2024

Introduction to Python Programming

Dr. Babasaheb Ambedkar Open University

Introduction to Python Programming

Course Writers

WASN Perera	Lecturer, Dept. of ECE, OUSL
AP Madurapperuma	Senior Lecturer, Dept. of ECE, OUSL
PTR Dabare	Lecturer, Dept. of Mechanical Eng, OUSL
HUW Ratnayake	Senior Lecturer, Dept. of ECE, OUSL
BK Werapitiya	Lecturer, Dept. of ECE, OUSL
S Rajasingham	Lecturer, Dept. of ECE, OUSL
NT De Silva	Project Assistant, Dept. of ECE, OUSL
GSN Meedin	Lecturer, Dept. of ECE, OUSL
MHMND Herath	Lecturer, Dept. of ECE, OUSL

Content Editor

Prof. (Dr.) Nilesh K. Modi	Professor and Director, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad
Mr. Nilesh N. Bokhani	Assistant Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad

Acknowledgement: The content in this book is modifications based on the work created and shared by the Open University of Sri Lanka (OUSL) for the subject Programming with Python used according to terms described under Creative Commons Attribution-Non Commercial-Share Alike 4.0 License

ISBN:

Printed and published by: Dr. Babasaheb Ambedkar Open University, Ahmedabad While all efforts have been made by editors to check accuracy of the content, the representation of facts, principles, descriptions and methods are that of the respective module writers. Views expressed in the publication are that of the authors, and do not necessarily reflect the views of Dr. Babasaheb Ambedkar Open University. All products and services mentioned are owned by their respective copyrights holders, and mere presentation in the publication does not mean endorsement by Dr. Babasaheb Ambedkar Open University. Every effort has been made to acknowledge and attribute all sources of information used in preparation of this learning material. Readers are requested to kindly notify missing attribution, if any.

Index

Unit	Topic	Page No.
	Block 1	
1	**Introduction to Basic Programming Concepts with Python**	5
	1.1 Need for programming languages	
	1.2 Programming languages	
	1.3 Examples of Programming languages	
	1.4 What is programming?	
	1.5 Getting to know Python	
	1.6 Features of Python Programming	
	1.7 Download, installation and run the first program with Python	
2	**Variables, Expressions and Statements**	25
	2.1 Data types and their values	
	2.2 Variables in Python	
	2.3 Differentiating variable names and keywords	
	2.4 Operators, Operands and Expressions	
	2.5 Interactive mode and script mode	
	2.6 Order of operations	
	2.7 Comments in Programs	
3	**Control Structures, Data structures-and Linked lists, queues**	39
	3.1 Selection Structure	
	3.2 Iteration structures	
	3.3 Data Structures	
4	**Functions**	55
	4.1 Why Functions	
	4.2 How to write a function definition in Python	
	4.3 Key things to remember when defining functions	
	4.3.1 How to call a function in Python	
	4.4 Flow of execution	
	4.5 Functions with arguments	

Unit	Topic	Page No.
	4.5.1 Different Argument types used in Python	
	4.6 Void Functions	
	4.7 Functions with return statements	
	4.8 Built-in functions	
	4.9 Type Conversion functions	
	4.10 Math functions	
	Block 2	
5	**Strings**	69
	5.1 Strings in Python	
	5.2 String Operations	
	5.3 String methods	
	5.4 Parsing strings	
6	**Object Orientation as a Programming Paradigm**	82
	6.1 Overview of object-oriented programming (OOP)	
	6.2 Basic concepts of objects and classes	
	6.3 Methods in Python	
	6.4 Operator overloading	
7	**Object Oriented Concepts**	94
	7.1 Object Oriented concepts	
	7.2 Inheritance	
	7.3 Multiple inheritance	
	7.4 Data Encapsulation	
	7.5 Polymorphism	
8	**Error and Exemption Handling**	103
	8.1 Errors	
	8.2 Semantic Errors	
	8.3 Exceptions	
	8.4 Exception Handling	

Unit	Topic	Page No.
	Block 3	
9	**Testing**	115
	9.1Software Testing	
	9.2Testing Methods: Black box and White box	
	9.3Testing Frameworks for Python	
	9.4Example of Python Unit Testing	
10	**Debugging and Profiling**	126
	10.1Finding and removing programming errors Software Testing	
	10.2Introduction to the profilers	
11	**Handling Data with Python**	139
	11.1What is a Database?	
	11.2Database Concepts	
	11.3Introduction to SQLite	
	11.4SQL CRUD statements	
	11.5 Introduction to database constraints	
12	**Role of Python in Mobile Application Development**	153
	12.1Mobile Application Development Environments	
	12.2Uses of Python in Mobile Application Development	
	12.3Open Source Python Libraries	
	12.4Kivy	
	Block 4	
13	**Mobile Application Development with Python**	168
	13.1Android Mobile Application Development using Kivy	
	13.2Buildozer	
	13.3Packaging with Python-for-android	
	13.4Packaging your application for the Kivy Launcher	
	13.5The Kivy Android Virtual Machine	
14	**Python Graphical User Interface development**	176
	14.1 Graphical User interface (GUI)	

Unit	Topic	Page No.
	14.2Different types of packages for GUI development in Python	
	14.3Tkinter (GUI toolkit that comes with Python)	
	14.4GUI with wxPython	
15	**GUI programming using kivy libraries**	204
	15.1 Basic GUI programming (user password GUI)	
	15.2 Kivy Layouts and Widgets	
	15.3 Kv language	
	15.4 Developing a Calculator using python and kivy	
	15.5 Develop a Calculator using python and kv language	
	15.6 GUI with check boxes	
	Appendix 1: Answers to Activities given in Python book	
	Unit-01	
	Unit-02	
	Unit-03	
	Unit-04	
	Unit-05	
	Unit-06	
	Unit-07	
	Unit-08	
	Unit-09	
	Unit-010	
	Unit-011	
	Unit-012	
	Unit-013	
	Unit-014	
	Unit-015	

Unit 1: Introduction to Basic Programming Concepts with Python

1

Unit Structure

1.1 Need for programming languages

1.2 Programming languages

1.3 Examples of Programming languages

1.4 What is programming?

1.5 Getting to know Python

1.6 Features of Python Programming

1.7 Download, installation and run the first program with Python

1.8 Python for Mobile App Development

Introduction

The purpose of this study unit is to give you the first step towards the programming language while motivating you to create useful, elegant and clever programs. Hence, you will be able to get the first steps towards turning yourself into a person who is skilled in programming.

The next part of this unit will help you to identify features of Python, which is a high-level programming language that is widely used in web development, mobile application development, analysis of computing, scientific and numeric data, creation of desktop GUIs, and software development. Furthermore, at the end of this unit we discuss about mobile application development using Python since it is regarded as one of the easiest programming languages being developed.

After studying this unit student should be able to:

Outcomes

- identify the importance of learning to write programs.
- explain the concepts of programming and the roles of a programmer.
- identify the features and the use of Python programming language.
- write down the basic steps of solving a given problem

Terminology

high level language : Any programming language like C, Java, Python, which is designed to be easy for programmers to remember, read and write.

Interpret : To execute a program written in a high level language by translating it line by line.

Compile : To convert a program written in a high level language into machine understandable form to be executed later

Source code :	A program written in a high-level language before compilation.
Program :	A set of instructions that specifies how to carry out a task.
Algorithm:	A general process, a set of step by step instructions for solving a problem.
Debugging :	The process of finding and correcting errors in a program.
Semantics :	study of meaning in words in any language including programming languages

1.1 Need for programming languages

We live surrounded by computers, mobile phones, tablet PCs and other digital devices. We can think of these digital devices as our “personal assistants” who can make our lives easy. It is said that the computers are built to continuously ask us the question, “What would you like me to do next?”

Writing programs could be a very creative and rewarding. You can write programs for many reasons like; as your day job, as a hobby, as a help to someone else to solve a problem etc.

Computers are so efficient and contain a large amount memory to perform different activities. If we have a mechanism or a language to instruct the computers (since computers can only understand the machine language) we can use them to perform/support our day to day activities. Interestingly, the kinds of things computers can do best are often the repititive tasks that humans find boring.

It is essential to master the languages that you can communicate with the computers (programming languages) so that you can delegate some of your work to the computer and save your time.

1.2 Programming languages

Only the machine language can be understood by the microprocessors, i.e. binary values (10101 series). There is a set of machine language instructions available for each microprocessor. These machine languages are also called low level languages. It is difficult to write lengthy instructions using binary language due to poor readability, understandability and the maintainability of binary instructions. Therefore, higher level languages have been created using machine language.

Assembly language is a low level language that is similar to the machine language and it uses symbolic operation code to represent the machine operation code. A high-level language is a programming language that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages

These languages are easy to read and understand. The generationof programing languages are shown in the Figure 1.1.

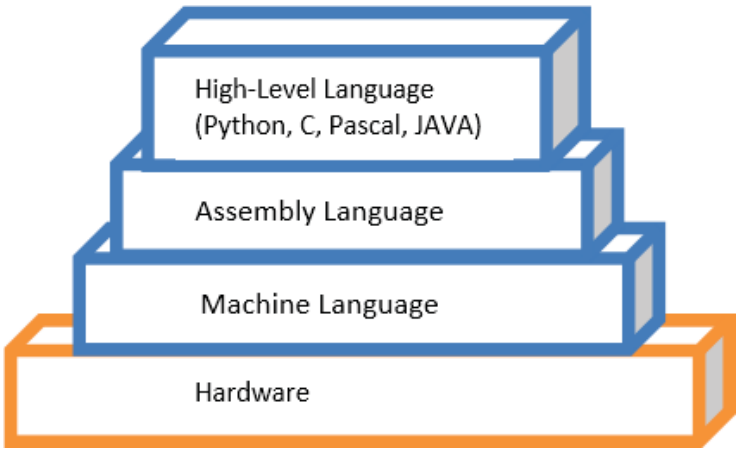


Figure 1.1: Generations of programming languages

A programming language like C is suitable for implementing programs in hardware level (on a micro-controller etc), while some high level languages like Java, C# are really good for large scale software development.

1.3 Examples of Programming languages

C language is a very popular programming language in the electronic and communication field. C is also used to program most microcontrollers. It is a general-purpose programming language initially developed by Dennis Ritchie. C has facilities for structured programming and its design provides constructs that map efficiently to typical machine instructions. C is one of the most widely used programming languages of all time. C compilers are available for the majority of computer architectures and operating systems.

C++ is a general purpose programming language. It has both structured and object-oriented programming features, while it also facilitates some hardware level programming such as low level memory manipulation. It is designed with a bias for systems programming (e.g. embedded systems, operating system kernels), with performance, efficiency and flexibility of use as its design requirements.

C++ has also been found useful in many other contexts, including desktop applications, servers (e.g. e-commerce, web search, SQL), performance critical applications (e.g. telephone switches, space probes) and entertainment software, such as video games.

Java - is a programming language that is concurrent, object- oriented, and specifically designed with less implementation dependencies

Java applications are typically compiled to byte code that can run on any Java virtual machine (JVM) regardless of computer architecture. Java is one of the most popular programming languages in use, particularly for the client-server web applications. Java was originally developed by James Gosling at Sun Microsystem's (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

Python is a dynamic object-oriented programming language that can be compared with Java and Microsoft's .NET-based languages as a general- purpose substrate for

many kinds of software development. It offers strong support for integrating with other technologies, higher programmer productivity throughout the development life cycle, and is particularly well suited for large or complex projects with changing requirements.

Python is also being used in mission critical applications in the world's largest stock exchange, forms the basis for high end newspaper websites, runs on millions of cell phones, and is used in industries as diverse as ship building, feature length movie animation, and air traffic control. It is a rapidly growing open source programming language. It is available for most operating systems, including Windows, UNIX, Linux, and Mac OS.

Hardware Description Language (HDL) is a specialised computer language used to program the structure, design and operation of electronic circuits, and most commonly, digital logic circuits.

A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit. It also allows for the compilation of a HDL program into a lower level specification of physical electronic components, such as set of masks used to create an integrated circuit.

A hardware description language looks much like a programming language such as C. It is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time. Two types of popular HDLs are VHDL and Verilog HDL.

Prolog is declarative programming language where the programmer specifies a goal to be achieved and the Prolog system works out how to achieve it. This language consists of a series of rules and facts. This is a frequently used language in Artificial Intelligence as this is a logic language that is particularly suited to programs that involve symbolic or non-numeric computation.

VisiRule is a graphical tool for designing, developing and delivering business rule and decision support applications, simply by drawing a flowchart that represents the decision logic.

In computer programming' a language or set of commands/instructions that describe the actions are required. Using these instructions machines that can perform a number of different tasks. In order to write instructions to perform something, each action must have a precise, unambiguous meaning. Therefore, all programming languages intended to manage the process of converting the human requirements to a computer solution. A programmer has to play a vital role in establishing the above goals in a successful manner.

Activity

Activity 1.1

List four reasons to choose Python as first language. You may need the help of an Internet search engine to attempt this activity.

Feedback: This is a peer-reviewed activity where the learner should share their findings with other learners in class.

1.4 What is programming?

There are two main skill you need to acquaire to be a programmer.

- You need to know the syntax of programming language with its concepts and keywords. Only then you would be able to write instructions in that language.
- You must also learn to solve a problem logically. If you know how to come to a solution logically, then you can write an algorithm for it, which can be coded in any programming language.

Every programming language has different vocabulary and grammar (syntax) but the concepts and problem-solving skills are universal across all programming languages.

Here, you will learn the vocabulary and grammar of Python language which will enable you to write a program to solve any problem.

1.4.1 Words and sentences

Just like any other programming language, Python vocabulary is actually very small. This is in contrast to the huge vocabulary in any natural language. This small "vocabulary" is called the set of "reserved words" (key words) in python. These words contain very special meaning to the Python. When Python sees these words in a program, they have a unique meaning to Python. Later as you write programs you will make up your own words that have meaning to your program called variables. You have many options in choosing names for your variables, but you cannot use any of Python's reserved words as a name for a variable.

When we train a dog, we use special words like "sit", "stay", and "fetch". When you talk to a dog and do not use any of the reserved words, they just look at you with a quizzical look on their face until you say a reserved word.

For example, if you say, "I wish more people would walk to improve their overall health", what most dogs likely hear is, "blah blah blah walk blah blah blah blah." That is because "walk" is a reserved word in dog language. Many suggest that the language between humans and cats has no such reserved words.

The reserved words in the language where humans talk to Python include the following:

And	from	not	while
Del	elif	global	with
As	or	else	if
assert	pass	yield	break
except	import	print	class
Exec	in	raise	continue
finally	is	return	def
For	lambda	try	

Unlike a dog, Python is already completely trained. When you say "try", Python will try every time you say it without fail.

We will learn these reserved words and how they are used, gradually on our way through the course. Now, let us focus on the Python equivalent of "speak". A very simple example of how Python speak is given below:

print 'Welcome to the world of programmers!'

1.4.2 Understanding Interpreters and Compilers

Python is a **high-level** language intended to be easy for humans to read, write and understand when solving problems with computers. The actual

hardware inside a computer does not understand anything written in high- level languages. (Not only Python but all the other high-level languages such as Java, C++, PHP, Ruby, Pascal, JavaScript, etc)

The Central Processing Unit (CPU) inside a computer, understands only **machine language** or **binary code**. Machine language seems quite simple given that there are only zeros and ones in its vocabulary, but its syntax (set of rules for writing a program) is very complex than any high level language. Therefore, instead of writing programs in machine language, we use various translators so that programs can be written in high-level languages like Python, C or Java and then translators can convert these to machine language for actual execution by the CPU.

Since machine language depends on its computer hardware, it is not **portable.** Therefore, a program written in a high-level language need to be translated to a specific machine language using an interpreter or a compiler, for it to be excuted in any machine.

There are two types of Programming language translators:

- Interpreters
- Compilers

An **interpreter** reads the source code, parses (decompose and analyse) the source code, and interprets the instructions line by line at run-time. Python is an interpreted programming language where we can run the source code interactively. When a line in Python (a sentence) is typed, Python interpreter processes it immediately and we can type another line of Python code.

A **compiler** takes the entire source program as the input, and runs a process to translate the souce code of a high-level language into the machine language. As a

result, compiler creates the resulting machine language instructions which can be used for later execution. The compiled output is called an executable file.

Since it is not easy to read or write a program written in machine language, it is very important to have interpreters and compilers that allow us to translate the programs written in high-level languages to machine languages.

1.4.3 Programs

A program is a collection of instructions that perform a specific task when executed by a computer. Then the computer behaves in a predefined manner. The tasks performed might be something mathematical, but it can also be a symbolic computation or compiling a program. A computer program is written by a computer programmer in a programming language. A program is like a recipe which consists of list of ingreddients (variables) and a list of directions (statements) that tell the computer what to do with the variables.

A few basic instructuons can be found in many programming languages which are not just for Python programs, they are part of every programming language from machine language up to the high-level languages.

input: It collects whatever inputs the program needs to accomplish a task and get it via the keyboard, from a file, or some other device.

output: This gives the answer to the user and display relevant data on the screen, send data to a file etc.

math: This uses mathematical operations such as addition, substraction etc.

conditional execution: There are some condtions and check them and execute the appropriate code

repetition: Write some instructions once and can use it again and again usually with some variation

program logic/content: content of the program which enables the final solution and it may contain the following parts

Any program (complicated or simple) is made up of instructions that look like these. In programming, you can break a large problem in to small sub problems. Those small sub problems could be able to perform using the identified steps above. How this decomposition is done, we will revisit when discussing algorithms.

1.4.4 Correcting errors

Any program can have errors. It is very difficult to write even a small program without errors. It is the responsibility of the programmer to make sure that the code he writes is error free. Programming errors are called bugs and the process of finding them and correcting them is called debugging.

From early days of programming, it was identified that there are three types of defects possible in programs. These are described in detail below.

1. **Syntax errors**

   Just like any other programming language, Python can execute a program only if it is written in correct syntax i.e. in the correct grammar of the language. If the syntax is wrong, the interpreter displays an error message indicating where the error is.

   e.g. print (x) if written as print(x without the closing bracket, a syntax error will be displayed

   Until you get used to Python syntax, you will have to spend a considerable amount of time correcting syntax errors.

2. **Runtime errors**

   Runtime errors will appear once the program start running. These are also called Exceptions. In simple programs runtime errors are not very common.

3. **Semantic errors**

   Semantic errors are the logical problems in a program. If there is a semantic error, most probably the program will not give an error message but will give a wrong output.

e.g. a never ending running program means it is repeating without a terminating condition.

You will learn more about errors, exception handling and testing in later Units as well.

Indentation and comments in Python

Many languages arrange program instructions into blocks using curly brackets { } or BEGIN and END keywords. These languages also encourage programmers to indent blocks to make the programs readable though indentation is not compulsory. However, Python uses only indentation to delimit blocks, which makes it mandatory to indent program instructions.

