.add(tab1, text='Tab 1')      # Add the tab

tab2 = ttk.Frame(tabControl)
tabControl.add(tab2, text='Tab 2')      # Add a second tab

tab3 = ttk.Frame(tabControl)
tabControl.add(tab3, text='Tab 3')      # Add a third tab

tabControl.pack(expand=1, fill="both")  # Pack to make tabs visible
```

Next, we use another built-in widget of tkinter:canvas. A lot of people like this widget as it has powerful capabilities

```
# Tab Control 3 -------------------------------
tab3_frame = tk.Frame(tab3, bg='blue')
tab3_frame.pack()
for orange_color in range(2):
    canvas = tk.Canvas(tab3_frame, width=150, height=80,
                       highlightthickness=0, bg='orange')
    canvas.grid(row=orange_color, column=orange_color)
```

211
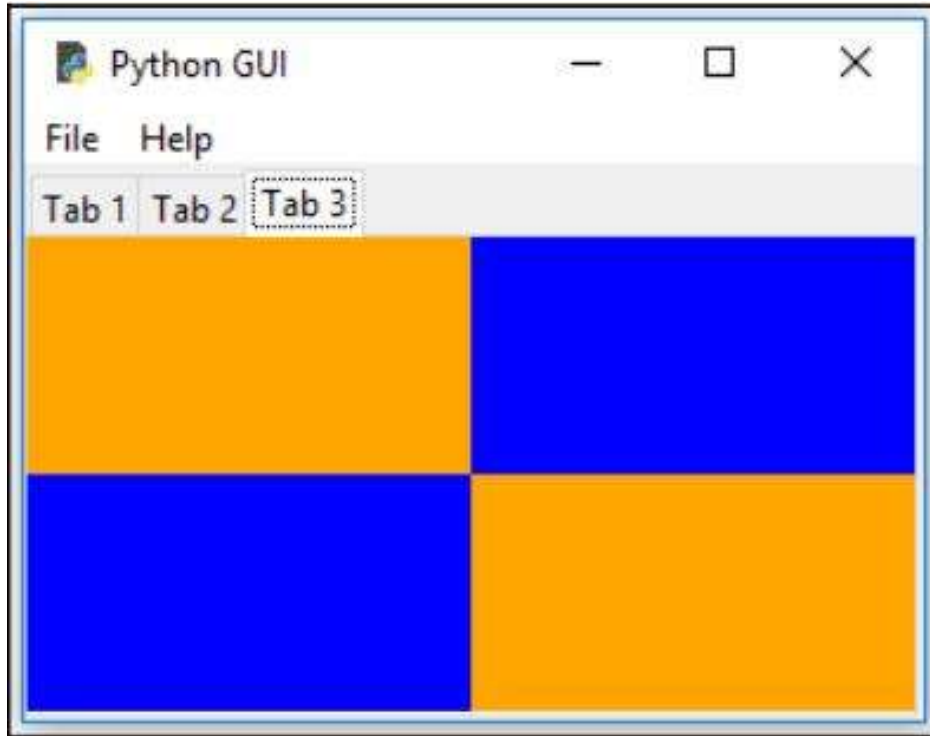
Fig 27 Canvas output

14.5 SUMMARY

- In this chapter we discuss how to add messagebox, tooltip, progress bar, grid layout and other layout management and customized GUI features.

- Codes for every widget is covered with output.

14.6 QUESTIONS

Q1 Design a calculator with proper grid layout

Q2 Change the title of main screen. Write a small code for that

Q3 Discuss how to change the border width of the spin box

Q4 Difference between spinbox and combo box

Q5 Create a Menu driven program for addition, substraction, multiplication and division using Menu bar

14.7 REFERENCES

1. Python GUI programming Cookbook -Burkahard A Meier, Packt Publication, 2nd Edition.

*****

15

STORING DATA IN OUR MYSQL DATABASE VIA OUR GUI

Unit Structure

15.0 OBJECTIVES

At the end of this unit, the learner will be able to
● Demonstrate the steps for connecting python code to MySQL.
● Implement the Insert command
● Implement the Update command
● Implement the Delete command

15.1 INTRODUCTION

---

1. Before we can connect to a MySQL server, we have to have access to a MySQL server. The first thing in this chapter will show you how to install the free MySQL Server Community Edition.

2. After successfully connecting to a running instance of our MySQL server, we will design and create a database that will accept a book title, which could be our own journal or a quote we found somewhere on the Internet. We will require a page number for the book, which could be blank, and then we will insert   the quote we like from a book, journal, website or friend into our MySQL database using our GUI built in Python

3. We will insert, modify, delete and display our favorite quotes using our Python GUI to issue these SQL commands and to display the data.

213

4. CRUD is a database term you might come across that abbreviates the four basic SQL commands and stands for Create , Read, Update, and Delete.

15.2 CONNECTING TO A MYSQL DATABASE FROM PYTHON

1. Before we can connect to a MySQL database, we have to connect to the MySQL server.

2. In order to do this, we need to know the IP address of the MySQL server as well as the port it is listening on.

3. We also have to be a registered user with a password in order to get authenticated by the MySQL server.

4. You will need to have access to a running MySQL server instance and you also need to have administrator privileges in order to create databases and tables.

5. There is a free MySQL Community Edition available from the official MySQL website. You can download and install it on your local PC from h:ttp://dev.mysql.com/downloads/.

6. In order to connect to MySQL, we first need to install a special Python connector driver. This driver will enable us to talk to the MySQL server from Python.