Hash (#) symbol is used to write a comment in Python which is used to increase the readability of the program. Comments are not executed/interpreted and used for the purpose of explaining the instructions for the developers and other users.

```
# new block starts with function definition
def add_numbers(a, b):

    # statements are inside this block
    c = a + b return c

# an if statement which starts a new block
if (X > 10):

    # one statement inside this block
    print("X is larger than 10")

    # this is outside the block print("Out
of if' statement")
```

In many languages we use special characters like semicolon (;) to mark the end of each instruction. Python examines only ends of lines to determine whether instruction

```
# individual instructions
print("Hello World!")
print("Here's a another new instruction") a = 2

# This instruction spans more than one line
b = [1, 2, 3,
    4, 5, 6]

# Following is allowed but recommended to avoid
c = 1; d = 5
```

has ended. Sometime, if we want to put more than one instruction on a line, we can use a semicolon.

1.4.5 The Programming Process

The primary concern of programming is to solve a problem which can range from great scientific or national importance, to something as trivial as relieving personal boredom! Here we discussed a basic approach to solve such problems which consists of the following steps:

1. Identify the Problem
2. Design a Solution
3. Write the Program
4. Check the Solution

Identify the Problem*:*

In the process of identifying the problem first we need to collect the requirements of the given scenario and then analyse the gathered requirements to identify the problem. In the requirement stage, you are trying to work out exactly what your program will be required to do. The next step, which is analysis, looks at the list of requirements and decide exactly what your solution should do to fulfil them. As there are various solutions to a single problem, here your aim is to focus on only the solution that you have selected.

Design a Solution :

This stage focuses on how you're going to turn the previously identified specification into a working program. A design is simply a higher-level description of a list of steps instructing the computer what it should do. This stage does not depend on any special programming language. Usually, special notations like pseudocode or flowcharts are used in the design stage to illustrate problem solution. This step helps a programmer to take the design as an initial step to build a computer program.

Write the program

The three stages of writing a program are Coding, Compiling and Debugging. Coding is the act of translating the design into an actual program using some form of programming language. Compilation is the translation of source code which is written in some programming language into the machine code which can be understood by the processor. Debugging is the process of finding and resolving of defects that prevent correct operation of computer software or a system

Solution

This step is very important where it tests your creation to check that it does what you wanted it to do. This step is necessary because although the compiler has checked that the program is correctly written, it cannot check whether what you've written actually solves your original problem.

1.5 Getting to know Python

Python is a popular high-level programming language used for general- purpose programming, created by Guido van Rossum and first released in 1991. It has wide range of applications such as web development, scientific and mathematical computing, network programming, desktop graphical user Interfaces etc.

About the origin of Python, Van Rossum wrote in 1996:

“Over six years ago, in December 1989, I was looking for a ‘hobby’ programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I

decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)."

Activity	Activity 1.2
	Find applications of Python in different areas. You may need the help of an Internet search engine to attempt this activity. **Feedback:** This is a peer-reviewed activity where the learner should share their findings with other learners in class.

1.6 Features of Python Programming

Python's features include :

- Python is intended to be a simple and highly readable language. It is designed to have an uncluttered visual layout, often using English keywords where other languages use punctuation.
- It is a free and open-source software where you can freely use and distribute even for commercial use.
- It ensures portability by providing you to move a Python program from one platform to another, and run it without any changes.
- It is a high-level and interpreted language where you don't have to worry about memory management, garbage collection and so on.
- A large number of standard libraries are available to solve common tasks, which makes life of a programmer much easier since you don't have to write all the code yourself.

1.7 Download, installation and run the first program with Python

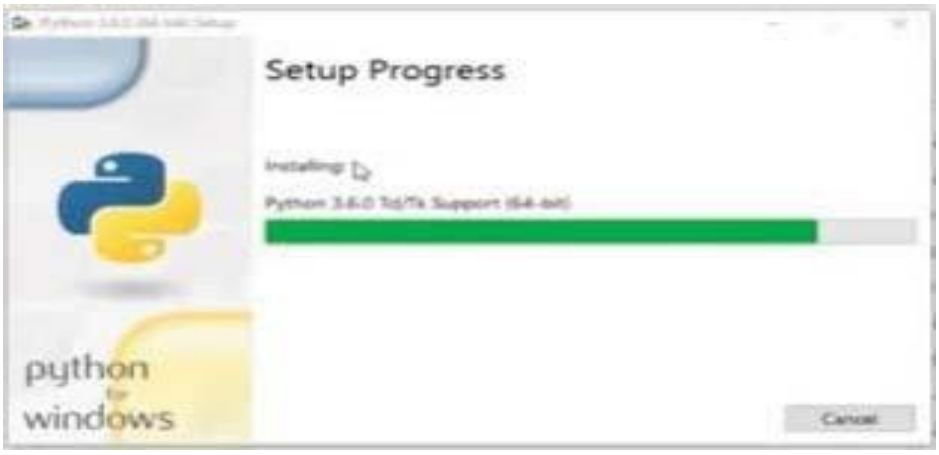
1. First, go to the https://www.python.org and download Python on your computer. Click on Python Download.

Video 2:
How to download and install Python

You may watch this video with screen cast to see how to install Python in your computer. Then install Python in your computer and get ready for coding.
URL: https://youtu.be/T1JBsYZqbOY

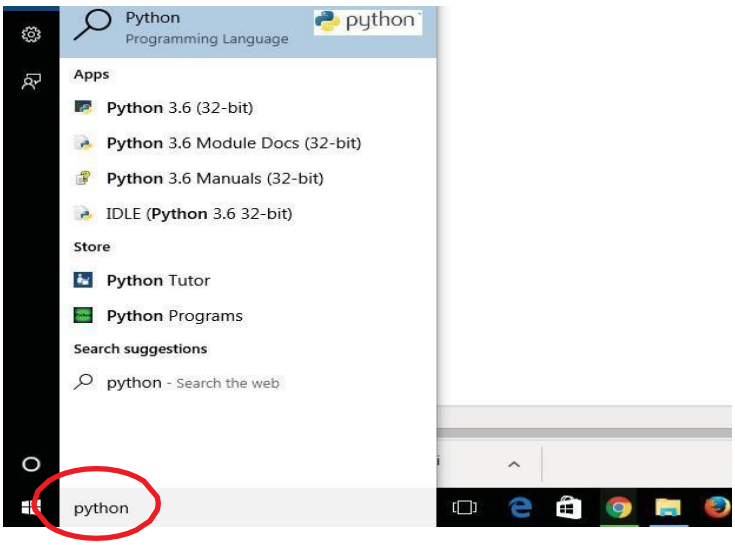
2. After downloading, click on the downloaded file. Select “Install now” and select the check box to add Python to the path.

3. Wait till the installation process complete.

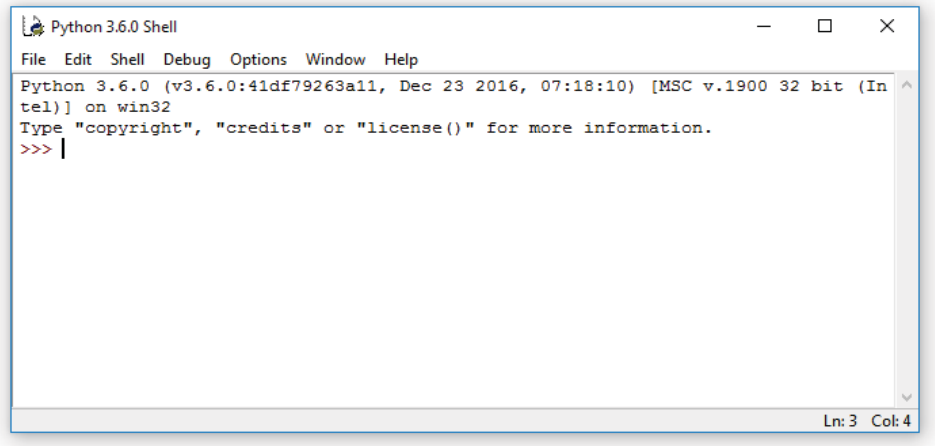


4. The setup is complete now. Click on close.

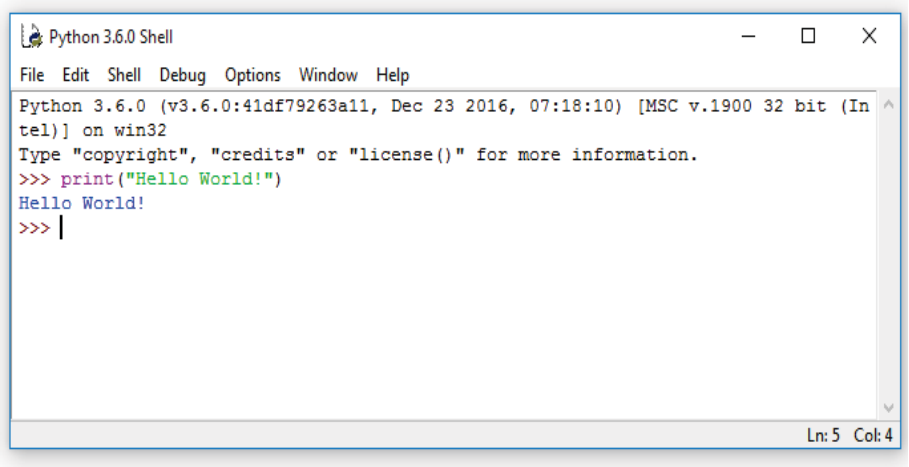
5. Search for Python IDLE console for your first program from the start menu.



6. Now you can write your own Python script.



7. Write your first program with Python.



To run a program with more than one line you have to write the program in the editor in Python IDLE. A program with more than one statement is called a script. Then save it in the dirctory that you would use to save all programs. It will be saved as a *.py for example myFirstProg.py file.

You can run the script from Windows command prompt typing python myFirstProg.py if the path to Python.exe file is already set. To set the path you have to go to Window settings ‹ environment ‹ path and add the new path for example something like c:\user\prog\python36-32\python\ Similary for Unix and AppleOS also the path to python need to be set before executing a program.

As a script is a sequence of instructions or statements, as and when a statement is excuted, the result appears one after the other. Write following set of instructions in an editor and run it to see the result

```
print('Helloworld!')
x =2020

print(x)


output
```

Note that the assignment statement produces no output. We will discuss scripting again in Unit 2.

Activity Activity 1.3

Download and install Python as described above. Then set the path and type the given example script and run it.

1.8 Python for Mobile App Development

Python is a high-level programming language that is widely used in mobile application (commonly called 'app') development. Kivy is a new library that Python programmers can use to program apps. It runs on iOS, Android, MacOS, Windows and Linux. As Python is Object-oriented, it helps to solve a complex problem intuitively. This provides you the ability to divide these complex problems into smaller sets by creating objects.

Unit summary

Summary	In the first part of this unit, we discussed basics in programming and the necessary skills to be a programmer. The second part of this unit is about the types of programming languages which includes C, C++, Java, Prolog etc. Further, it explains the basic concepts of programming related to Python programming language. Then we discuss about how to download, install and run the first program with Python. At the end of this unit you were introduced to using Python for Mobile App Development
References and Further Reading	
	Allen B. Downey (2012). Think Python. Retrieved 20 Dec2016, http://greenteapress.com/wp/think-python/ Download this book for free at http://greenteapress.com/thinkpython/thinkpython.pdf Python Basics. © Copyright 2013, 2014, University of Cape Town and individual contributors. This work is released under the CC BY-SA 4.0 licence. Revision 8e685e710775. https://python-textbok.readthedocs.io/en/1.0/Python_Basics.html
Attribution	Some content of this unit – Unit 1, are taken from 'Python for Everyone (PY4E)' site https://www.py4e.com/materials Copyright Creative Commons Attribution 3.0 - Charles R. Severance

Unit 2: Variables, Expressions and Statements

2

Unit Structure

2.1 Datatypes and their values

2.2 Variables in Python

2.3 Differentiating variable names and keywords

2.4 Operators, Operands and Expressions

2.5 Interactive mode and script mode

2.6 Order of operations

2.7 Comments in Programs

Introduction

In this unit we will discuss about one of the most powerful features of any programming language, which is the ability to manipulate variables. Further it focuses your attention towards statements and expressions. A statements is an instruction that the Python interpreter can execute and an expression is a combination of values, variables, operators, and calls to functions.

Outcomes

- explain identifiers, variable, constants, assignment and expressions used in Python.
- identify basic concepts of input and output .
- apply string manipulation techniques in python.

Terminology

value: a unit of data such as a number, set of characters etc

variable: A memory location that holds a data unit or a value.

2.1 Datatypes and their values

In any programming language we find values of different data types. Value is an important concept in programming. It could be a letter, a string of characters or a number. Since we relate values to a certain data types, let us look at few examples.

There are different types of values such as:

Integers (int):

```
>>> x = 1
```

Floating point values (float):

```
>>> x = 4.3
```

Complex values (complex):

```
>>> x = complex(4., -1.)
```

String:

>>> x = "Hello World"

Python supports all primary data types such as integers, floating point numbers and strings. Other than those, it has built-in support for data types such as lists, tuples, dictionaries and complex numbers.

Python interpreter can find the type of a given value using ‘type’ keyword.

```
>>>type('Hello World !')
<type'str'>
>>>type(20)
<type'int'>
```

Python supports a number of built-in types and operations. Though different languages may handle variable types uniquely, most are quite similar. Python variable types, like the rest of the language, are quite simple. Variable types are determined by the value stored within the variable. Unlike most other languages, keywords like "int", "String", or "bool" are not required in Python, as Python supports type inferencing. The most general compatible variable type is chosen.

2.2 Variables in Python

A variable is something that holds a value which may change later. In simplest terms, a variable is just a box that you can put stuff in. You can use variables to store all kinds of stuff, but for now, we are just going to look at storing numbers in variables. Ability to manipulate variables is a versatile feature in any programming language.

Use of assignment statement (‘= ‘ operator) in Pthon, creates new variables and gives them values as well:

```
>>> student_name = 'Senuri Gamage'
>>> student_no = 5
>>> student_average = 47.561
```

In the above example:

The first assigns a string to a new variable named student_name.

The second gives the integer 5 to a new variable named student_no.

The third assigns an approximate value for student_average.

A state diagram as shown below can be used to represent variables in a paper. Variable's current value shows what state each of the variables is in. Below shows the result of the previous example.

```
student_name -> 'Senuri Gamage'
student_no -> 5
student_average -> 47.561
```

The assignment ('=') statement links a name, on the left hand side of the operator, with a value, on the right hand side. This is why you will get an error if you enter:

17 = n

When reading or writing code, say to yourself "n is assigned 17" or "n gets the value 17" or "n is a reference to the object 17" or "n refers to the object 17". Don't say "n equals 17".

Unlike many other high level languages, variable types are not declared before assigning values. The type of a variable is decided by the type of the value it is initialized or later refers to. For example;

```
>>>type(student_name)
output
<type 'str'>
>>>type(student_no)
output
 <type 'int'>
 >>>type(student_average)
 output
 <type float >
```

2.3 Differentiating variable names and keywords

Generally, any programmer is taught to choose meaningful names for variables which helps human readers to understand what they are used for. You may construct variable names to represent what the variable is used for.

Naming Conventions of the variables in Python

Variable names can be quite long and can have both English letters and numbers. However, the variable name must start with a letter. It is adviced to start with a lowercase letter. The underscore character "_" can appear in a name where it can connect multiple names such as student_name and student_number. Variable names are case sensitive. Case sensitivity example: myVariable, myvariable, and Myvariable represent three separate variable names.

If your variable has an illegal name, it will cause a syntax error:

```
>>> 3subjects_average = 57
SyntaxError: invalid syntax

>>>class = 'Botany'
Syntax Error: invalidsyntax
```

3subjects_average is illegal because it does not begin with a letter. It

seems that class is wrong too. Can you guess why?

class is keyword in Python. Keywords define the language's syntax rules and structure, and they cannot be used as variable names.

There are 31 keywords in Python version 2 as shown below.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

Keywords may change with the new versions and in Python version 3, exec is not considered as a keyword. Instead, a new keyword nonlocal is added.

When you come across compilation errors with variable names, and it is not clear why, see whether it is on this list of keywords.

2.4 Operators, Operands and Expressions

Symbols that represent some actions are called Operators. The action could be addition of two numbers. The operator performs an action on operands such as two numbers.

The operators use +, -, *, / and % to perform computations Addition, Subtraction, Multiplication, Division and Modulus. See the blow examples.

```
>>> x = 2
>>> y = 3
>>> z = 5
>>>x * y 6
>>>x * y + z 11
>>> (x + y) * z 25
```

Division operation results in different results for Python 2.x (like floor division) and Python 3.x. Floor division means after performing the division, result is given as the lower integer to the value. Therefore, in Python 2, when both of the operands are integers, the result is truncated to be an integer.

In Python 2:

```
>>> 10 / 3
3
```

In Python 3:

```
>>> 10 / 3

3.3333333333333335
```

In Python 3, the result is given as type float.

If any one of the operands is a floating-point number, Python performs floating-point division, and the result is given as type float:

```
>>> 5.0 / 2
2.5
```

In programming, an expression is any legal combination of symbols that represents a value. The expression is consists with values, variables, and operators. In the domain of computing, expressions are written by developers, interpreted by computers and evaluated.

In the expression,

x + 3

x and 3 are operands

+ is an operator

An expression does not necessarily do anything, it evaluates to, that is, reflects, a value. For example, 3 is an expression which evaluates to 3. Following are examples for expressions assuming that the variables has been assigned a particular value.

8888

x

x + 88

One physical line of code is considered as one statement which does something. Most often, one physical line of code will correspond to one statement. Within a statement you can find expressions. A statement is an instruction that the Python interpreter can execute.

2.5 Interactive mode and script mode

There are certain advantages that can be achieved with an interpreted language compared to compiled one. Here, you can test line by line in interactive mode before you write few lines together in a script. However, there are some differences with Interactive and Script mode.

For example,

```
>>>minutes = 25
>>>minutes * 60 1500
```

The first line, the variable minutes represent a value. However, it does not have a visible effect. The second line is an expression where the python compiler is able to interpret. Therefore, it displays the result in seconds for the input value of 25 minutes.

If same lines are typed into a script and executed, you cannot expect the same output. That is because these expressions have no visible effect. However, Python does evaluate all statements but do not display result unless asked with a print statement.

```
minutes = 25
```

Video 3:

Data Types in Python

You may watch this video first and then attempt activity 2.1.

URL: https://youtu.be/F_TWZi4rg-0

Activity

Activity 2.1

Write following statements one by one to see what they do:

55
x = 55
x + 1

Then write all 3 statements in notepad as a script, run it and observe the output.

Finally, change all expressions into a print statement and then run the script.

2.6 Order of operations

In mathematics and computer programming, the order of operations (or operator precedence) is a collection of rules that reflect conventions about which procedures to perform first in order to evaluate a given mathematical expression. Python uses the standard order of operations as taught in Algebra and Geometry classes at high school or secondary school. That is, mathematical expressions are evaluated in the following order (memorized by many as PEMDAS), which is also applied to parentheticals.

Note that operations which share a table row are performed from left to right. That is, a division to the left of a multiplication, with no parenthesesbetween them, is performed before the multiplication simply because it is to the left.