7. The driver is freely available on the MySQL website and comes with a very nice online tutorial. You can install it from:

8. http://dev.mysql.com/doc/connector-python/en/index.html

9. There is currently a little bit of a surprise at the end of the installation process. When we start the .msi installer we briefly see a MessageBox showing the progress of the installation, but then it disappears. We get no confirmation that the installation actually succeeded.

10. One way to verify that we installed the correct driver, that lets Python talk to MySQL, is by looking into the Python site-packages directory.

11. If your site-packages directory looks similar to the following screenshot and you see some new files that havemysql_connector_python in their name, well, then we did indeed install something…

12. The official MySQL website mentioned above comes with a tutorial, at the following URL:http://dev.mysql.com/doc/connector-python/en/connector-python- tutorials.html

13. import mysql.connector as mysql

    conn = mysql.connect(user=<adminUser>, password=<adminPwd>, host='127.0.0.1')

    print(conn)

    conn.close()

If running the preceding code results in the following output printed to the console, then we are good.

14. If you are not able to connect to the MySQL server, then something probably went wrong during the installation. If this is the case, try uninstalling MySQL, reboot your PC, and then run the MySQL installation again. Double-check that you downloaded the MySQL installer to match your version of Python. If you have more than one version of Python installed, that sometimes leads to confusion as the one you installed last gets prepended to the Windows path environmental variable and some installers just use the first Python version they can find in this location.

15. In order to connect our GUI to a MySQL server, we need to be able to connect to the server with administrative privileges if we want to create our own database.

16. If the database already exists, then we just need the authorization rights to connect, insert, update, and delete data.
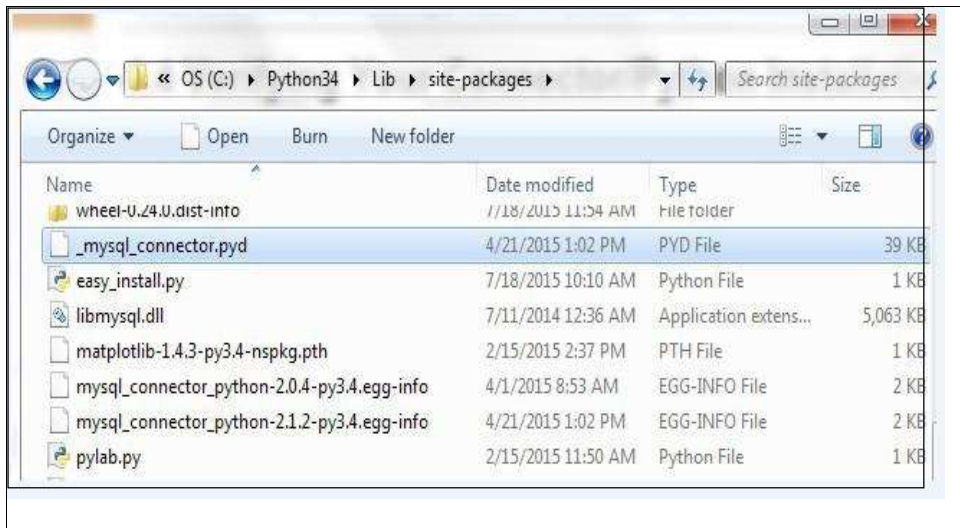
Fig. 1 Place of MySQL in drive folder

Fig. 2 After Installation of MysQL

15.3 CONFIGURING THE MYSQL CONNECTION, DESIGNING THE PYTHON GUI DATABASE

1. We used the shortest way to connect to a MySQL server by hard-coding the credentials required for authentication into the connection method. While this is a fast approach for early development, we definitely do not want to expose our MySQL server credentials to anybody unless we *grant* permission to databases, tables, views, and related database commands to specific users.

2. A much safer way to get authenticated by a MySQL server is by storing the credentials in a configuration file, which is what we will do in this recipe.

3. We will use our configuration file to connect to the MySQL server and then create our own database on the MySQL server.

4. First, we create a dictionary in the same module of thMe ySQL.py code.
   a. # create dictionary to hold connection info dbConfig = {
   b. 'user': <adminName>,   # use your admin name 'password': <adminPwd>, # use your admin password 'host': '127.0.0.1', # IP address of localhost
   c. }

5. Next, in the connection method, we unpack the dictionary values. Instead of writing,
   a. mysql.connect('user': <adminName>, 'password': <adminPwd>, 'host': '127.0.0.1')

6. we use(**dbConfig)   , which does the same as above but is much shorter.
   a. import mysql.connector as mysql # unpack dictionary credentials conn = mysql.connect(*