The following Table 2.1 shows the operator precedence in Python from lowest to highest. Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature

Table 2.1: Operator precedence

Operator	Description
lambda	Lambda Expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, !=, ==	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, %	Multiplication, Division and Remainder
+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation
x.attribute	Attribute reference
x[index]	Subscription
x[index:index]	Slicing
f(arguments ...)	Function call

(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
{key:datum, ...}	Dictionary display
`expressions, ...`	String conversion

Operator precedence affects how an expression is evaluated. For example, if a = 20, b = 10, c = 15 and d = 5 the value of the following expressions are:

x = (a + b) * c / d; The value of x is 90

x = a + (b * c) / d; The value of x is 50

2.7 Comments in Programs

There will always be a time in which you have to return to your code. Perhaps it is to fix a bug, or to add a new feature. Regardless, looking at your own code after six months is almost as bad as looking at someone else's code. What one needs is a means to leave reminders to yourself as to what you were doing.

For this purpose, you leave comments. Comments are little snippets of text embedded inside your code that are ignored by the Python interpreter. The natural language can be used for commenting the code. It can appear anywhere in the source code where whitespaces are allowed. It is useful for explaining what the source code does by:

- explaining the adopted technical choice: why this given algorithm and not another, why calling this given method...
- explaining what should be done in the next steps (the TODO list): improvement, issue to fix...
- giving the required explanation to understand the code and be able to update it yourself later or by other developers

It can also be used to make the compiler ignore a portion of code: temporary code for debugging and code under development.

The following guidelines are from PEP 8 (Index of Python Enhancement Proposal), written by Guido van Rossum.

- Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!
- Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).
- Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.
- You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence.
- When writing English, follow Strunk and White.
- Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious.

```
x = x + 1         # Increment x
```

Don't do this:

But sometimes, this is useful:

```
x = x + 1        # Compensate for border
```

Activity

Activity 2.1

Write following statements one by one to see what they do:

55
x = 55
x + 1

Then write all 3 statements in notepad as a script, run it and observe the output.

Finally, change all expressions into a print statement and then run the script.

- 13 + 25 * 10

Check your answers using Python interpreter.

Let us try solving a different kind of a problem now. How do you solve an equation in Python?

e.g. if volume of a sphere with radius r is given as 4/3 nr^3. What is the volume of a sphere with radius 10? (Assume n = 3.14)

>> volume_sphere = 4/3*3.14 *10***3

Activity

Activity 2.3

Volume of a cylinder with height h and radius r is given as nr2h. What is the volume of a cylinder with height 6 and radius 2?

Unit summary

Summary	The first part of this study unit focused on a very important feature of Python language, which is about variables and their types. Moreover, it discussed about variable naming conventions and Python's keywords. We also discussed about statements which is a 'line of code' that the Python interpreter can execute. A statement could be one or many expressions. An expression is a combination of operands and operators. Furthermore, we were able to identify the types of operators which are supported for Python language.

References and Further Reading

1. https://www.py4e.com/materials
   © Copyright 2013, 2014, University of Cape Town and individual contributors. CC BY-SA 4.0 licence. Revision 8e685e710775. Copyright CC-BY 3.0 - Charles R. Severance
2. https://python-textbok.readthedocs.io/en/1.0/index.html
3. http://opensask.ca/Python/Overview/VariablesAndDataTypes.html

Unit 3: Control Structures, Data structures-and Linked lists, queues

3

Unit Structure

3.1 Selection Structure

3.2 Iteration structures

3.3 Data Structures

Introduction

In a program, a control structure is a block of code that determines the order in which the statements are executed. It also directs the flow of the program based on the given logic. There are two key control structures, namely, selection and iteration to structure the segments of code. Selection statements allow a programmer to write statements that will be executed based on the satisfaction of some condition while iteration statements allow the repetition of the same set of statements multiple times.

A data structure is a particular way of organising data in a computer that enables efficient use. In this unit we will also discuss common data structures in Python.

Upon completion of this unit you will be able to:

Outcomes

- explain the use of control structures and data structures in a program.
- identify appropriate control structures and data structures for a given scenario.
- apply suitable data structures to model a solution for a simple problem.

Terminology

iterative: repeating

slice: part of a datastructure

string: a special datatype

3.1 Selection Structure

As in many other languages, Python provides an if ….else statement. the general syntax of the if … else statement is;

if<condition>:
<Statements to be run if the condition evaluates to true>
else:
<Statements to be run if the condition evaluates to false>
For example, the following program prints a warning message if the weight of a customer's luggage exceeds the allowed limit of 25 kgs.

Example 3.1:

```
allowedlimit = 25
extraweight=0
weight = float(input("How many kilograms does your" "baggage weigh? "))
if weight > 25:
        extraweight = weight - allowedlimit
        print('Your luggage has ',extraweight,'kgs extra') print('There will be an extra
        chrage of',extraweight*10)
else:
        print("Your luggage weight is within the limit")
print("Thank you for your business.")
```

The above program checks if the weight is above the allowed limit. If it above the limit, it informs the customer the extra payment involved. Otherwise, it will inform that the weight is within the limit. Else clause is used to execute involved in the 'otherwise' part of the condition.
If statements can be embedded within one another and are called nested if condition. The level of nesting depends on the logic of the program and you can have any

number of nested statements. For example, the following modifies the example 3.1 to allow customers with a small extra weightage to go through without a payment.

Example 3.2

```
allowedlimit = 25
extraweight=0
weight = float(input("How many kilograms does your baggage weigh? "))
if weight > allowedlimit:
        extraweight = weight - allowedlimit if extraweight <5:
        print("Your luggage has ", extraweight, "kgs extra") print("Since it is a small
        value, we will not charge you this time")
else:
        print("Your luggage has ", extraweight, "kgs extra")
        print("There will be an extra chrage of ",extraweight*10,"\n")
else:
        print("Your luggage weight is within the limit")
        print("Thank you for your business.")
```

In some cases, we need to do things based on multiple tests. For example, let us say a teacher decided to give grades to her students as follows:

Mark greater than or equal to 80	Excellent
Mark greater than or equal to 65 but less than 80	Good
Mark greater than or equal to 50 but less than 65	Pass
Mark less than 50	Fail

We can write a program to print a grade according to a student's mark with multiple if statements as given in example 3.3a.

Example 3.3 a	Example 3.3 b

#Program: Printing grades
mark = float(input("Enter the students mark"))
if mark >= 80:
grade = "Excellent" else:
# check grades for the condition Good 	
if mark >= 65:
grade = "Good"
else:
# check grades for the condition Pass 	
if mark >= 50:
grade = "Pass"
else:
# check grades for the condition Fail 	
grade = "Fail"
print("Your grade for this" "subject is:", grade)
	#Program: Printing grades
	mark = float(input("Enter the students mark"))
	if mark >= 80:
	grade = "Excellent"
	elif mark >= 65:
	grade = "Good"
	elif mark >= 50:
	grade = "Pass"
	else:
	# grade has to be fail grade = "Fail"
	print("Your grade for this" "subject is:", grade)

Python provides another statement called elif, which allows us to write more compact code with less indentation as given in example 3.3b. elif is a short hand for else if. In all of the above else part is optional.

3.2 Iteration structures

Python provides multiple statements to implement iteration, i.e. repeating the same set of instructions with different value sets. while statement and for statement are two such key statements.

3.2.1. While statement

In a while statement executes a set of instructions if a given condition is true. It takes the form of

while<condition>

<set of statements>

Only if the condition is true, then the set of statements is executed. This is how the while loop operates:

Step 1: Evaluate the condition

Step 2: If the condition is false continuing the program with the next statement that comes after the set of statements within while condition

Step 3: If the condition is true execute the set of statements within while condition

Step 4: Go back to step 1 and evaluate the condition again

The following is an example of the use of the while loop: It prints numbers from 1 to 9 (last number not less than 10). The condition n < 10 will be true for all numbers from 0 to 9. the condition will become false when n becomes 10, because 10 is not less than 10).

```
#while loop n = 0
while n < 10:
        n = n + 1
        print(n)
        print('All numbers printed')
```

As you can see, it is important that when the condition is true, it changes sometime during the execution of the set of statements. Otherwise, it will run for ever (known as an infinite loop). In the above example, if we forget to increase n by 1 during each iteration, it will become an infinite loop.

3.2.2 statement

Unlike other languages, Python for statement is not based on a counter variable. Instead, for loop iterates through a set of values given as a list or a string. The syntax of the for loop is

for<item> in <list>:

<do something with the item>

The loop will iterate through the list or the string assigning one item or character to the control variable, then executing the block of statements using that value.

```
import math
numberlist = [1,2,3,4,5,6]
for targetvalue        in numberlist:
        print(targetvalue, "squared is",int(math.pow(targetvalue,2)))
```

In the example, control variable is named targetvalue. Starting with the first value in the list, it will take values in the list in the given sequence. In this example, it will take values 1,2,3,4,5 and 6 in respectively. During each iteration it will print the square value of the control variable.

Video 4:

Control Structures in Python

You may watch this video on Control Structures in Python Before attempting the Activity 3.1.

URL: https://youtu.be/5ogPFfF8wGw

Activity

Activity 3.1

Write a program using a while loop to execute even numbers starting from 20 to 60

3.2.3 break,continueand pass Statements

The break statement ends execution of the nearest enclosing loop or conditional statement in which it appears. It can be used to terminate or change the ongoing iteration. break statement is commonly used with the looping statements. In while and for loops, it terminates the nearest enclosing loop, skipping the optional else clause if the loop has one. If a for loop is terminated by break, the loop control target keeps its current value. When break passes control out of a try statement with a finally clause, that finally clause is executed before really leaving the loop.

Another useful statement is the continue statement. When the continue statement is found, It continues with the next cycle of the nearest enclosing loop. When continue passes control out of a try statement with a finally clause, that finally clause is executed before really starting the next loop cycle.

Some examples of the use of break and continue statements in while and for loops are shown below.

This program will print only numbers from 1 to 4. When n becomes 5 it will finish the current while statement and run the next statement which is print('All numbers printed')

n = 0
while n < 10:
n = n + 1
print(n)
if n == 5:
break
print('All numbers printed')
	#while loop
	n = 0
	while n < 10:
	n = n + 1
	if n % 2 == 0:
	continue
	else:
	print(n)

Print all numbers from 1 to 4. Though the condition to check is n < 10, when n is 5 the break statement will go to the next statement after the while statement.	This program will only print odd numbers. When an even number is found, it will go to the next iteration of the loop, starting at the while statement.

This program will print numbers from 1 to 10 only if the number is odd. When the number is even (i.e. the remainder when the number is divided by 2 is 0), it goes to the next iteration. When n becomes 5 it will finish the current while statement and run the next statement which is print('All numbers printed').

There is one other statement known as the pass statement which is used when the body segment in a program is empty. That is, when a statement is required to be written so that the program syntax is correct, but no action is required. In such cases a pass statement is used as follows.

```
if x < 0:

pass # need to handle negative values!
```

Or

```
>>> while True:
. . . pass # Busy-wait for keyboard interrupt (Ctrl+C)
. . .
```

3.3 Data Structures

Python has a rich repertoire of data structures that allow the manipulation of sets of data easily and efficiently. Here we discuss some of the basic data structures such as List, Tuple, Dict.

At this point, it is useful to understand two types of data: immutable and mutable. Immutable data are values that cannot be changed once created.

-For example, int, float, long, str, tupe etc are immutable data types whereas list, set and dict are mutable data types. It is important to understand that only mutable objects support methods that change the object in place, such as reassignment of a sequence slice, which will work for lists, but raise an error for tuples and strings.

Video 5:

Working with Data Types and Data Structures

Y ou may watch this video with a screen cast to see how to work with Python Data Types.

URL: https://youtu.be/1lwY4eaI-0o

A slice is a part of a data item such as a list and the range of the slice is marked using the notation [start index: stop index]. Slice operator (:) also has an extended version that includes a stride parameter as follows: [start index: stop index: stride]. The stride value is similar to the step value in range statement.

Example 3.4

```
# Slicing lists characters =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n']
simpleslice = characters[3:8]
print('Some characters are: ', simpleslice)
slicewithstride = characters[3:8:2]
print('Slicing with stride parameter:',slicewithstride)


output
Simple slice: ['d', 'e', 'f', 'g', 'h'] Slicing with stride paramter:     ['d', 'f', 'h']
```

In the above, the Simple slicing extracts the characters from elements at index 3 to element at 7 (upper index - 1). Remember that indexing starts with 0.

Slicing with strid parameters extracts every 2nd character from elements at index 3 to element at 7 (upper index - 1).

Since these parameters are optional, omitting some characters will work as follows:

characters[start:end] # items start through end-1

Table 3.1 slicing operator examples with a list

characters[start:]	items start through the rest of the array
characters[:end]	items from the beginning through end-1
characters[:]	a copy of the whole array
characters[-1]	last item in the array. - sign indicates extracting from the end of the list.
characters[-2:]	last two items in the array
characters[:-2]	everything except the last two items

The following list of operations with mutable objects is given in official Python documentation.

Table 3.2 list of operations with mutable objects

s[i] = x	item i of s is replaced by x
s[i:j] = t	slice of s from i to j is replaced by the contents of the iterable t of the same size
del s[i:j]	same as s[i:j] = []
s[i:j:k] = t	the elements of s[i:j:k] are replaced by those of t
del s[i:j:k]	removes the elements of s[i:j:k] from the list
s.append(x)	appends x to the end of the sequence (same as s[len(s):len(s)] = [x])
s.clear()	removes all items from s (same as del s[:])
s.copy()	creates a shallow copy of s (same as s[:])
s.extend(t) or s += t	extends s with the contents of t (for the most part the same as s[len(s):len(s)] = t)
s *= n	updates s with its contents repeated n times
s.insert(i, x)	inserts x into s at the index given by i (same as s[i:i] = [x])
s.pop([i])	retrieves the item at i and also removes it from s
s.remove(x)	remove the first item from s where s[i] == x. Raises an error when the value is not found.

s.reverse()	reverse the items of s in place. Note that it does not return the reversed list.

These operations are applicable to mutable data types.

3.3.1 List

A list is a data structure that can represent an ordered set of items. Lists are said to mutable in that the size of the list and the items within it can be changed after creating a list.

The format for a list is a set of items enclosed by square brackets [].

Example 3.5

```
List of strings:['Sri Lanka', ' India', 'Pakistan', 'bangaladesh', 'Nepal']
List of numbers:[1,3,5,7,11,13]
A list of mixed items:['Saman', 45, '-34.67, 'kumari'] The empty list:[]
```

We have already seen how we can iterate through values in a list using a for statement. Other operations that can be done on lists were described in the previous section under Data Structures.

Video 6:

Operators and Control Structures in Python

You may watch this video on Operators and Control Structures in Python before attempti ng the Activity 3.2.

URL: https://youtu.be/qxw690359YY

Activity

Activity 3.2

Write a program using a for loop to print the cube value (power of 3) of five numbers given in a list.

3.3.2 RangeType

Range is a built in type that returns an arithmetic sequence. In versions prior to version 3, range was a function that returned a list. Range generates a list of numbers upto, but not including, the stop value in the range statement. The general formats of range are;

```
range[stop].
range([start], stop [,step]) : where values for
[start] and [,step] are optional.
```

All values for start, stop and step must be positive or negative integers. If value for start is not specified, the default values of 0 is assigned. For step the default value is 1.

For example:

```
range(5) will generate 0,1,2,3,4
range(5,10) will generate 5,6,7,8,9
range(0,10,3) will generate 0,3,6,9
```

range type is commonly used in for loops to iterate through a known sequence of values.

Example 3.6: Repeat a sequence 10 times, the following uses the range type to generate numbers from 0 to 9 and then print each one.

```
for item in range(10):
# range will generate values from 0 to 9 print(item)
```

```
cities = ['Colombo', 'Kandy', 'Jaffna', 'Matara', 'Anuradhapura']
for item in range(len(cities)):
# range will use the length of the list
print('City in position ',item,' of the list is ', cities[item])
```

If you want to convert the arithmetic progression generated by range to a list, a conversion can be used.

Example 3.8 : Convert values generated by range into a list of value

```
valuelist = list(range(5) Now, valuelist = [0,1,2,3,4]
```

3.3.3 Tuples

Tuples are similar to lists in that they collect related data together. A tuple is of fixed size and is immutable, i.e. cannot be changed once created. Tuples are useful when you want to create a record structure of related data items. Tuples are enclosed within brackets. For example,

```
aruna = ('Aruna', 'Silva', 1978, '747872567').
```

You can use indexes to select items from a tuple as in a list or a string. for example aruna[2] will return 1978.

Tuples are immutable objects, so you cannot change the values of a tuple, for example, the following statement is illegal.

aruna[2] = 1979

However, we can create a new tuple using the values from the existing tuple, and adding some extra values given as a tuple

for example,

aruna = aruna[0:1] + (1979,) + aruna[3:]

will assign the tuple ('Aruna', 'Silva', 1979, '747872567') to the variable aruna. Note that a tuple with a single value has to be ended with a comma after the value.

One of the very powerful operation with tuples is the tuple assignment. It allows values in a tuple given in the right side of the assignment operator to be assigned to a set of variables given in the left.

for example

```
(firstname, lastname, birthyear, phone) = aruna
<will do the following assignments>
firstname = 'Aruna', lastname = 'Silva', birthyear = 1979, phone = '747872567'
```

Remember that the number of variables on the left side should be the same as the number of values in the tuple given in right

Activity

Activity 3.3

Write a program to create a list with a car details and display them.

3.3.4 Dictionaries

Dictionaries are another useful data structure. It consists of a set of key: value pairs enclosed within curly brackets {}. For example, the following is a list of telephone numbers:

```
phonebook = {'Saman':112847365, 'Iresha':772726286,'Aziz':1124273254, 'Bhavani':717976475}
```

We use an index to access value from a list, string or a tuple, values of

Access a value from a dict	phonebook['Aziz']
New values can be assigned to a dict	phonebook['Harry'] = 779274946
Values from a dict can be deleted with del	del phonebook['Saman']

Keys and values can be assigned to lists	names = list(phonebook.keys()) phonenos = list(phonebook.values())
Check if a key is in the dict	'Iresha' in phonebook:
Use an expression to create a dict	(number:number** for number in (1,2,3,4,5)
Both key and value from a dict can be retrieved with a loop using the items() method of dict	for key,value in phonebook.items(): print(key,value)

Table 3.3 Way to access values from a list, string or tuple

Unit summary

Summary	In this unit you learned the control structures that determine the execution of a program. We discussed the two main control structures, selection and iteration providing appropriate examples on how to program in Python. We also discussed the manupulation of simple data structures that is available in Python

References and Further Reading

1. Allen B. Downey (2012). Think Python. http://greenteapress.com/wp/think-python/
2. https://www.ibiblio.org/g2swap/byteofpython/read/index.html

4

Unit 4: Functions

Unit Structure

4.1 Why Functions

4.2 How to write a function definition in Python

4.3 Key things to remember when definingfunctions

4.4 Flow of execution

4.5 Functions with arguments

4.6 Void Functions

4.7 Functions with return statements

4.8 Built-in functions

4.9 Type Conversionfunctions

4.10 Math functions

Introduction

Functions in Python help you to reuse the code and make your coding more reliable.

Upon completion of this unit you will be able to:

Outcomes

- describe the importance of functions in Python.
- explain the function definition in Python
- use the void functions and return statements
- explain the difference between different function argument types and use them when defining Python functions
- select built-in functions in Python to write programs in Python

Terminology

Header : The first line of afunction
Body : The set of Python statements written inside a functiondefinition

Voidfunction : A function that does not return anyvalue

Flowofexecution :The order in which the Python statementsare executed during a program run

4.1 Why Functions

In the previous unit you may have learned some Python programming. Sometimes you may have noticed that certain tasks must be repeatedly carried out and so, some Python statements must have been repeated without you noticing. A way to correct this problem is to introduce functions to your coding. By introducing functions you can reuse your coding which were already written.

4.2 How to write a function definition in Python

To introduce a function, you need to extract commonly used sequences of steps into one body of coding and label it as a function.

The header is the first line of the function definition and the body is the rest of the function definition.

A function definition consists of the keyword def in the header followed by the function name which needs to be followed by a sequence of parameters enclosed in parentheses. The header has to end with a colon after these parentheses. If the parentheses are left empty it means that this function does not take any arguments.

The inputs to the functions are known as arguments which can be either a value or a variable. The function body may consist of many Python statements which are indented.

Remember that you need to always define the function before it is first called.

Example 4.1

```
1. def print_twice():
2. print "Hello"
3. print "Hello"
```

Line 1 is the header of the function. It state that the name of the function is print_twice and there is no argument accepted by this function.

Line 2 and Line 3 constitute the body of the function which will perform the task of printing.

Output of the program written for Example 4.1

Hello

Hello

4.3 Key things to remember when definingfunctions

- first character of a function name cannot be a number or a special character
- Python keywords are reserved and cannot be used as the name of a function
- Avoid giving the same name to a variable and a function
- To end a function definition simply enter an empty line
- Statements inside a function will not execute until the function is called
- A function definition will not generate any output left uncalled

4.3.1 How to call a function in Python

- Simply call the function using the function name. If any arguments are there then you need to write the arguments as well.
- To call function written in Example 1 just type the function name as follows
- print_twice()

4.4 Flow of execution

The order in which the Python statements are executed by the interpreter is known as flow of execution. Execution of Python statements in a function starts with the first one, and carried out in a sequence one after the other, from start to end.

In the middle of a function if a function call appears then the interpreter will jump to this function execute it and come back to original function.

Video 7: **Introduction to Functions**	
	This video will further explain the basic concepts of function in Python. URL: https://youtu.be/XHAA_tCkfXM

4.5 Functions with arguments

Consider the following function definition. This function is written so that it will accept any value as an argument and print the value twice.

Example 4.2

```
1. def      print_twice( myVariable ):
2. print(myVariable)
3. print(myVariable)
```

Line 1 is the header of the function. It states that the name of the function as print_twice and the argument is myVariable

Line 2 and Line 3 are the body of the function which performs the task of printing.

The above function can be called with any argument value as shown below

```
4.  >>>print_twice('Hello')
5.  >>>Hello
6.  >>>Hello
7.  >>> print_twice('Now I know how to define afunction inPython')
8.  >>>Now I know how to define a function inPython
9.  >>>Now I know how to define a function inPython
10. >>>print_twice(21)
11. >>>21
12. >>>21
13. >>>print_twice( '#' * 5)
14. >>>#####
15. >>>#####
16. >>>x = ' Python programming iseasy'
17. >>>print_twice(x)
```

```
18. >>> Python programming iseasy
19. >>> Python programming iseasy
```

In Line 10 this function uses an expression as an argument.

Lines 13, 14, 15 and 16 depict how the same function can be used with a variable as an argument. In this case the variable is x and its value is ‘Python programming is easy'.

4.5.1 Different Argument types used in Python

There are 4 argument types which are commonly used when writing Python codes.

- Default arguments
- Required arguments
- Keyword arguments
- Variable-length arguments

Default arguments

Consider the following example which demonstrate the use of default arguments.

Example 4.3

```
1. def find_average(x, y= 12):
2. average = (x + y)/2
3. print('The average of', x , 'and',y, 'is', average)
4. def main():
5. find_average(4, 6)
6. find_average(4)
7. main()
```

The output of the above program is:

The average of 4 and 6 is 5.0

The average of 4 and 12 is 8.0

Line 5 and 6 both call the function find_average with arguments. Line 6 uses the default argument, as the function is called with only one argument.

In the above Example the result 8 is given as the function uses the default argument 12 for the variable y.

Required arguments

As it name implies required arguments expect to be called matching with exactly the function definition. Consider the following example which demonstrate the use of required arguments.

Example 4.4

```
1. def print_twice( myVariable ):
2. print myVariable
3. print myVariable
4. def main():
5. print_twice( )
6. main()
```

Line 5 of the above program calls the function print_twice() without any arguments.

The above program will give the following result:

TypeError: print_twice() takes exactly 1 argument (0 given)

Keyword arguments

With the use of the Keyword arguments, the programmer is able to call the function with arguments in any order and still the interpreter will match the values for the arguments and execute the program accordingly.

Consider the following example which depicts the use of Keyword arguments:

Example 4.5

```
1. def printNumbers( x , y ):
2. print'Value of x variable is' , x)
3. print('Value of y variable is', y)
4. def main():
5. printNumbers( y = 3, x = 6)
6. main()
```

The above program gives the following output:

Value of x variable is 6

Value of y variable is 3

Variable-length arguments

When programming you may come across situations where you will not be aware of the number of arguments involved with a certain function when defining the function. Still you can use such functions with the variable- length arguments as shown in the following example.

Example 4.6

```
1. def add(*args):
2. i = 0
3.         for j in args:
4.         i = i + j
5.         print(i)
6. add(3,6)
7. add(2.3,5)
8. add(3,6,8)
9. add(3,6,8,12)
```

The output of the above program is

9

7.3

17

29

The important thing to remember here is the use of the single-asterisk in front of the argument, *args.

Activity	Activity 4.1 Write a Python program which will accept a number within the range of 1 and 20, and find whether it is a prime number or not. If it is a prime number, function should return True and if not False. Note that prime numbers are numbers which are divisible only by 1 and by the number it-self. What is the argument type that you use here?

4.6 Void Functions

In Example 1 and 2 we were working with void functions. void functions simply carry out the tasks inside the function but will not return any value to where it is called.

4.7 Functions with return statements

To return any result from a function we need to use the return statement inside the function body.

Consider the following example written to find the square root of any number.

Example 4.7

```
1. Def squareN ( y );
2. squareValue = y × y
3. return squareValue
4. answer = squareN(5)
5. print(answer)
```

The output of the above program is 25. You can see that the print statement is written outside the function squareN.

Line 1 is the header of the function and it takes y variable as the argument.

Line 2 executes the mathematical operation to find the square root of the given number.

Line 3 return the value of the squareValue

Line 4 depicts that it is end of the function

Line 5 calls the function squareN and assigns the result returned by the function

Line 6 prints the answer of the result

4.8 Built-in functions

The functions that we used in earlier examples are all user defined functions. Python has built-in functions which we can use by simply calling them by their names and relevant arguments. You do not have to define these built-in functions.

Consider the built-in function len which gives the length of a string as its output.

Example 4.8

```
len('Now I know how to use functions in Python')
```

This function will give the output 33

Built-in functions can be dealing with strings, numbers and/or mathematical operations.

Some built-in-functions and their functionality are listed below. For the complete list of built- in functions available in Python, visit the web page https://docs.python.org/3/library/functions.html

- abs(x) :- Return the absolute value of a number. Here the argument x may be an integer or a floating point number. When the argument is a complex number, the function will return its magnitude.
- bool([x]):- Return a Boolean value, either True or False. Here if x value is false or omitted, this returns False; otherwise the function bool will return True.
- chr(i):- Return the string representing a character whose Unicode code point is the integer i. For example, if i value is 112 then the chr(112) returns the string ‘p’ and if the i value is 169 then the chr(169) returns the string ‘©’.
- divmod(a, b):- Return a pair of numbers consisting of their quotient and remainder of an integer division. This function accepts two (non complex) numbers as arguments. If the operand types are mixed, then the rules for binary arithmetic operators apply.
- max(iterable, *[, key, default]):- Return the largest item in an iterable argument.
- max(arg1, arg2, *args[, key]):- Return the largest of two or more arguments.
- min(iterable, *[, key, default]):- Return the smallest item in an iterable argument
- min(arg1, arg2, *args[, key]):-Return the smallest of two or more arguments.
- round(number[, ndigits]):- Return number rounded to ndigits precision after the decimal point. If ndigits argument is omitted or is None, then the function returns the nearest integer to its input number

Video 7: **Working with Functions**	
	You may watch this video with a screen cast to see how to write functions in Python before attempting Activity 4.2 https://youtu.be/P8ZDJ2876NA

Activity	Activity 4.2 Using built-in functions available in Python write a program to round the number 397.234567 to 3 decimal places.

4.9 Type Conversionfunctions

One type of built-in functions that are available in Python are type conversion functions. These functions convert values from one type of values to another type.

- The int function converts any value to an integer but it does not round the number.
    - Int('9') gives the output as 9
    - int(9.7) gives the output as 9
    - int(-102) gives the output as -102
    - int('Introduction to Python') gives the output as an
    - error message
- The float function converts integers and strings to floating point numbers
    - float(88) gives the output as 88.0
    - float('88') gives the output as error message
- The str function converts its arguments to a string
    - str(9.7) will give the output as '9.7'

4.10 Math functions

To execute mathematical operations in Python you can use the math module available in its library. Before we use any math function we need to import this math module to our program by typing import math

To perform different tasks, different types of math functions are available such as

- Number representation functions
- Power and logarithmic functions
- Trigonometric functions
- Angular Conversion functions
- Hyperbolic functions
- Constants

After importing the math module to access the functions in it, it is necessary to specify the name of the module and the name of the function. Module name and function name are separated by a dot.

Example 4.9

```
ratio = signalPower / noisePower

decibels = 10 * math.log10(ratio)
```

Lambda Function

Lambda functions in Python are unnamed functions which are also known as anonymous functions. These functions can be very helpful when writing short functions which do not need a function definition with the keyword def.

Consider the following two Python programs written to get the average of two numbers using a normal function and using a lambda function.

Example 4.10

Without the lambda function	lambda function

def find_average(x, y):
average = (x + y)/2 print average
def main():
find_average(4, 6)
main()
	print(lambda x, y: (x+ y)/2) (4,6)
	
	or
	
	g = lambda x, y: (x+ y)/2
	print (g((4,6))

5	5

Both gives same output (5).

Activity Activity 4.3

Write a lambda function to print the square of a number.

Unit summary

Summary

In this unit you learnet the importance of using functions and how to use functions. Further you learnt how to define functions, to write functions with different argument types, describe the difference between the in- built functions and user defined functions, to write user defined functions, to use in- built functions, import math module in to a Python program and to use lambda functions

References and Further Reading

1. Allen B. Downey (2012). Think Python.
   http://greenteapress.com/wp/think-python/
   Download this book for free at
   http://greenteapress.com/thinkpython/thinkpython.pdf
2. Swaroop C H , A Byte of Python,
   https://python.swaroopch.com/l
   Download this book for free at
   https://python.swaroopch.com/

Unit 5: Strings 5

Unit Structure

5.1 Stringsin Python

5.2 String Operations

5.3 Stringmethods

5.4 Parsing strings

Introduction

Strings are an important data type commonly used in Python. In this unit we learn how to create strings and manipulate them.

Upon completion of this unit you will be able to:

:

Outcomes

- *explain*the structure of astring.
- *apply* basic operations onstrings
- *apply* methods in srings to a program

Terminology

Immutable : An assigned value to a sequence cannotbe changed during execution

Sequence : A set of values ordered according to meaningor value

Slice : A part of a string witin a specific range ofindices

traverse : To go through the items in asequence

5.1 Stringsin Python

A string is a sequence of characters and can be created by enclosing characters in quotes.

flower = 'jasmine’

A character can be accessed one at a time with the bracket operator:

```
>>> flower = 'jasmine’ >>> letterOne = flower [0]
```

The second statement extracts the character at index position 1 from the flower variable and assigns it to letterOne variable. In Python, similar to C language, the

index is an offset from the start of the string. Offeset starts from 0 and increase, so the offset of the first letter is zero.

value	J	A	s	m	i	n	e
offset	0	1	2	3	4	5	6

Value of the index must be an integer always.

Python has no character type and the characters are treated as strings of length one which each one is called a ‘substring’.

A string can be specified using single quotes or double quotes which means the same in Python.

‘Can you pick some flowers?’ works exactly the same way as “Can you pick some flowrs?”

Multi-line strings can be written with triple quotes and single or double quotes can be given inside triple quotes.

For example you can write;

“` Anjana asked “Hello there, are you picking flowers?” “Yes, do you want any?” answered Mala …..”’

5.1.1 String formatting

Often we need to print a message including text, numbers etc stored in other variables. In such a case string formatting is needed. The % sign which is used to show where the value of variable should be, is called a placeholder.

Example 5.1

```
countryName = “Sri Lanka”
flowerName = “Blue water lily”

print(“ Name of my country is %s, “ %countryName)
print(National flower of my country is %s, %flowerName)
```

The format sequence '%s' is a place holder for a string as shown above, while %d is for an integer as shown below:

```
>>> mark = 50 >>> '%d' % mark '50'
```

There is a special built-in function in Python for string formatting called

format, which allows a string to be constructed from other information.

Example 5.2

```
marks = 75
name = 'Dulani'
print('{0} has obtained {1} marks for Physics last year'.format(name, age))
```

5.2.1 Escape Sequences

A character in escape sequence is used to denote a special character or which has been reversed for a special purpose. An escape sequence starts with a backslash ‘\’.

e.g. inserting a new line in a string;.

```
print(“Hello I am Neela.\n, This is my friend Anjali”)
```

e.g. inserting apostrophes;

```
print(“Hello I\’m Neela.\n, This is my friend Anjali.”)
```

Common escape sequences are given in Table 5.1 Table 5.1 A Common set of escape sequence

Sequence	Meaning
\\	Literal backslash
\’	Single quote
\”	Double quote
\n	New line
\t	Tab

5.2 String Operations

In Python language, we find many built-in functions which perform operations on strings. Using ‘len’ function to find length, finding substrings, comparisons and concatenation are few common operations.

5.2.1. Finding the length of a string using len

Using ‘len’ function we can find out the number of characters in a string:

```
>>> flower = 'jasmine' >>> len (flower) 7
```

However, if you try to get the last letter of a string as given below, you will see an error message:

```
>>> length = len(flower) >>> last = flower[length] IndexError: string index out of range
```

Though there are seven letters in ‘Jasmine’, we started counting from zero. Therefore, the seven letters are numbered from 0 to 6. In oreder to find the last character, we should subtract 1 from length:

```
>>> last = flower[length-1] >>> print last
```

Otherwise, there is a possibility to use negative indices, which count backward from the end to the beginning. An expression like flower[-1] would give the last letter of the sring as the output. Similarly, index can be further decreased to [-2], [-3] until [-6].

5.2.2 Using a Loop to Traverse a String

When a string is manipulated, one character is accessed at a time. Usually, we start from the first position, select each character in turn, work with that character, and continue till we reach the last character. Going along a string, inspecting each character is called a ‘traversing the string’. String traversal can be easily done with a while loop :

Example 5.3

```
index = 0

while index < len(flower): character_in_flower = flower[index]

print character_in_flower index = index + 1
```

Activity

Activity 5.1

Write a ‘while’ loop that starts at the last character in the string and works its way backwards to the first character in the string, printing each letter on a separate line. You should print the characters in reverse order.

5.2.3 String slices

When talking about strings, a ‘slice’ means it is part of a string. A string slice may contain one or more characters:

```
>>> mystring = 'MorningGlory' >>> print mystring[0:7] Morning
```

The operator used to extract a slice, [n:m] returns the part of the string from the character “n” to the character “m”. This operator includes the character denoted by “n” but exclude the character denoted by “m”.

If the first index (before the colon) is omitted, then extracting the slice starts from the beginning. Similarly,if the second index is omitted, the slice will be extracted from the given position till end of the string:

```
>>> flower = 'jasmine' >>> flower[:3] 'jas' >>> f[3:] 'ine'
```

When specifying a slice, if the first and the second indices are equal or if the first one greater than or equal to the second one, extracted slice would be an empty string:

```
>>> flower = 'jasmine' >>> flower[3:3]
```

A string is called ‘empty’ if it contains no characters. Length of an empty string is 0.

Activity Activity 5.2

Given that flower is a string, what would be the results if following statements are excuted,

I. flower[:],
II. flower[3:7],
III. flower[3:3] ?

5.2.4 Strings are immutable

In Unit 3 we discussed about immutable data types. Strings are also immutable, which means once a value is assigned to a string the value cannot be changed during the execution. Therefore, one item cannot be changed using the [] operator on the left side of an assignment.

Example 5.4

```
>>> greeting = 'hi,there!'

>>> greeting[0] = 'H'

TypeError: ‘str’ object does not support item assignment
```

Error message above indicates that a new value cannot be assigned to an existing string during the process of execution. If a change is necessary, we can use string a manipulation function like concatenation to carry out the change we want as shown below.

```
>>> greeting = 'hi,there!' >>> new_greeting = 'H' + greeting[1:] >>> print new_greeting
Hi,there!
```

You may note that in this manipulation the original string is not changed at all.

5.2.5.Substring

The word ‘in’ acts as a boolean operator in Python. It compares two strings and returns True if the first string is a substring of the second:

```
>>> 's' in 'Jasmine' True
>>> 'e' in 'Mala'
False
```

5.2.6 String comparison

The comparison operators work on two strings to see if the two strings are the same:

```
fruit1 = ‘orange’
if fruit1 == 'orange': print 'correct fruit' if fruit1 != ‘banana’: print ‘wrong fruit’
```

```
word1 = green word2 = blue
If word1< word2:
  Print(word1 + “comes before”+ word2) elif word1 > word 2:
Print(word1 + “comes after”+ word2)
```

Comparison operations are also useful for arranging words in alphabetical order. Uppercase and lowercase letters are handled differently in Python and if arranged, all uppercase letters come before lowercase letters. Therefore, where case is not considered, all letters should be converted to lowercase before comparison.

5.3 Stringmethods

String is an example of a Python object. An object contains both data (the actual string itself) as well as methods (or functions) which are built into the object and are available to any instance of the object.

Use the type function to find out the type of the object as shown below.

```
>>> myFolder = 'findFunctions'
>>> type(myFolder)
Output
<class 'str'>
```

Python has a function called ‘dir’ which lists the methods available for an object.

```
>>>dir(myFolder)


output
[' add ', ' class ', ' contains ',
' delattr ', ' dir ', ' doc ', ' eq ', ' format ', ' ge ', ' getattribute ',
 getitem ', ' getnewargs ', ' gt ',
 hash ', ' init ', ' init_subclass ',    ' iter ', ' le ', ' len ', ' lt ',
' mod ', ' mul ', ' ne ', ' new ', ' reduce ', ' reduce_ex ', ' repr ',
 rmod ', ' rmul ', ' setattr ', ' sizeof ', ' str ', ' subclasshook ', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

You can use ‘help’ to get some information about a function or a method.