```
*dbConfig)
print(conn)
```

7. This results in the same successful connection to the MySQL server, but the difference is that the connection method no longer exposes any mission-critical information.

8. Now, placing the same username, password, database, and so on into a dictionary in the same Python module does not eliminate the risk of having the credentials seen by any one per using the code.

9. In order to increase database security, we first move the dictionary into its own Python module. Let's call the new Python modulG euiDBConfig.py .

10. We then import this module and unpack the credentials, as we did before.

11. import GuiDBConfig

    as guiConf # unpack

    dictionary credentials

    conn   =

    mysql.connect(**guiConf.dbConfig)

    print(conn)

12. Now that we know how to connect to MySQL and have administrator privileges, we can create our own database by issuing the following commands:

13. GUIDB = 'GuiDB'

    # unpack dictionary credentials

    conn =

    mysql.connect(**guiConf.dbCon

    fig) cursor = conn.cursor()

    try:

    cursor.execute("CREATE DATABASE {} DEFAULT CHARACTER SET 'utf8'".

    format(GUIDB))

    except mysql.Error as err:

    print("Failed to create DB: {}".format(err))

    conn.close()

14. In order to execute commands to MySQL, we create a cursor object from the connection object.

    A cursor is usually a place in a specific row in a database table, which

we move up or down the table, but here we use it to create the database itself.

15. We wrap the Python code into tary...except block and use the built-in error codes of MySQL to tell us if anything went wrong.

16. We can verify that this block works by executing the database-creating code twice. The first time, it will create a new database in MySQL, and the second time it will print out an error message stating that this database already exists.

17. We can verify which databases exist by executing the following MySQL command using the very same cursor object syntax.

18. Instead of issuing the CREATE DATABASE command, we create a cursor and use it to execute the SHOW DATABASE command, the result of which we fetch and print to the console output.

19. ..import mysql.connector
    as mysql import
    GuiDBConfig as
    guiConf
    # unpack dictionary credentials
    conn = mysql.connect(**guiConf.dbConfig)
    cursor = conn.cursor()
    cursor.execute("SHOW
    DATABASES")
    print(cursor.fetchall())

    conn.close()

20. Running this code shows us which databases currently exist in our MySQL server instance. As we can see from the output, MySQL ships with several built-in databases, such as information_schema , and so on. We have successfully created our owuidb database, which is shown in the output. All other databases illustrated come shipped with MySQL.

15.4 USING THE INSERT COMMAND

1. Creating New databases
1. import mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",p asswd = "google")
4. #creating the cursor object
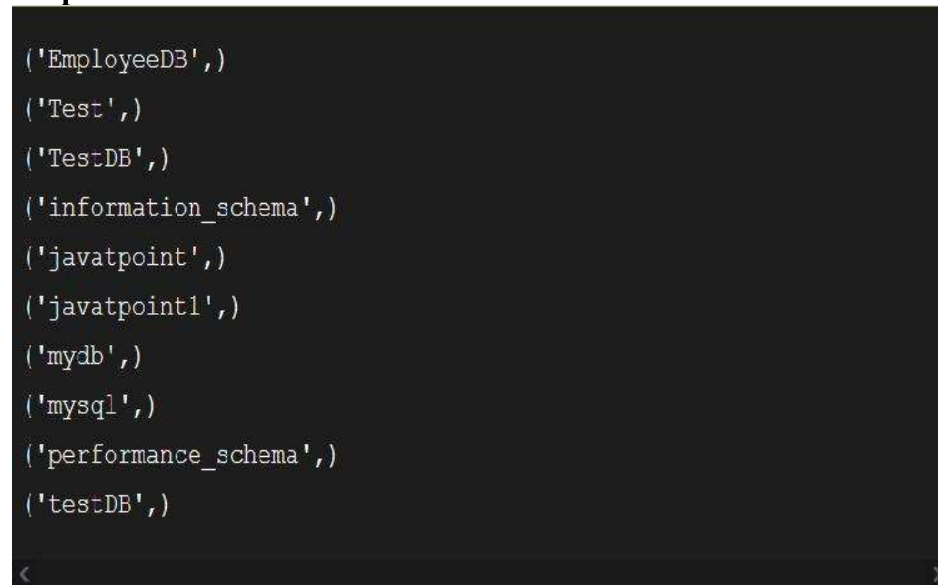
218

5. cur = myconn.cursor()

6. try:

7.   dbs = cur.execute("show databases")

8. except:

9.   myconn.rollback()

10. for x in cur:

11.   print(x)

12. myconn.close()

Output