```
>>> help(str.capitalize)
Output
Help on method_descriptor: capitalize(self, /)
        Return a capitalized version of the string.
More specifically, make the first character have upper case and the rest lower case.
```

Refer following website for more documentation for string methods would be https://docs.python.org/library/stdtypes.html#string-methods

Calling a method is similar to calling a function. But the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter. A method also takes arguments and return a value.

Let us take another example where the method upper takes a string and returns another string after converting the original string to all uppercase letters:

Method uses the syntax word.upper() in contrast to function syntax which could be upper(word).

```
>>> string1 = 'angel' >>> string2 = string1.upper()
>>> print string2
Output
ANGEL
```

Here the name of the method is upper and the method is applied to string1. Since the method does not take any arguments, empty parentheses are used.

Calling a method is named as invocation. Here we invoke upper method on string1.

There is another useful method called find in Python language. In the following example, we invoke find on string1 while passing the letter we search as a parameter.

```
>>> string1 = 'angel' >>> letter = string1.find('g')
>>> print letter
Output
2
```

In fact, find method can find substrings, not only characters:

```
>>> string1.find('gel')
Output
2
```

One common task we face usually is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string. For that, we can use use strip method:

```
>>>line = ' Hello world ! '>>> line.strip()
Output
>>>'Hello world!'
```

5.4 Parsing strings

Often, we want to look into a string and find a substring. For example if we are presented a series of lines formatted as follows:

```
From nirmal@ ou.ac.lk Fri Jan 2 08:10:110 2016
```

And we want to pull out only the second half of the address (i.e. ou.ac.lk) from each line. We can do this using the find method and string slicing.

First, we find the position of the @ symbol in the string. Then we find the position of the first space after the @ symbol. And then we use string slicing to extract the portion of the string which we are looking for.

```
>>> data = 'nirmal@ou.ac.lkFriJan208:10:102016' >>> atpos = data.find('@') >>> print atpos 21 >>> sppos = data.find(' ',atpos) >>> print sp-pos 31 >>> host = data[atpos+1:sppos]

>>> print host ou.ac.lk >>>
```

We use a version of the find method which allows us to specify a position in the string where we want find method to start searching. When we slice, we extract the characters from “one beyond the @-sign through up to but not including the space character”.

The documentation for the find method is available at docs.python.org/ library/string.html

Activity

Activity 5.3

Write a program to concatenate two strings Divya asked, and Great! Then can you let me have all the mangos, limes, oranges and apples? And then remove all punctuations from resulting string.

You may use one ‘if then else condition and a while loop Hint: two strings can be concatenated using ‘+’ operator

Unit summary

Summary	In this Unit you learned how to define a string, format a string and how to perform string operations on strings. String comparison, finding string length, string concatenation, extracting substrings and string slicing are the main operations that were discussed. We also discussed that strings are immutable and are treated as objects in Python which has its own set of functions.

References and Further Reading

1. Swaroop C H , A Byte of Python, CC-BY 4.0 International License Download this book for free at https://python.swaroopch.com/
2. Allen B. Downey (2012). Think Python.CC-BY-NC Download this book for free at http://greenteapress.com/thinkpython/thinkpython.pdf
3. Basic String Operations, Retrrieved June 2017 from https://www.learnpython.org/en/Basic_String_Operations

Unit 6: Object Orientation as a Programming Paradigm

6

Unit Structure

6.1 Overview of object-oriented programming (OOP)

6.2 Basic concepts of objects and classes

6.3 Methods inPython

6.4 Operator overloading

Introduction

Python is primarily designed as an object-oriented programming language. This unit will focus on Object orientation as a programming paradigm with creating and using classes, objects, attributes and methods. So, learning this unit will help you to write programs using Python's object-oriented programming support.

Upon completion of this unit, you will be able to:

Outcomes	• Identify the object oriented nature of Python. • Define class, object and method in Python. • Differentiate functions and methods in Python.
Terminology	**Object** : A real world entity or a concept **Class** : A user-defined type. **Field** : A variable that belong to a class. **Method** : Functions that belong to a class **Attribute** : A field or a method associated with a class

6.1 Overview of object-oriented programming (OOP)

In the "real" world, objects are the entities of which the world is encompassed. Everything that happens in the world is considered to be the interactions between the objects in the world. Real-world objects share two characteristics, attributes and behaviours.

For an example if we take the real world object “Dog”, Dogs have attributes (name, colour, breed, hungry) and behaviours (barking, fetching, wagging tail). Bicycles also have attributes (current gear, current pedal cadence, current speed) and behaviours (changing gear, changing pedal cadence, applying brakes). Dogs and Bicycles are physical objects. Similarly, we can consider conceptual objects such as a course that

a student will be registered for. Courses have state (course code, course title, number of credits, the department offering the course) and behaviour (offer course, exempt course, drop course).

In OOP, programs are made up of objects and functions that are required to work with those objects. So, programs contain object definitions and function definitions. An object definition directly corresponds to some entity or a concept in the real world, whereas the functions correspond to the methods those entities interact.

In general, a class is a blueprint where objects are instances of a particular class. For example, the vehicle class may inclu de objects like bicycle or a car.

Object oriented programming concept is a model organised around "class/object" concept rather than on functions. Objects are modeled after real-world entities.

6.2 Basic concepts of objects and classes

In Python, a class is defined as a type. A class creates a new type where objects are instances of that class. Even variables that store integers are treated as instances of class int. Using type function, we can find out the type of any object.

Functions are used to reuse the code in different places in a program. In Python there are inbuilt functions like print) and also enables you to create your own functions.

6.2.1 User-defined types

In Python new types can be defined and manipulated from Python code in a similar way that strings are defined manipulated. To define a new type, an extension module must be defined and supported by Python. It is easy as all extension types follow a general pattern. During the execution of a Python program, all Python objects are treated as variables of type PyObject*, which contain a pointer to the object's "type

object". Here, the type object determines which (C) functions get called when an attribute of an object is accessed. These C functions are called "type methods".

As an example, let us consider that we need to write a program which has a point in two-dimensional space. How do we represent a point in Python?

In mathematics, points are represented within two parentheses like this: (2,3) which means x coordinate is 2 and y coordinate is 3.

There are several ways we can represent a point in Python:

- store the coordinates in two separate variables
- store the coordinates as elements of a list or a tuple.
- create a new typecalled Point represent points as objects.

6.2.2. Defining Classes in Python

In general terms, 'Class' is a blueprint to create objects. In Python, class is a special data type. A Class contains some data items and methods that operate on those data items. Objects created as instances of such a defined class will be having all those data items and methods.

Class statement followed by the class name creates a new class. An example of a simple Python class is given below.

```
class Bike:
'''This the name space for a new class''' pass
```

'class' keyword defines the template that specify what data and functions will be included in any object of type Bike.

```
>>> myBike = Bike()
>>> print(myBike)
output
< main .Bike object at 0x033F7970>
```

Since Bike is defined at the top level, its name appears as "____Main__. Bike".

An object is an instance of a particular class. To create an object, we call the class using the class name and pass the arguments its _init_ method accepts. (passing arguments and the _init_ method will be discussed later).

Creating a new object is called instantiation of a calss. When an object is printed, Python displays to which class this object belongs to and the memory address it is stored (Memory address is given in hexadecimal as evidant from prefix 0x).

6.2.3. Attributes

The attributes are data members such as class variables or instance variables and methods. There are two types of attributes.

- Data attributes - Owned by an instance of a class
- Class attributes - Owned by the class as a whole these attributes can share with all the instances of a class.

```
class Bike:
gear = 1
speed = 0
```

Attributes are accessed via dot notation. Similary, values also are assigned to an instance also using dot notation:

```
>>> myBike.gear = 2
>>> myBike.speed = 45
```

Class methods have only one specific difference from ordinary functions. That is, they must have prefix that has to be added to the beginning of the parameter list which does not need a value when the method is called. This parameter is called self.

Here is an example to see how Python gives a value for ‘self’.

When you call a method of this object as myBike.method(arg1, arg2), this is automatically converted by Python into myBike.method(blank, arg1, arg2).

That is, even if you have a method without arguments, then you still have to have one argument which is ‘self’.

To read the value of an attribute we have to use the same syntax as we assigned values:

```
>>> myGear = myBike.gear
>>> print(myGear)
output
1
```

Here the variable name myGear can also be gear and that will not get mixed up with the attribute gear.

Dot notation can be used as part of any expression. For example,

```
>>> print(myBike.gear, myBike.speed)
(1,0)
```

An object can be passed as an argument to a function. For example,

```
def print_bike(k):
print(k.gear,k.speed)
```

print_bike(k) function takes a myBike as an argument and invokes print function,

```
>>> print_bike(myBike)
output
(1,0)
```

Inside the function, k will be substituted by the argument myBike. If k is changed inside the function, the values of myBike changes.

```
>>>def set_gear(newValue): gear = newValue
print("Gear is at ",%d, gear)
>>>set_gear(4)
Output
Gear is at 4
```

Next, we will see how to define methods in Python.

6.3 Methods inPython

A method, sometimes called a function, which defines a behaviour, is created by def statement. Each class has several methods that are associated with it. You may recall we used methods when manipulating strings. We need methods to work with user-defined types also.

Methods are similar to functions except for two differences. Since Methods are needed to work with a class, they are defined inside a class definition. Secondly, invoking a method is different from that of invoking a function.

Video 9: **ObjectOrientedBasics**	
	You may watch this video first and then attempt Activity 6.1. URL:https://youtu.be/2cB528Ww5F8

Activity	Activity 6.1
	Write a Student class which contains studentID, lastName, courseID. Input values to one object of type student and print the values.

6.3.1 Defining methods

Let us define a class named Student and create a function named display_student().

```
class Student:
regNo = ' '
 name = ' '
 fee = 0
def display_student(student): print(student.regNo,student.name,student.fee)
```

To call this function, need to pass student object as an argument.

```
>>> s1 = student()
>>> s1.regNo = AL205
>>> s1.name = 'Jayawan'
>>> s1.fee = 5000
>>> display_student(s1) Output
AL205, Jayawan, 5000
```

To make display_student a method in this class, the function definition has to be moved inside the class definition. It is important proper indentation is done so that it is clear that display_student is a method in class Student.

```
class Student:
        def display_student (student):
                print(student.regNo,student.name,student.fee)
```

Same method can be written with self as follows also,

```
class Student:
        def display_student(self):
                print(self.regNo,self.name,self.fee)
```

Now you know few basic details about how to define a class, how to initialize attributes, how to print attributes and how to define methods and invoke them. There are two important methods in Python called init and str. Now, let us see why they are important.

6.3.2. Constructor or init method

The init method is the constructor of a class. It is invoked when an object is created and also called the 'initialization' of the object.

As in any other language where variabls are initialized at the beginning, it is a good practice to write____init method first when a new class is written to instantiate objects.

An example for an init method would look like the following,

```
# inside class Student that include student_display method:
        def init (self, regNo = '0', name =' ',fee =0):
                self.regNo = regNo
                self.name = name
                self.fee = fee
```

Parameters of _____init method usually have the same names as the attributes. The following expression, means, that the value of the parameter regNo is assigned as the value of an attribute of self.

self.regNo = regNo

Parameters are not a must. If Student is called with no arguments, you will get the default values.

>>> s2 = Student()

>>> Student.display_student(s2) 0, ,0

Since the program is getting bigger now, it is much easier to type it in an editor and run the program.

The complete program would look like following:

```
Class Student (object):
def init (self, regNo = '0', name = ' ',fee =0):
        self.regNo = regNo
        self.name = name
        self.fee = fee
def display_student(self):
        print(self.regNo, self.name, self.fee)
```

Please note the indentation. Proper indentation is a must in Python.

Video 10:

Classes and Object Declaration

You may watch this video first and then attempt Activity 6.2.
URL: https://youtu.be/NYC34dE-XY8

Activity Activity 6.2

For the bicycle class, write an init method that initialize gear and speed to 1 and 0 respectively.

6.3.3 The strmethod

_____str is a special method in Python that would print a string which would describe an object.

Following is an example for str method for Student class.

```
class Student:
# inside class Student that include student_display method and _init_ method:
def str (self):
return('Student Reg No %s, Name %s, Fee %d'
%(self.regNo,self.name, self.fee))
>>>s1 = Student(13,Mala,3500)
>>>print(s1)
output
Student Reg No 13, Name Mala, Fee 3500
```

That is, when we print an object, str method is invoked. This method is very useful for debugging and print log files.

Next, we will see behaviours of operators in Python with add method and discuss operator overloading.

6.4 Operator overloading

When specially named methods are defined in a class, Python will automatically call them when instances of the class appear in the expressions. That is, by defining special methods, we can specify the behaviour of operators on programmer-defined types.

For example, if you define a method named add class, you can use the + operator on Student objects.

Here is an example for operator overloading:

```
class Student():
# inside class Student that include student_display,
_init_, _str_ methods write this method and run:
def add (self, fee_increment): total = self.fee + fee_increment return(total)
```

```
>>> start_fee = Student('13', 'Mala', 5000)
>>> print(start_fee+300) 5300
```

As you can see, when you apply the + operator to Student objects, Python invokes___add_____. To print the result, Python invokes____str_____.
Thus, for each and every operator in Python, a corresponding special method exists similar to ____add____. Here we are changing the behavior of an operator to work with programmer-defined types.

Activity	Activity 6.3
	Write a_____str method for the bicycle class and print it.

Unit summary

Summary	In this unit we had a brief overview of Object Orientation as a programming paradigm. We studied the syntax of the class statement, and how to define objects, classes and methods in Python. We also discussed method definitions, _init_, _str_ methods and how to define operator overloading methods.

References and Further Reading

1. Python for Everybody, Oject oriented programming, accessed web (2018), https://www.py4e.com/html3/14-objects Copyright CC-BY 3.0 - Charles R. Severance

2. Object-Oriented Programming in Python 1.0, http://python- textbok.readthedocs.io/en/1.0/Classes.html

3. Allen B. Downey (2012). Think Python. http://greenteapress.com/wp/think-python/

Unit 7: Object Oriented Concepts

7

Unit Structure

7.1 Object Oriented concepts

7.2 Inheritance

7.3 Multiple inheritance

7.4 Data Encapsulation

7.5 Polymorphism

Introduction

There are three widely used programming paradigms named as procedural programming, functional programming, and object-oriented programming. Python supports both procedural and Object Oriented Programming (OOP).

OOP is a programming paradigm that uses objects and their interactions to design applications and computer programs.

Upon completion of this unit, you will be able to:

Outcomes	• Identify object oriented concepts such as inheritance and polymorophism • Apply object oriented concepts such as inheritance and polymorophism to solve real world problems
Terminology	**Inheritance:** A way of defining a new class that is a subtype of the previously defined class which is called a super class. Subtype has all attributes of the super class and its behaviours. **IS-A relationship:** The relationship between a child class and its parent class.

7.1 Object oriented concepts

There are some more basic programming concepts in OOP such as Inheritance, Encapsulation, and Abstraction. In this unit we will discuss object oriented concepts in Python with Inheritance, Encapsulation and Polymorphism.

7.2 Inheritance

Concept of inheritance enables to represent objects according to their similarities and differences. All similar attributes and functions can be defined in a base class which is called the super class. Inheritance can also be seen as a way of arranging objects in a hierarchy from the most general to the most specific.

An object which inherits from another object is considered to be a subtype of that object. When we can describe the relationship between two objects using the phrase is-a, that relationship is inheritance. We also say that a class is a subclass or a child class of a class from which it inherits, or that the previous class is its superclass or parent class.

We can refer the most generic class at the base of a hierarchy as a base class. We can put all the functionality that the objects have in common in a base class, and then define one or more subclasses with their own custom functionality.

Here is a simple example of inheritance:

```
class Person:
def init (self, name, surname, idno): self.name = name
self.surname = surname self.idno = idno

class Student(Person):
UNDERGRADUATE, POSTGRADUATE = range(2)

def init (self, student_type, *args, **kwargs): self.student_type = student_type
self.classes = []
super(Student, self). init (*args, **kwargs) def enrol(self, course):
self.classes.append(course)

class StaffMember(Person): PERMANENT, TEMPORARY
= range(2)

def init (self, employment_type, *args, **kwargs): self.employment_type =
employment_type super(StaffMember, self). init (*args, **kwargs)

class Lecturer(StaffMember):
def init (self, *args, **kwargs): self.courses_taught = []
super(Lecturer, self). init (*args, **kwargs) def assign_teaching(self,
course):
self.courses_taught.append(course)

>>>Mali = Student(Student.POSTGRADUATE,"Mali", "Silva", "S045")
Mali.enrol(a_postgrad_course)
saman = Lecturer(StaffMember.PERMANENT,"saman", "perera", "077")
saman.assign_teaching(an_undergrad_course)
```

Base class is Person, which represents any person associated with a university and one subclass represents students where as another represents staff members.One more subclass represents Staff Members who are lecturers.
This example represents both student numbers and staff numbers by a single attribute, number, which is defined in the base class. It uses

different attributes for student (undergraduate or postgraduate) and also for staff members (permanent or a temporary employee).
This example contains a method in Student for enrolling a student in a course, and a method in Lecturer for assigning a course to be taught by a lecturer.
The init method of the base class initialises all the instance variables that are common to all subclasses. Each subclass override the init method so that we can use it to initialise that class's attributes. Overriding Methods can override parent class methods to add special or different functionality in to the subclass.

When the current class and object are passed as parameters, super function return a proxy object with the correct init method, which we can then call. In each of the overridden init methods we use the method's parameters which are specific to our class inside the method, and then pass the remaining parameters to the parent class's init method.
A common convention is to add the specific parameters for each successive subclass to the beginning of the parameter list, and define all the other parameters using *args and **kwargs then the subclass doesn not need to know the details about the parent class's parameters. Because of this, if we add a new parameter to the superclass's init , we will only need to add it to all the places where we create that class or one of its subclasses.

7.3 Multiple inheritance

Python programming allows multiple inheritance. It means you can inherit from multiple classes at the same time.

```
class SuperClass1():
        def method_s1(self):
                print("method_s1 called")

class SuperClass2():
        def method_s2(self):
                print("method_s2 called")

class ChildClass(SuperClass1, SuperClass2):
        def child_method(self):
                print("child method")
c = ChildClass()
c.method_s1()
c.method_s2()
```

In this example ChildClass inherited SuperClass1 and SuperClass2. And also object of Child Class is now able to access method_s1 and method_s2.

Activity Activity 7.1

1. Write a Person Class. Make another class called Student that inherits it from Person class.
2. Define few attributes that relate with Student class, such as school they are associated with, graduation year, GPA etc.
3. Create an object called student Set some attribute values for the student, that are only coded in the Person class and another set of attribute values for the student, that are only in the Student class.
4. Print the values for all of these attributes.

7.4 Data Encapsulation

Encapsulation hides the implementation details of a class from other objects. The aim of using encapsulation is that the data inside the object should only be accessed through a public interface. If we want to use the data stored in an object to perform an action or calculate a derived value, we define a method associated with the object which does this.

Encapsulation is a good idea for several reasons:

- The functionality is defined in one place and not in multiple places.
- It is defined in a logical place - the place where the data is kept.
- Data inside our object is not modified unexpectedly by external code in a completely different part of our program.
- Encapsualtion encorages the concept of data hiding

In the previous unit we discussed few more concepts in object oriented design. There, we identified objects like student and bicycle, and later defined classes to represent them. It was clear that the objects and the real world entities have close associations. However, there will be no clear association between objects and real world concepts all the time. So, as a programmer, you need to find out how they should interact. In such a case, we can discover class interfaces by data encapsulation as we discovered function interfaces by encapsulation and generalisation.

Protected members can access only within its class and subclasses. By prefixing the name of the variable with a single underscore, you can define protected variables in Python.

And also by using double underscore ( ) in front of the variable or a function name you will be able to define private members.

```
class Person:
def init (self): self.fname = 'saman' self. lname = 'perera'
def PrintPersonName(self):
return self.fname +' ' + self. lname
#Outside class P = Person() print(P.fname)
print(P.PrintPersonName()) print(P. lname)

#AttributeError: 'Person' object has no attribute ' lname'
```

In this example accessing public variables outside the class definition is possible. But cannot access private variable outside the class.

7.5 Polymorphism

In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes. For example, given a base class shape, polymorphism enables the programmer to define different area methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the area method to it will return the correct result.

For example, sum is a built-in function which adds the elements of a sequence. As long as the elements in this particular sequence support addition, sum function will work.

The Student class provides an add method. Therefore, we can use it to work with buil-in method ‘sum’.

```
>>> s1 = Student(99, 200)
>>> s2 = Student(99, 300)
>>> s3 = Student(99, 500)
>>> full_fee = sum([s1, s2, s3])
>>>print(full_fee) 1000
```

You can define functions to work with any type of data. Sometimes these may even operate on types not intended originally.

Functions which can work with different types are called polymorphic. They facilitate code reuse.

7.5.1. Overriding and Overloading

Overriding is replacing a method of the superclass with a new method in the subclass. The new method in the subclass is automatically called instead of the superclass method.

To override a method in the base class, subclass needs to define a method with same method name and same number of parameters as method in base class.

```
class one ( ):
def init (self): self. a = 1
def method_one(self): print("method_one from class one")
class two (one):
def init (self): self. b = 1
def method_one(self): print("method_one from class two")
c = two()
c. method_one ( )
```

Video 11:

Multiple Inheritance and Encapsulation

You may watch this video first and then attempt Activity 7.2. URL: https://youtu.be/sXKrnYdDJ8w

Activity

Activity 7.2

A CEO buys a car. Later on the CEO buys two new cars BMW and a Mercedez. There is a driver for the CEO who chooses a car to drive to the office..

1. Identify the classes involved in this scenario.
2. Select appropriate superclass and subclasses
3. Implement move method inside the superclass.
4. Invoke the move method in superclass by creating instances of subclasses from a sub class.
5. Implement the move method inside the subclasses.
6. Override the move methods by creating instances of sub class.

Unit summary

Summary	In this unit you learned many fundamentals concpts of Object Oriented Programming. We discussed inheritance, customization with subclasses, Polymorphism and encapsulation in Python with real world examples. In this unit you have also learned ways to reuse programming codes with inheritance.

References and Further Reading

1. Python for Everybody, Oject oriented programming, accessed web (2018), https://www.py4e.com/html3/14-objects
Copyright CC-BY 3.0 - Charles R. Severance

2. Allen B. Downey (2012). Think Python. http://greenteapress.com/wp/think-python/
Download this book for free at http://greenteapress.com/thinkpython/thinkpython.pdf

Unit 8: Error and Exemption Handling

8

Unit Structure

8.1 Errors

8.2 Semantic Errors

8.3 Exceptions

8.4 Exception Handling

Introduction

In this unit we discuss about different types of errors that can occur in a program, how to identify them and how to correct them. The three types of errors that will be discussed in this unit are: Syntax errors, Runtime errors and Semantic errors. The latter part of this unit will discuss the definition of exception and the importance of handling it using appropriate techniques.

Upon completion of this unit, you will be able to:

Outcomes	• Identify the possibilities where programs can lead to unexpected outcomes • Describe the techniques used to distinguish among the type of errors • Describe the errors and exceptions handling mechanisms in Python • Write robust code using appropriate error handling techniques
Terminology	Error: Mistake done while writing the program Fault: Manifestation of an error Recursion: Process of defining a function or calculating a number by repeated application of an algorithm.

8.1 Errors

Since there are many kinds of errors that can occur in a program, it is highly important to distinguish them and solve them. The following are the types of errors that occurs in a Python program. These errors can be found in the process of debugging:

- Syntax errors occur when translating the source code into byte code. These errors indicate the mistakes in the syntax of a program such as using a keyword as a variable or misspelling a keyword. For example SyntaxError: invalid syntax.
- Runtime errors appear when the program is run. They are produced by the interpreter and are also called exceptions as they mean something exceptional has happened. Runtime error messages display where the error occurred after execution of which function. For example:
- Semantic errors will not make the program stop immediately. Instead they will let the program run without displaying any error messages but produce an incorrect result. For example, one statement may not be executed in the program but will produce a result which is incorrect.

8.1.1 Syntax Errors

Syntax errors are the ones which is easiest to solve when you find them. Sometimes, syntax error messages are vague and it is not easy to understand what has happened. For example, messages like SyntaxError: invalid syntax and SyntaxError: invalid token, do not give any useful information to fix the problem.

However, the good thing about this is that the message informs there is an error at one particular point in the program. The location pointed may not be very accurate but it will be closer to where the error is and is usually the preceding line.

Example 8.1:

```
>>> while True print 'Welcome to Python'
SyntaxError: invalid syntax
```

Here are some guidelines to avoid some common syntax errors:

1. Do not use Python keywords as variable names.
2. Check whether Python keywords are misspelled.
3. Check whether strings matching quotation marks (‘ ’).
4. Make sure that “:” is there at the end of the header of every compound statement like class, for, while, if, and def statements.
5. Check whether all strings in the code have matching quotation marks.
6. Any unclosed parenthesis such as (, {, or [makes Python continue with the next line as part of the current statement.

Proper indentation makes sure that statements are written in the way they are supposed to be executed. It enables programmers to detect mistakes before running the program. Since, there could be issues in mixing spaces and tabs, use a text editor that generates consistent indentation.

Especially, it is useful to give serious consideration when writing and using variables. It is advisable to write functions that use global variables first. In a new class, always encapsulate related variables as attributes and transform the associated functions into methods.

8.1.2 Runtime Errors

Once the program is syntactically correct, it will start running. Following are a few factors that would occur when runtime errors occur.

1 If the program does nothing, it could be because the functions and classes in the program do not actually invoke anything to start execution.

2 If your program is caught in an infinite loop or infinite recursion, it would also not do anything.

Infinite loop

To make sure that the loop is excuted, we can write a few print statements immediately before the loop and after the loop that says “entering the loop” and “exit loop” respectively. If only first message is displayed and not the second, that means the loop is not ending. Never ending loops are also called ‘Infinite Loops’.

Example 8.2:

```
1. Print('Entering the loop')
2. x = 14
3.     while x < 15:
4.     # write something to do with x but do not decrement
5.     print "x: ", x
6.     print('Exit loop')
```

When the program is run you will see value of x being oriented without stopping. Then you can stop the program and add,

x= x - 1

Then you will see the last print statement.

Infinite recursion

If there is an infinite recursion, it will cause the program run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. That is, there should be some condition that causes the function or the method to return without making a recursive invocation. Then you must identify the base case. Like with the infinite loop, you can add print statements at the beginning of the function and in the middle so that an output is printed every time the function or the method is invoked.

Even if you do not get this error but suspect there would be an issue with a recursive method or function, can verify it by adding print statements as discussed above.

When something goes wrong during runtime, a message will be printed including the name of the exception, (this would be discussed under the section 'Exceptions') and the line of the program where the problem occurred. It can be traced back to the place where the error occurred.

8.2 Semantic Errors

Semantic erors are the logical errors or problems in the algorithm. It is hard to debug these as the interpreter provides no information about the problem. Only the programmers know what the program is expected to do. It would be easy if you can make a connection between source code and the output and then start debugging. One way to do this is to add a few well-placed print statements and the other is to setup the debugger, inserting and removing breakpoints, and execute the program step by step.

To verify the correctness of an algorithm, break it into smaller programs or modules and test each module separately. Then test the integrated system.
For example, lets look at a scenario for calculating the average of three numbers.
Example 8.3

```
num1 = float(input('Enter a number 1: '))
num2 = float(input('Enter a number 2: '))
num3 = float(input('Enter a number 3: '))
average = num1+num2+num3/3
print ('The average of the three numbers is:', average
```

Say that inputs are given as 5,6,8, then the output of the above example is:

```
('The average of the three numbers is:', 13.66666666666666 )
```

But the output of the actual calculation, i.e. average of 3 numbers should be:

```
('The average of the three numbers is:', 6.333333333333333)
```

Therefore the output of the program stated above is incorrect because of the order of operations in arithmetic.

In order to rectify this problem, the following parenthesis should be added:

average= (num1+num2+num3)/3

Hence the program should be as Example 8.4.

Example 8.4:

```
num1 = float(input('Enter a number: '))
num2 = float(input('Enter a number: '))
num3 = float(input('Enter a number: '))
average= (num1+num2+num3)/3
print ('The average of the the three numbers is:',average)
```

The above example would give you an idea of how a program may not give you an error, but logically be incorrect.

Activity	Activity 8.1
	List the different types of errors and explain how you can identify them separately.

8.3 Exceptions

Programming errors detected during the excution are called exceptions. An exception modifies the flow of the program due to a fault. These exceptions are generally not handled by programs, and Python interpreter display error messages as shown below.

Example 8.5:

```
>>>12*3/0
Traceback (most recent call last):
        File "<pyshell#14>", line 1, in <module>
12 *3/0
```

```
ZeroDivisionError: division by zero
>>> 111 + num
Traceback (most recent call last):
        File "<pyshell#14>", line 1, in <module>
NameError: name 'spam' is not defined
>>> 111 + '222'
Traceback (most recent call last):
        File "<pyshell#14>", line 1, in <module>
TypeError: unsupported operand type(s) for +: int and str
```

Usually, the last line of the error message indicates the type of the exception such as ZeroDivisionError, NameError and TypeError. For built-in exceptions there are standard exception names. However, for user-defined exceptions names must be provided and it is good to adhere to standard practices.

Following are a set of exceptions used as base classes for many other built-in exceptions.

- **exception BaseException**

In Python all exceptions are considered to be instances of a class derived from this exception. But it is not meant to be directly inherited by user-defined classes. If _str_ is called on an instance of this class, the set of the argument(s) to the instance is returned, or the empty string if there are no arguments.

args - some built-in exceptions expect a number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

- **exception Exception**

All exceptions that are built-in but not exiting the system, are derived from this class. All user-defined exceptions should also be derived from this class.

- **exception ArithmeticError**

This is the base class for built-in exceptions which are raised for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError.

- **exception BufferError**

This is raised when a buffer related operation cannot be performed.

- **exception LookupError**

This is the base class for the exceptions that are raised when a key or index used on a mapping or sequence is
invalid: IndexError, KeyError. This can be raised directly by codecs.lookup().
There is a set of exceptions called 'Concrete exceptions' which are raised frequently. They are, exception AssertionError, exception AtrributeError, exception NameError, exception TypeError and exception IndexError.
Other than these there are many other built-in exceptions.

Activity	Activity 8.2
	What are exceptions and why is it important to handle them appropriately? State with examples..

8. 4 Exception Handling

Python exceptions are written in a hierarchical structure. Figure 8.1 from the Python Library Reference depicts how an exception starts at the lowest level possible (a child) and travels upward (through the parents), waiting to be caught.
When an exception is met, the Python program would terminate. Then you have to catch the error to see what went wrong. When programming, if you do not know what types of exceptions may occur, you can always just catch a higher level exception.
For example, if you did not know that ZeroDivisionError from the previous example was a "stand-alone" exception, you could have used the ArithmeticError for the exception and caught that. As the Figure 8.1

shows, ZeroDivisionError is a child of ArithmeticError, which in turn is a child of StandardError, etc.

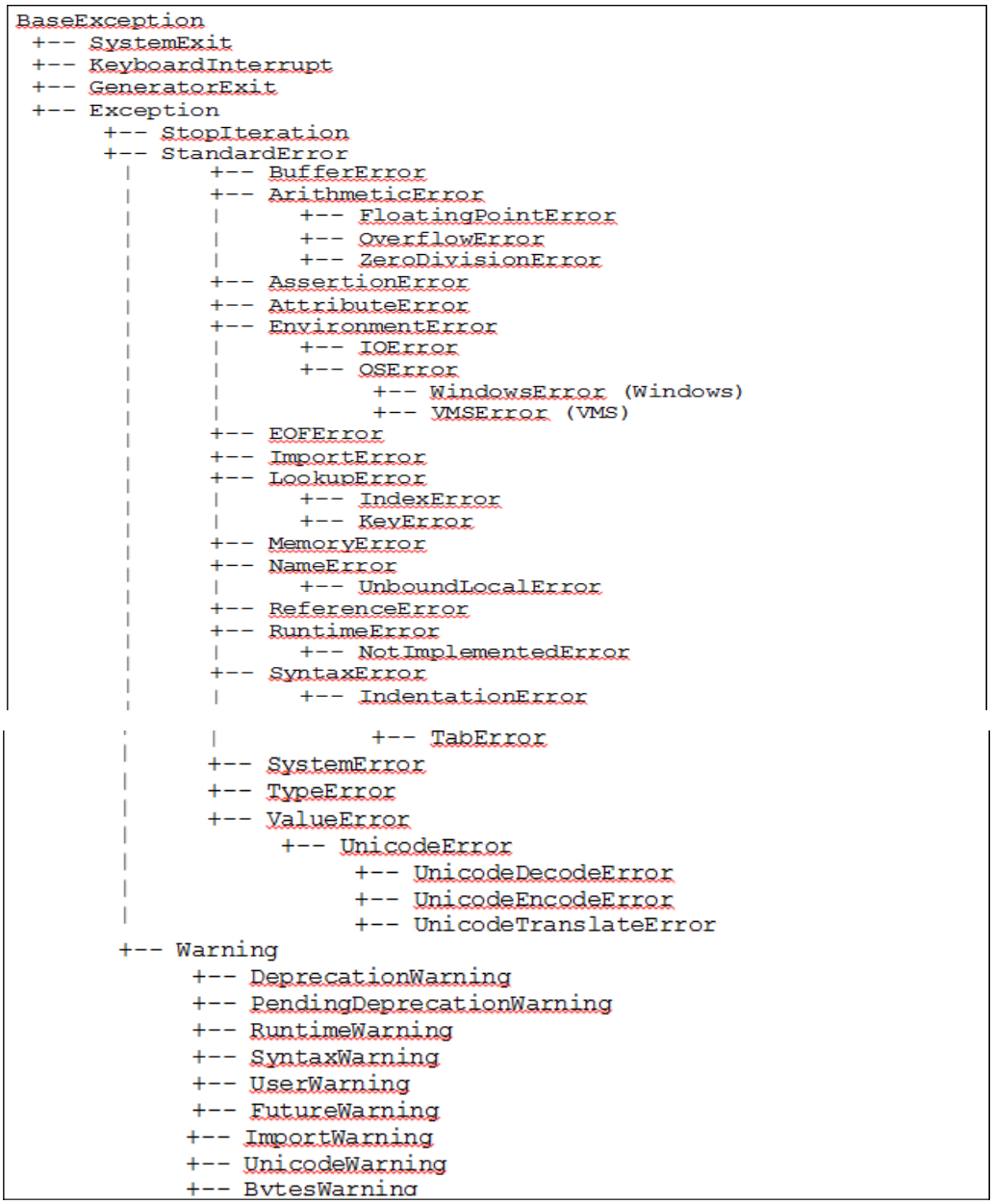


Figure 8.1: Exception Hierarchy

Reference: https://docs.python.org/2/library/exceptions.html#bltin-exceptions

We can handle multiple exceptions in a similar way. For example, suppose we plan to use ZeroDivisionError and include FloatingPointError as well. If we want to have the

same action taken for both errors, can simply use the parent exception ArithmeticError as the exception to catch. That way, when either a floating point or zero division error occurs, we don't need to have a separate case for each one.

8.4.1 User-Defined Exceptions

You can create new exceptions by creating a new exception class. They are defined similar to other classes usually offering a number of attributes that allow information about the error to be extracted by handlers for the exception. For different error conditions, specific subclasses should be created as shown in Example 8.6.

Example 8.6

```
class Error(Exception):
        """Base class for exceptions in this module."""  pass
class InputError(Error):
        """Exception raised for errors in input. Attributes:
        Expression - input expression in which the error occurred
        message - explanation of the error """
        def  init (self, expression, message): self.expression = expression self.message
= message

class TransitionError(Error):
        """Raised when an operation attempts a state transition that's not allowed.
Attributes:
        previous - state at beginning of transition next - attempted new state
        message - explanation of why the specific transition is not allowed   """
def  init (self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Reference: https://docs.python.org/3/tutorial/errors.html

Exceptions are usually defined with names that end in “Error,” as per the naming standard for exceptions. Many standard modules define their own exceptions to report errors that may occur in functions they define.

Activity	Activity 8.3
	Describe what user-defined exceptions are.

Unit summary

Summary	The objective of this study unit is to introduce the types of errors and the techniques to distinguish among them in order to resolve these errors. In the first part of this unit, we discuss definitions of errors and the techniques commonly used to identify such errors. In the the latter part of the unit we discuss error handling which gives an overview of what exceptions are and the techniques to handle them efficiently. You also get to know about user-defined exceptions that are supported by Python.

References and Further Reading

1. ErrorsandExceptions,Retreived10th June 2017, https://docs.python.org/2/library/exceptions.html#bltin-exceptions https://docs.python.org/3/tutorial/errors.html

2. Downey, A. (2013). Think Python. Chapter 8. Needham, MA: Green Tea Press. Retrieved March 18, 2017, from green tea press, http://greenteapress.com/wp/think-python/

Unit 9: Testing 9

Unit Structure

9.1 Software Testing

9.2 Testing Methods: Black box and White box

9.3 Testing Frameworks for Python

9.4 Example of Python Unit Testing

Introduction

This session introduces you to the different testing methods that can be used with any programming language as well as Python. The two main menthods we discuss here are black box testing and white box testing. This unit also gives an overview of the test framework called unit-test for Python. Along with it, we examine few examples on how to write a simple test case.

Upon completion of this unit you will be able to:

Outcomes	• Explain testing methods used in Python • Differentiate black box testing and white box testing • Select a suitable test framework for Python • Write simple test cases in Python
Terminology	Recursion: A function applied within its own definition Framework An abstraction in software providing a generic functionality that can be modified.