```
('EmployeeDB',)
('Test',)
('TestDB',)
('information_schema',)
('javatpoint',)
('javatpoint1',)
('mydb',)
('mysql',)
('performance_schema',)
('testDB',)
```

Fig 3 Already Existing Output

1. 2. import mysql.connector

2. #Create the connection object

3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")

4. #creating the cursor object

5. cur = myconn.cursor()

6. try:

7.   #creating a new database

8.   cur.execute("create database PythonDB2")

9.   #getting the list of all the databases which will now include the new database PythonDB

10.   dbs = cur.execute("show databases")

11. except:

12.   myconn.rollback()

13. for x in cur:

14.　　print(x)

15. myconn.close()

output:


```
('EmployeeDB',)
('PythonDB',)
('Test',)
('TestDB',)
('anshika',)
('information_schema',)
('javatpoint',)
('javatpoint1',)
('mydb',)
('mydb1',)
('mysql',)
('performance_schema',)
('testDB',)
```

Fig 4 Created new database

3 Creating the table:

1. We will create the new table Employee. We have to mention the database name while establishing the connection object.

2.　　We can create the new table by using the CREATE TABLE statement of SQL. In our database PythonDB, the table Employee will have the four columns, i.e., name, id, salary, and department_id initially.

1. import mysql.connector

2. #Create the connection object

3. myconn = mysql.connector.connect(host = "localhost", user = "root",p asswd = "google",database = "PythonDB")

4. #creating the cursor object

5. cur = myconn.cursor()

6. try:

7.　　#Creating a table with name Employee having four columns i.e., na me, id, salary, and department id

8. dbs = cur.execute("create table Employee(name varchar(20) not null, i d int(20) not null primary key, salary float not null, Dept_id int not nul l)")

220

9. except:

10.   myconn.rollback()

11.   myconn.close()

Fig 5 Output of create Table

3. Insert Operation:

1. The **INSERT INTO** statement is used to add a record to the table. In python, we can mention the format specifier (%s) in place of values.

2. We provide the actual values in the form of tuple in the execute() method of the cursor

3. Consider the Following example

1. import mysql.connector

2. #Create the connection object

3. myconn = mysql.connector.connect(host = "localhost", user = "root",p asswd = "google",database = "PythonDB")

4. #creating the cursor object

5. cur = myconn.cursor()