9.1 Software Testing

Testing can be defined as execution of a software for the purpose of verification and validation. Verification is the process of making sure that the product is build the right way whereas validation is the process of making the right product. Testing strategies vary in their effectiveness at finding faults. There is no direct relationship between testing and reliability. However, more stringent the level of testing the system survives, the more reliable it is likely to be.

The popular representation of the software life cycle shown in Figure 9.1 depicts the role of testing towards the end of development. It also illustrates how activities in the first half of the life cycle contribute to testing, by providing test specifications, at various levels. For example, the detailed design stage produces a test specification for the unit (or module) test,

which determines how the code should be tested for conformance to the design.

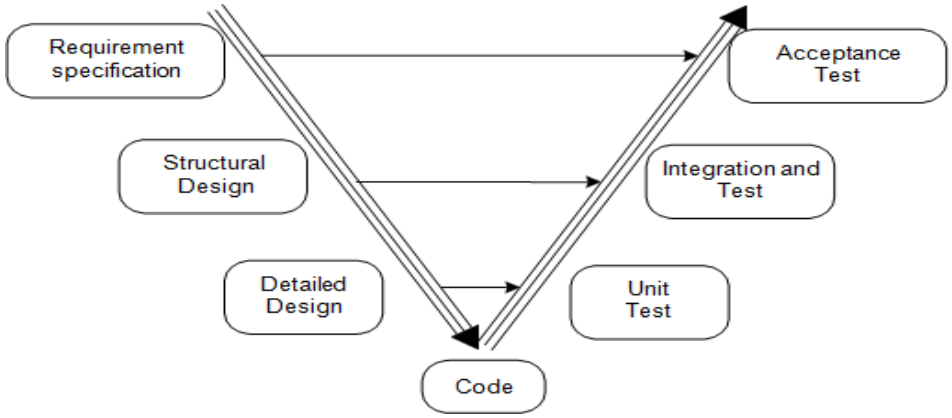


Figure 9.1 Testing in various stages of the life cycle

In terms of intermediate products, there are two inputs to any testing stage: the product to be tested, and the relevant test specification.

In general, the principal stages of testing during software development can be summarised as follows.

Testing during the development

- **Unit or modules testing -** Completed modules Vs module designs
- **Integration testing -** Integrated program Vs system design

- **Functional testing -** Completed system Vs system specification
- **System testing -** Completed system Vs objectives

First, module tests are carried out as a check on the coding of module designs.
Next, the integrated software is tested against its higher level design. Then, the functionality of the system it is tested against the specification.
Finally, the system test addresses non-functional aspects of the specification, like performance and reliability.
These stages of testing are used by the developer with the aim of finding and eliminating faults before the system is delivered to the customer or the marketplace. Following are the stages of testing after development:

Testing after development

- **Acceptance testing -** Completed system Vs requirements of real users
- **Alpha test -** User and developer test the system using real data
- **Beta test -** Release of product to a section of the market for real use and fault reporting
- **Installation testing -** Tests to check the installation process
- **During use - Using spare** capacity to do additional automatic testing

The precise selection of which of these post-development test stages to use depends on the kind of system and whether it is critical. Often, these decisions are driven by market considerations rather than technical reasons.

Functional and Non-functional Testing

Functional testing verifies that all the specified requirements have been incorporated. These are designed to determine that the developed software system operates the way it should. Non-functional testing is designed to find out whether your system will provide a good user experience.

Examples of non-functional tests include:

- Load/Performance testing
- Compatibility testing

- Localisation testing
- Security testing
- Reliability testing
- Stress testing
- Usability testing
- Compliance testing

9.2 Testing Methods: Black box and White box

There are many testing methods. In this unit we will consider two main types called Black Box Testing and White Box Testing.

In black box testing, test cases are derived from the requirements without reference to the code itself or its structure. Here, the program is treated as a black box and testers check whether expected output is given for a selected set of inputs.

In white box testing, test data are derived from the internal logic of the program such as endless loops and wrong branching statements. The structure of a program is said to be tested exhaustively if every possible

path through the software has been executed at least once. However even this 'exhaustive' testing is not guaranteed to activate every possible fault. Faults that depend on specific data values could still remain: e.g. if the correct line of code which find the absolute value of x-y and compare it with contstant_p,

if abs(x-y) < constant_P,

were to be erroneously replaced by

if (x-y) < constatnt_P

The error would remain undetected if all test cases exercising this part of the code had x > = y.

Moreover, in most real-life cases, the number of paths through a program is very large. Therefore, something less than full path coverage is usually chosen.

Next, we will discuss an important black box testing technique called boundary value analysis.

9.2.1 Boundary Value Analysis

Here we choose test cases directly on, above and beneath the boundary of the inputs. Example:

For x valid if $-1.0 \leq x \leq 1.0$

test the boundary values such as $x = -1.0, 1.0, -1.001, 1.001$

i.e. in general min, min-1, max, and max+1

It is known that many software faults occur at the boundaries of input domains.

Video 7: **Software Testing Types**	
	You may watch this video and then attempt Activity 9.1. URL: https://youtu.be/GiHIZPpsoTY

Activity Activity 9.1

Explain the difference between white box testing and black box testing.

9.3 Testing Frameworks for Python

Python has a unit testing framework called unittest which is sometimes referred to as “PyUnit”. It had been inspired by Junit for Java.

unittest is important as it supports test automation. It provides other facilities such as sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. It also provides classes that make it easy to support these features.

test fixture - represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case - is the smallest unit of testing and checks for a specific response to a particular set of inputs. unittest provides a base class called TestCase which can be used to create new test cases.

test suite - is a collection of test cases that should be executed together.
test runner - is a component which coordinates the execution of tests and provides the outcome to the user. The runner may use a graphical user interface, a text interface, or return a special value to indicate the results of executed tests.

TestCase and FunctionTestCase classes supports the concepts test case and test fixture, respectively. The former should be used when creating new tests, and the latter can be used when integrating existing test code with a unittest-driven framework. When building test fixtures using **TestCase,** the setUp() and tearDown() methods can be overridden to provide initialization and cleanup for the fixture.

With **FunctionTestCase,** existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it is successful, the cleanup method is run whatever the outcome of the test. Each instance of the TestCase will only be used to run a single test method, so a new fixture is created for each test.

TestSuite class facilitate implementation of test suits. It allows individual tests and test suites to be aggregated; when the suite is executed, all tests are added directly to the suite and in “child” test suites are run.

Activity Activity 9.2

Explain the concepts used in unittest framework.

9.3 Example of Python Unit Testing

The unit test module provides a rich set of tools for constructing and running tests. Here, we will discuss a small subset of those tools sufficient to meet the needs of most users.

Let us examine how we can test few methods used in string objects.

Example 9.1:

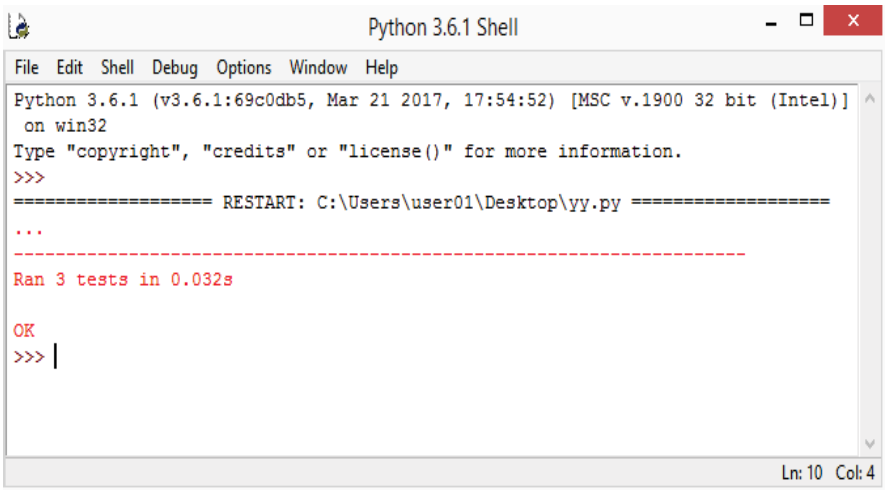
```
import unittest
class TestStringMethods(unittest.TestCase):

    def test_upper(self):
    self.assertEqual('kala'.upper(), 'KALA')

    def test_isupper(self):
    self.assertTrue('KALA'.isupper())
    self.assertFalse('Kala'.isupper())
```

```
        def test_split(self): s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])

#check that s.split fails when the separator is not a string with
self.assertRaises(TypeError):
        s.split(2)
        if  name  == ' main ':

unittest.main()
```

A testcase is created as a subclass of unittest.TestCase. Here, the three individual tests are named with ‘test’ prefix so that these names indicate to the test runner that they represent tests.

At the beginning of each method, assertEqual() is called to check for an expected result; assertTrue() or assertFalse() to verify a condition; or assertRaises() to verify that a specific exception is raised. These methods

The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:



Instead of unittest.main(), there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with the following lines as shown below:

```
suite
=unittest.TestLoader().loadTestsFromTestCase(TestStrin gMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

Running the revised script from the interpreter or another script produces the following output:

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\Users\user01\Desktop\yy.py ==========
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
>>>
```

The above examples show some commonly used unittest features, which are sufficient to meet many of everyday testing needs.

Video 13:

Unit Testing in Python

You may watch this video and then attempt Activity 9.3.

URL: https://youtu.be/agMq0XVkFmk

Activity	Activity 9.3
	Write a test case for a string method that test for a “FOOD”.

Unit summary

Summary	The objective of this study unit is to introduce you to the testing methods used in Python and identify the differences between black box testing and white box testing. It also enables you to familiarise with a suitable framework for Python along with writing simple test cases. Later we discussed individual tests that are defined with methods and a few examples on the ways to run the tests with a finer level of control.

References and Further Reading

1. Jackson, C., & Jackson, C. (2014). Chapter 14. In Learning to program using Python. Utgivningsort okänd: Createspace, Retrieved March 18, 2017, www.sandal.tw/upload/Python_programming_2nd_Edition.pdf **Download thisbook for free at** www.sandal.tw/upload/Python_programming_2nd_Edition.pdf
2. Downey, A. (2013). Think Python. Debugging. Needham, MA: Green Tea Press. Retrieved March 18, 2017, from green tea press, http://greenteapress.com/wp/think-python/ **Download this book for free at** http://greenteapress.com/wp/think-python/ unittest – Unit testing framework https://docs.python.org/2/library/unittest.html

Unit 10: Debugging and Profiling

10

Unit Structure

10.1 Finding and removing programming errors

Software Testing

10.2 Introduction to the profilers

Introduction

This session introduces you to the important modules that handle the basic debugger funtions such as analysing stack frames and set breakpoints etc. It also gives a brief description of what profilers are and how profiling is performed on a Python applications.

Upon completion of this unit you will be able to:

Outcomes	• explain the importance of performing Python profiling • identify the modules that handles the basic debugger functions • use these modules appropriately in the Python program • perform profiling on an existing application
Terminology	**Debugging:** The process of finding and removing programming errors **Framework** An abstraction in software providing a generic functionality that can be modified.

10.1 Finding and removing programming errors Software Testing

Finding and removing programming errors is called debugging in shorten form. It is an important skill that all programmers should acquire since it is an integral part of programming.

In Python, there are two important modules that play a major role in debugging. They are bdb, the debugger framework and pdp, the Python debugger.

The following sections give a brief description on both the modules.

The bdp module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following syntax is used to define the exception, which is raised by the bdb class for quitting the debugger.

exception bdb.BdbQuit

The following class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

class bdb.Breakpoint(self, file, line, temporary=0, cond=None, funcname=None)

Breakpoints are indexed by number through a list called bpbynumber and by (file, line) pairs through bplist. The former points to a single instance of class Breakpoint. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a funcname is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

Breakpoint instances have the following methods:

- deleteMe()

Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

- enable()

Mark the breakpoint as enabled.

- disable()

Mark the breakpoint as disabled.

- pprint([out])

  Print all the information about the breakpoint:

  - The breakpoint number.
  - If it is temporary or not.
  - Its file, line position.
  - The condition that causes a break.
  - If it must be ignored the next N times.
  - The breakpoint hit count.

The other module that is important for debugging is called Python Debugger also known as (pdb). Use of this debugger to track down exceptions will allow you to examine the state of the program just before the error.

The module pdb defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible and it is defined as the class pdb. The extension interface uses the modules bdb and cmd.

Activity Activity 10.1

What is a debugger framework (bdb)? State each function it handles with examples..

The debugger's prompt is (Pdb). Typical usage to run a program under control of the debugger is shown in Example 10.1

Example 10.1

```
>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
><string>(0)?()
(Pdb) continue
><string>(1)?()
(Pdb) continue NameError: 'spam'
><string>(1)?()
(Pdb)
```

pdb.py can also be invoked as a script to debug other scripts. For example:
Python -m pdb myscript.py

When invoked as a script, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After normal exit of the program, pdb will restart the program. Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.
The typical usage to break into the debugger from a running program is to insert,

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the c command.

The typical usage to inspect a crashed program is shown in Example 10.2.

Example 10.2:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last): File "<stdin>", line 1, in <module>
        File "./mymodule.py", line 4, in test
                test2()
        File "./mymodule.py", line 3, in test2
                print spam
        NameError: spam
>>> pdb.pm()
./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

pdb.run(statement[, globals[, locals]])

executes the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type continue, or you can step through the statement using step or next (all these commands are explained below). The optional global and local arguments specify the environment in which the code is executed; by default the dictionary of the module main is used.

pdb.runeval(expression[, globals[, locals]])

Evaluate the expression (given as a string) under debugger control.

When runeval() returns, it returns the value of the expression.

Otherwise this function is similar to run().

pdb.runcall(function[, argument, ...])

Call the function (a function or method object, not a string) with the given arguments. When runcall() returns, it returns
whatever the function call returned. The debugger prompt appears as soon as the function is entered.

pdb.set_trace()

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

pdb.post_mortem([traceback])

Enter post-mortem debugging of the given traceback object. If no traceback is given, it uses one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

pdb.pm()

Enter post-mortem debugging of the traceback found in sys.last_traceback.

The run* functions and set_trace() are aliases for instantiating the Pdb class and calling the method of the same name. If you want to access further features, you have to do this yourself:

class pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip= None)

Pdb is the debugger class.

The completekey, stdin and stdout arguments are passed to the underlying cmd.Cmd class;

The skip argument, if given, must be an iterable of glob-style module name patterns. (Glob module finds all path names matching a specified pattern) The debugger will not step into frames that originate in a module that matches one of these patterns.

Example call to enable tracing with skip:

import pdb; pdb.Pdb(skip=['django.*']).set_trace()

10.2 Introduction to the profilers

A profile is a set of statistics that describes how often and for how long various parts of the program are executed. These statistics can be formatted into reports via the pstats module.

cProfile and profile provide deterministic profiling of Python programs.

The Python standard library provides three different implementations of the same profiling interface:

1. cProfile is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on lsprof, contributed by Brett Rosen and Ted Czotter.

Available in Python version 2.5.

3. profile, a Python only module whose interface is imitated by cProfile, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

Changed in Python version 2.4: Now also reports the time spent in calls to built-in functions and methods.

4. hotshot was an experimental C module that focuses on minimising the overhead of profiling, at the expense of longer data post-processing times. It is no longer maintained and may be dropped in a future version of Python.

The profile and cProfile modules export the same interface, so they are mostly interchangeable; cProfile has a much lower overhead but is newer and might not be available on all systems. cProfile is really a compatibility layer on top of the internal _lsprof module. The hotshot module is reserved for specialized usage.

Note: The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python code.

Activity Activity 10.2

What is a Python debugger (pdb) and what are the debugging functionalities it supports?

How to Perform Profiling

To profile a function that takes a single argument, you can do:

```
import cProfile import re

cProfile.run('re.compile("test|bar")')
```

Use profile instead of cProfile if the latter is not available on your system.

The above action would run re.compile() and print profile results like the following figure:

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

Figure 10.1: Profile Results

The first line indicates that 197 calls were monitored. Of those calls, 192 were primitive, meaning that the call was not induced via recursion. The next line: 'Orderedby: standard name', indicates that the text string in the far right column was used to sort the output. The column headings include:

- **ncalls**

for the number of calls,

- **tottime**

for the total time spent in the given function (and excluding time made in calls to sub-functions)

- **percall**

is the quotient of tottime divided by ncalls

- **cumtime**

is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

- **percall**

is the quotient of cumtime divided by primitive calls

- **filename:lineno(function)**

provides the respective data for each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the run() function:

```
import cProfile import re

cProfile.run('re.compile("test|bar")', 'restats')
```

The pstats.Stats class reads profile results from a file and formats them in various ways.

The file cProfile can also be invoked as a script to profile another script.

For example:

```
Python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

-o writes the profile results to a file instead of to stdout

-s specifies one of the sort_stats() sort values to sort the output by. This only applies when -o is not supplied.

The **pstats** module's **Stats** class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
p = pstats.Stats('restats') p.strip_dirs().sort_stats(-1).print_stats()
```

The strip_dirs() method removed the extraneous path from all the module names. The sort_stats() method sorted all the entries according to the standard module/line/name string that is printed. The print_stats() method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

p.sort_stats('cumulative').print_stats(10)

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand which algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

p.sort_stats('time').print_stats(10)

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

p.sort_stats('file').print_stats(' init ')

```
p.print_callees()

p.add('restats')
```

Activity Activity 10.3

What is profiling?
Refering to the examples above, profile a function that takes in a single argument.

Unit summary

Summary	The objective of this unit is to introduce you to important modules that handle the basic debugger functions and it also provides few examples in order to use these debugger functions appropriately. Further it provides an introduction on how to perform profiling in Python along with examples that will help you achieve it.
References and Further Reading	
	1. pdp – The Python Debugger https://docs.python.org/2/library/pdb.html 2. Profiling - https://developer.android.com/studio/profile/battery-historian

Unit 11: Handling Data with Python 11

Unit Structure

11.1 What is a Database?

11.2 Database Concepts

11.3 Introduction to SQLite

11.4 SQL CRUD statements

11.5 Introduction to database constraints

Introduction

In this unit you will learn about the importance of databases in real world applications and importance of using SQLite with Python.

First, you will learn about integrating database concepts into real world applications and data management using SQLite. Next, you will learn how to use these different techniques and when to use them.

It is more complicated to write the code to use a database to store data than Python dictionaries or flat files so there is little reason to use a database unless your application truly needs the capabilities of a database.

The situations where a database can be quite useful are:

(1) When your application needs to make many small random updates within a large data set,

(2) When your data is so large that it cannot fit in a dictionary and you need to look up information repeatedly, or

(3) You have a long-running process that you want to be able to stop and restart and retain the data from one run to the next.

You can build a simple database with a single table to suit many application needs, but most problems will require several tables and links/relationships between rows in different tables. When you start making links between tables, it is important to do some thoughtful design and follow the rules of database normalization to make the best use of the database's capabilities. Since the primary motivation for using a database is that you have a large amount of data to deal with, it is important to model your data efficiently so your programs run as fast as possible.

On completion of this unit, you will be able to integrate these data management functionalities to Python applications that you develop.