6. sql = "insert into Employee(name, id, salary, dept_id, branch_name) v alues (%s, %s, %s, %s, %s)"

7. #The row values are provided in the form of tuple

8. val = ("John", 110, 25000.00, 201, "Newyork")

9. try:

10. #inserting the values into the table

11. cur.execute(sql,val)

12. #commit the transaction

13. myconn.commit()

14. except:

15. myconn.rollback()

16. print(cur.rowcount,"record inserted!")

17. myconn.close()

Fig 6 Insert Operation Output

3.1 Insert Multiple rows

1. We can also insert multiple rows at once using the python script. The multiple rows are mentioned as the list of various tuples.

2. Each element of the list is treated as one particular row, whereas each element of the tuple is treated as one particular column value (attribute).

•   import mysql.connector

#Create the connection object

myconn = mysql.connector.connect(host = "localhost", user = "root",pass
wd = "google",database = "PythonDB")

 #creating the cursor object
 cur = myconn.cursor()

sql = "insert into Employee(name, id, salary, dept_id, branch_name) value
s (%s, %s, %s, %s, %s)"
val = [("John", 102, 25000.00, 201, "Newyork"),("David",103,25000.00,2
02,"Port of spain"),("Nick",104,90000.00,201,"Newyork")]
    try:
    #inserting the values into the table
    cur.executemany(sql,val)

    #commit the transaction
    myconn.commit()
    print(cur.rowcount,"records inserted!")

 except:
    myconn.rollback()
    myconn.close()

Fig 7 Multiple Insertion Output

223
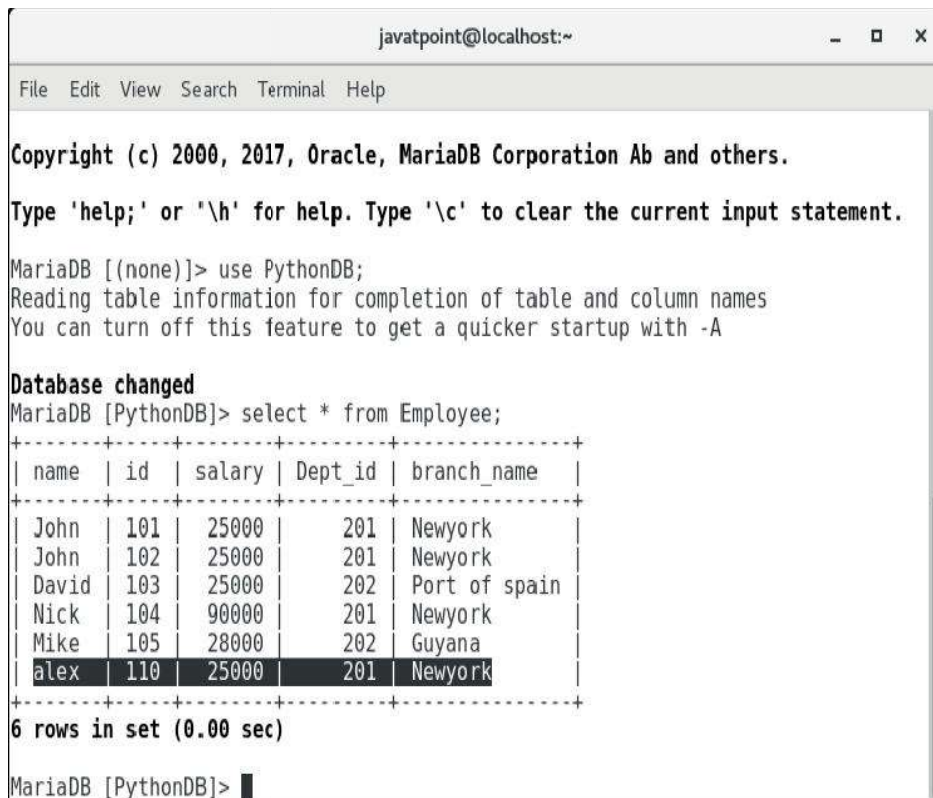
15.5 USING THE UPDATE COMMAND

The UPDATE-SET statement is used to update any column inside the table. The following SQL query is used to update a column.

1. import mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",p asswd = "google",database = "PythonDB")
4. #creating the cursor object
5. cur = myconn.cursor()
6. try:
7.   #updating the name of the employee whose id is 110
8.   cur.execute("update Employee set name = 'alex' where id = 110")
9.   myconn.commit()
10. except:
11.   myconn.rollback()
12.   myconn.close()

Fig 8 Update command output

224

15.6 USING THE DELETE COMMAND

The DELETE FROM statement is used to delete a specific record from the table. Here, we must impose a condition using WHERE clause otherwise all the records from the table will be removed.

The following SQL query is used to delete the employee detail whose id is 110 from the table.

```
import mysql.connector

  #Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

 #creating the cursor object
 cur = myconn.cursor()
 try:
    #Deleting the employee details whose id is 110
    cur.execute("delete from Employee where id = 110")
    myconn.commit()
 except:

    myconn.rollback()
    myconn.close()
```

15.7 STORING AND RETRIEVING DATA FROM MYSQL DATABASE.

1. The SELECT statement is used to read the values from the databases. We can restrict the output of a select query by using various clause in SQL like where, limit, etc.

2. Python provides the fetchall() method returns the data stored inside the table in the form of rows. We can iterate the result to get the individual rows.

```
import mysql.connector

 #Create the connection object

myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

 #creating the cursor object
 cur = myconn.cursor()

 try:
    #Reading the Employee data
    cur.execute("select * from Employee")
```

225

```
        #fetching the rows from the cursor object
        result = cur.fetchall()
        #printing the result

        for x in result:
            print(x);
    except:
        myconn.rollback()

    myconn.close()
```

2 Reading specific column

1   We can read the specific columns by mentioning their names instead
    of using star (*)

1. import mysql.connector

2. #Create the connection object

3. myconn = mysql.connector.connect(host = "localhost", user = "root",p
    asswd = "google",database = "PythonDB")

4. #creating the cursor object

5. cur = myconn.cursor()

6. try:

7.   #Reading the Employee data

8.   cur.execute("select name, id, salary from Employee")

9.   #fetching the rows from the cursor object

10.   result = cur.fetchall()

11.   #printing the result

12.   for x in result:

13.   print(x);

14. except:

15.   myconn.rollback()

16.   myconn.close()

```
('John', 101, 25000.0)

('John', 102, 25000.0)

('David', 103, 25000.0)

('Nick', 104, 90000.0)

('Mike', 105, 28000.0)
```

Fig 9 Select operation Output

15.8 SUMMARY

- In this chapter, CRUD (Create,Read, Update, Delete) operation is disccused along with example.

- Also chapter discusses the installation steps and configuring steps of MYSQL in python

15.9 QUESTIONS

1. Create a Library data base, perform CRUD Operations?
2. Discuss the steps of installation of Mysql in python
3. Why there is need to configure the Python Mysql Data base?

15.10 REFERENCES

1.Python GUI programming Cookbook -Burkahard A Meier, Packt Publication, 2nd Edition.

*****