Upon completion of this unit you will be able to:

Outcomes	• Setup database designing environment for SQLite. • Identify the constraints when designing a database.

	• Create a database using SQLite and connect with a Python application. • Use appropriate CRUD operations to manipulate data.
 Terminology	**DBMS:** A DataBase Management System (DBMS) is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data. **CRUD:** create, read, update, and delete (as an acronym CRUD) are the four basic functions of persistent storage. **SQL Constraint:** SQL constraints are used to specify rules for the **data in a table.** Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table.

11.1 What is a Database?

A database is a collection of 'well-organised' information, so that it can be easily accessed, managed and updated. Inside a database, data is organized into rows, columns and tables (like a spreadsheet), and it is indexed to make it easier to find relevant information. Most of the e-commerce sites and other types of dynamic websites are experiencing this flexible feature in their applications.

Normally, a database management system provides users with the ability to control, read and/or write data in a database. Data get updated, expanded and deleted as new data are added. Databases are capable of creating and updating themselves,

querying the data that they contain and running applications against them. Normally, a database manager provides users with the ability to perform the above tasks.

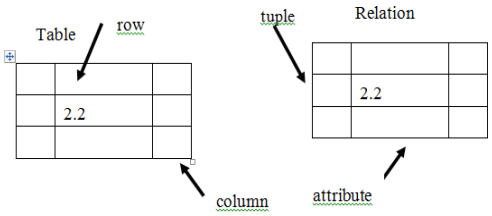
There are many different database management systems (DBMS) including: Oracle, MySQL, Microsoft SQL Server, and SQLite which are used for a wide variety of purposes.

We will focus on SQLite DBMS in this session because, it is a very common database and is already built into Python. SQLite is designed to be embedded into other applications to provide database support within the application. The Firefox browser also uses the SQLite database internally as do many other products.

You will learn more about integrating database systems into applications in a later unit. In the following section, we will be discussing about the basic idea behind the database concepts.

11.2 Database Concepts

As we discussed before, when you first look at a database it will look like a spreadsheet with multiple sheets. The primary data structures in a database are: tables, rows, and columns. In technical terms, the concepts of table, row, and column can be more formally referred to as relation, tuple, and attribute, respectively.



Now, let us get familiar with these new terms.

- Relation: An area within a database that contains tuples and attributes. More typically called a “table”.
- Tuple: A single entry in a database table that is a set of attributes. More typically called “row”.

- Attribute: One of the values within a tuple. More commonly called a “column” or “field”.

11.3 Introduction to SQLite

SQLite is an embedded and easy to use relational database engine. It is a self-contained, server-less, zero-configuration and transactional SQL database engine. It is very fast and lightweight, and the entire database is stored in a single disk file. SQLite is used in many applications as internal data storage. The Python Standard Library includes a module called "sqlite3" intended for working with this database. This module is a SQL interface compliant with the DB-API 2.0 specification.

11.3.1 Installing SQLite manager on Firefox

Since this unit focuses on developing mobile applications using Python to work with data in SQLite database files, many operations can be done more conveniently using a Firefox add-on called the SQLite Database Manager, which is freely available on:

https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/

Using the browser you can easily create tables, insert data, edit data, or run simple SQL queries on data in the database.

11.3.2 Cre Creating a database Connection

To use the SQLite3 module we need to add an import statement to our Python script:

```
import sqlite3
```

When we create a database table we must tell the database that the names of each column and the type of data which we are planning to store in each column.

Various data types supported by SQLite are depict in below table:

Datatypes
VARCHAR(10),
NVARCHAR(15)
TEXT
INTEGER
FLOAT
BOOLEAN
CLOB
BLOB
TIMESTAMP
NUMERIC(10,5)
VARYINGCHARACTER(24)
NATIONALVARYINGCHARACTER(16)

The code to create a database file and a table named ‘StudentDetails’ with two columns in the database is shown in Example 11.1

Example 11.1

```
import sqlite3


conn = sqlite3.connect('school.sqlite3')
cur = conn.cursor()
cur.execute('DROP TABLE IF EXISTS StudentDetails')
cur.execute('CREATE TABLE StudentDetails (studentName TEXT, StudentID INTEGER)')
conn.commit() conn.close()
```

The connect operation makes a “connection” to the database stored in the file school.sqlite3 in the current directory. If the file does not exist, it will be created. The reason this is called a “connection” is that, sometimes the database is stored in a separate “database server” than the server on which

we are running our application. In our simple examples, the database will just be a local file in the same directory as the Python code we are running.

A cursor is like a file handler that we can use to perform operations on the data stored in the database. Calling 'cursor ()' is very similar conceptually to calling open() when dealing with text files.

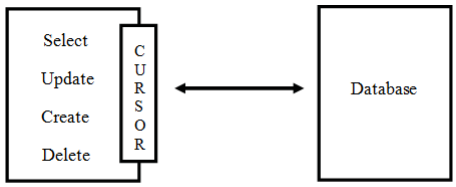


Figure 11.1 Available SQLite commands

Once we have the cursor, we can begin to execute commands on the contents of the database using the execute() method.

Database commands are expressed in a special language that has been standardized across many different database vendors to allow us to learn a single database language. That is called Structured Query Language or SQL for short.

The first SQL command removes the StudentDetails table from the database if it exists.

```
cur.execute('DROP TABLE IF EXISTS StudentDetails')
```

Then the second command will creates a table named 'StudentDetails' with a text column named 'StudentName' and an integer column named 'StudentID'.

```
cur.execute('CREATE TABLE StudentDetails (StudentName TEXT, StudentID INTEGER)')
```

11.4 SQL CRUD statements

Insert Command

Now that we have created a table named StudentDetails, we can insert or enter some data into that table using the SQL INSERT operation. The SQL INSERT command

indicates which tables we are using and then defines a new row by listing the fields we want to include (StudentName, StudentID) followed by the VALUES (we specify the values as question marks (?, ?)) and a list of corresponding values for each of the fields.

Example 11.2

```
cur.execute('INSERT INTO StudentDetails (StudentName, StudentID) VALUES ( ?, ? )', ( ''ABC Perera ', 15033 ) )
conn.commit()
```

Select Command

Then we use 'SELECT' command to retrieve rows and columns from a database table. The SELECT statement lets you specify which columns you would like to retrieve as well as a WHERE clause to select which rows you would like to see. It also allows an optional ORDER BY clause to control the sorting of the returned rows.

```
cur.execute(SELECT * FROM StudentDetails WHERE StudentID = 3513)
conn.commit()
```

Using * indicates that you want the database to return all of the columns for each row that matches the WHERE clause.

You can also request that the returned rows be sorted by one of the fields as shown in Example 11.3.

Example 11.3

```
cur.execute (SELECT StudentName, StudentID FROM StudentDetails ORDER BY StudentID)
conn.commit()
```

Delete Command

To remove a row, you need to use 'WHERE' clause on a 'DELETE' statement. The WHERE clause determines which rows are to be deleted:

Example 11.4

```
cur.execute(DELETE FROM StudentDetails WHERE StudentID
= 15033)
conn.commit()
```

Update Command

The UPDATE statement specifies a table and then a list of fields and values to change after the SET keyword and then an optional WHERE clause to select the rows that are to be updated. If a WHERE clause is not specified, it performs the UPDATE on all of the rows in the table.

It is possible to UPDATE a column or columns within one or more rows in a table using 'UPDATE' statement as in Example 11.5:

Example 11.5

```
cur.execute(UPDATE StudentDetails SET StudentName = 'XYZ Perera' WHERE
StudentID = 15033)
conn.commit()
```

Alter table Command

The ALTER statement specifies a table name and the database which contains that table along with the ‘RENAME TO’ with a list of fields and values to change after the ALTER keyword.

It is possible to MODIFY a table name in a database using 'ALTER' statement as in 11.6:

Example 11.6

```
cur.execute(ALTER TABLE school.StudentDetails RENAME TO school.StudentInformaiton)
conn.commit()
```

Drop table Command

The DROP TABLE statement specifies a database name after the DROP TABLE keyword

It is possible to DELTE a database using ' DROP' statement as shown in Example 11.7:

Example 11.7’

```
cur.execute (DROP TABLE school.StudentInformaiton) conn.commit()
```

These four basic SQL commands (INSERT, SELECT, UPDATE, and DELETE) allow the four basic operations needed to create and maintain data.

11.5 Introduction to database constraints

When we design our table structures, we can tell the database that we would like it to enforce few rules on it. These rules will prevent us from making mistakes and ensure the accuracy of the data that we insert. There are four major constraints named as

- Key Constraint
- Domain Constraint
- Entity Integrity Constraint
- Referential Integrity Constraint

Example 11.8

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
(id INTEGER PRIMARY KEY, nic TEXT INTEGER, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
(from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

We indicate that the People table must be UNIQUE. We also indicate that the combination of the two numbers in each row of the Follows table must be unique. These constraints keep us from making mistakes such as adding the same relationship more than once.

We can take the advantage of these constraints in the following code: Example 11.9

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
VALUES ( ?, 0)''', ( friend, ) )
```

We add the OR IGNORE clause to our INSERT statement to indicate that if this particular INSERT would cause a violation of the “name must be unique” rule, the database system is allowed to ignore the INSERT. We are using the database constraint as a safety net to make sure that we do not inadvertently do something incorrect.

Similarly, the following code ensures that we don’t add the exact same Follows relationship twice.

```
cur.execute('''INSERT OR IGNORE INTO Follows
(from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

Again, we simply tell the database to ignore our attempted INSERT if it would violate the uniqueness constraint that we specified for the Follows rows.

Key Constraints

Now that we have started building a data model putting our data into multiple linked tables, and linking the rows in those tables using keys, we need to look at some terminology around keys. There are generally three kinds of keys used in a database model.

A logicalkey is a key that the “real world” might use to look up a row. In our example data model, the name field is a logical key. It is the screen name for the user and we indeed look up a user’s row several times in the program using the name field. You will often find that it makes sense to add a UNIQUE constraint to a logical key. Since the logical key is how we look up a row from the outside world, it makes little sense to allow multiple rows with the same value in the table.

A primarykey is usually a number that is assigned automatically by the database. When we want to look up a row in a table, usually searching for the row using the primary key is the fastest way to find the row. Since primary keys are integer numbers, they take up very little storage and can be compared or sorted very quickly.

A foreignkey is usually a number that points to the primary key of an associated row in a different table. An example of a foreign key in our data model is the from_id.

We are using a naming convention of always calling the primary key field name id and appending the suffix _id to any field name that is a foreign key.

Video 14: **Handling Data in Python**	
	This video will further explain data manipulation in Python. You may watch this Video with screencast an attempt Activity 11.1 URL : https://youtu.be/gEMPzwcSfC4

Activity

Activity 11.1

- Create a database using SQLite called ‘ABC_Organization’
- Create a table inside that database and name it ‘Employee’,
- Create following fields in ‘Employee’ table

NameoftheField	Datatype
EmployeeID	Integer(PrimaryKey)
EmpFirstName	Text
EmpLastName	Text
Gender	Boolean
NICNo	varchar(100)

- Insert an employee detail to the table (ex: EmployeeID : 105524, EmpFirstName : PQR, EmpLastName : Fernando, Gender : 1, NICNo : 895562987V)
- Update the first name of the employee to ‘PQWR’ where employee id is 105524
- Select fields EmpFirstName, EmpLastName as EmployeeName and NICNo from Employee table

Unit summary

Summary	This unit covered a lot of ground to give you an overview of the basics of using a database in Python it also provide few examples to create database connection and to write SQL commands. Further this unit discussed how to integrate data management functionalities to Python applications.

References and Further Reading

1. Eng, N., & Watt, A. (2013). Database design. Retrieved July 08, 2016, from BC Open TextBooks, https://opentextbc.ca/dbdesign/chapter/chapter-3-characteristics-and-benefits-of-a-database/

2. Andres Torres, Python. "Introduction To Sqlite In Python | Python Central". Python Central. N.p., 2017. Retrieved 25 January 2017 from

3. Bodnar, Jan. "Introduction To Sqlite - Sqlite Description And Definitions". Zetcode.com. N.p., 2017. Retrieved 28 January 2017 from

Unit 12: Role of Python in Mobile Application Development

12

Unit Structure

12.1 Mobile Application Development Environments

12.2 Uses of Python in Mobile Application Development

12.3 Open Source Python Libraries

12.4 Kivy

Introduction

In this unit we are going to learn the role of Python in mobile application development. First part of this unit will give you an insight to the existing mobile application development environments. Then you will learn the suitable development environments and different open source Python libraries available to support these development environments.

Furthermore, this unit will list the libraries that you can use in Android application development using Python. Kivy is one of the frameworks used by Python programmers to develop mobile applications. At the latter part of this unit, the steps to follow in setting up the application development environment will be explained.

Upon completion of this unit you will be able to:

Outcomes	• identify different mobile application development environments • explain the use of open source Python libraries for rapid development of applications • describe the use of python libraries available for android application development • illustrate the Kivy app architecture using a diagram • set up the application development environment using Kivy
Terminology	Kivy: an open source Python library for developing mobile apps and other multitouch application software pip: pip is a package management system used to install and manage software packages written in Python. wheel: A built-package format for Python OpenGL: Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphicsSQL

12.1 Mobile Application Development Environments

There is a variety of applications in various fields including business, entertainment, utilities, hospitality sector, games and much more, whose apps are made sure to fit to various screen sizes be it a smart phone, a tab or any other device. Android, iOS and Windows are few of the operating systems that run on these devices. Different development platforms are being used when developing mobile applications to run on top of these operating systems. The following table lists some of such development platforms used by developers all over the world.

Table 12.1: List of mobile application development platforms

Mobile Operating System	Mobile Application Development Platform	Programming Language
Android	Android Studio Android SDK, Eclipse	Java
iOS	Xcode IDE	Objective C
Windows	Visual Studio	C#

All the platforms listed above allow developers to implement mobile applications to run on Android, iOS or Windows only. Mobile application developers try to develop applications, making sure to fit to various screen sizes and operating systems.

Cross platform mobile development essentially makes use of frameworks allowing developers to create platform independent mobile applications predominantly utilising already familiar web standards like HTML/ JavaScript/CSS. These frameworks act more or less as a middleware or bridge and provide the platform specific implementation of API in the native programming language for the language of the framework to communicate with the native code of different platforms.

The top tools used for cross-formatting mobile application development are RhoMobile, PhoneGap, Appcelerator, Sencha Touch, MoSync, Whoop, WidgetPad, GENWI, AppMakr, Mippin, SwebApps, MobiCart, etc.

12.2 Uses of Python in Mobile Application Development

As you have already learnt, Python is not widely used in mobile application development. Still Python is used mostly in cross platform development due to its platform independent nature. Python runs on all major operating systems such as Windows, Linux/Unix, OS/2, Mac, Amiga, etc. The uses of Python in mobile application development are given below.

- To write mobile applications to run on multiple platforms
- As a scripting language to run on mobile devices

Android Google provides Android Scripting Environment (ASE) which allows scripting languages including Python to run on Android. QPython is another script engine that also runs on android devices like phone or tablet. It lets your android device run Python scripts and projects.

An example of using Python as a scripting language in Android is explained in this section.

Batterystats is part of the Android framework and collects battery data from any Android device. Battery Historian is an open-sourced project which is available on GitHub, which converts the data collected from Batterystats into an HTML visualisation that can be viewed in a browser.

To work with Batterystats and Battery Historian a Python script can be used and the steps to be followed are given below.

Step 1: Download the open-source Battery Historion Python script from GitHub (https://github.com/google/battery-historian).

Step 2: Unzip the file to extract the Battery Historian folder. Inside the folder, find the historian.py file and move it to the Desktop or another writable directory.

Step 3: Connect your mobile device to your computer.

Step 4: On your computer, open a Terminal window in Android Studio. Step 5: Change to the directory where you've saved historian.py,

for example: cd ~/Desktop

Step 6: Shut down your running adb server by entering the following command.

> adb kill-server

Step 7: Restart adb and check for connected devices by entering the following command. You will see a list of devices attached.

> adb devices

If a list of devices is not seen, then may be your phone is not connected properly. So, connect the phone and turn on USB Debugging. Then you should kill and restart adb.

Step 8: Reset battery data gathering by entering the following command.

> adb shell dumpsys batterystats --reset

When you reset, old battery collection data will get erased. Otherewise, there will be huge output.

Step 9: Disconnect your device from the computer to make sure that it draws current only from the device's battery.

Step 10: Use the particular app for a short time.

Step 11: Connect your phone again

Step 12: See whether your phone is recognised (use > adb devices)

Step 13: Then dump all battery data using the following command. This action may take some time.

> adb shell dumpsys batterystats > batterystats.txt

Step 14: Create a HTML version of the data dump for Battery Historian:

> Python historian.py batterystats.txt > batterystats.html Step 15: Open the batterystats.html file in your browser.

You can open the historian.py file and study how the Python script has been written.

Source: https://developer.android.com/studio/profile/battery-historian.html

12.3 Open Source Python Libraries

The language you choose for mobile development varies depending on many factors. Other than Java, C# and Objective C there are many more languages that support mobile development. Some of the examples are HTML5 for front end, C++ for Android and Windowsdevelopment, Swift to work along with Objective C. In this section we will look at the open source Python libraries available for rapid development of applications.

- Kivy - Kivy is a multi-platform application development kit, using Python
- PyGame - PyGame is a set of Python modules designed for writing games. PyGame allows us to easily program games in Python and port them to an Android application.
- PGS4A - Pygame Subset for Android
- SL4A- The SL4A project makes scripting on Android possible, it supports many programming languages including Python, Perl, Lua, BeanShell, JavaScript, JRuby and shell.
- PyGTK – PyGTK allows to create programs with a graphical user interface using the Python programming language

- WXPython - WXPython is a GUI toolkit for Python programming language. This is implemented as a Python extension module (native code) that wraps the wxWidgets cross platform GUI library, which is written in C++.
- PyQT and PySide – These are the two popular Python bindings for the Qt cross-platform GUI/XML/SQL C++ framework. Qt is a cross-platform application framework that is used for developing application software that can be run on various software and hardware platforms with little or no change in the underlying codebase, while still being a native application with native capabilities and speed.
- QPython - QPython is a script engine which runs Python programs on android devices.
- VPython - VPython allows users to create objects such as spheres and cones in 3D space and displays these objects in a window
- TkInter - TkInter is Python's standard GUI (Graphical User Interface) package.
- As you can see there are many open source libraries to develop different types of applications for multiple platforms. Kivy is one of such platform facilitate application development for multiple platforms. We will learn about Kivy in detail in the next section.

12.4 Kivy

Kivy allows you to write your code once and have it run on different platforms. This section will provide you a guide to get the tools you need, understand the major concepts and learn best practices. As this is an introduction, pointers to more information in developing an application will be given in Unit 13.

Using Kivy on your computer, you can create applications that run on:

- Desktop computers: OS X, Linux, Windows.
- iOS devices: iPad, iPhone.
- Android devices: tablets, phones.

- Any other touch-enabled devices supporting TUIO (Tangible User Interface Objects)

Kivy Architecture

Let us look at the architectural view of Kivy. Knowing the Kivy architecture will help you when developing applications using Kivy platform.

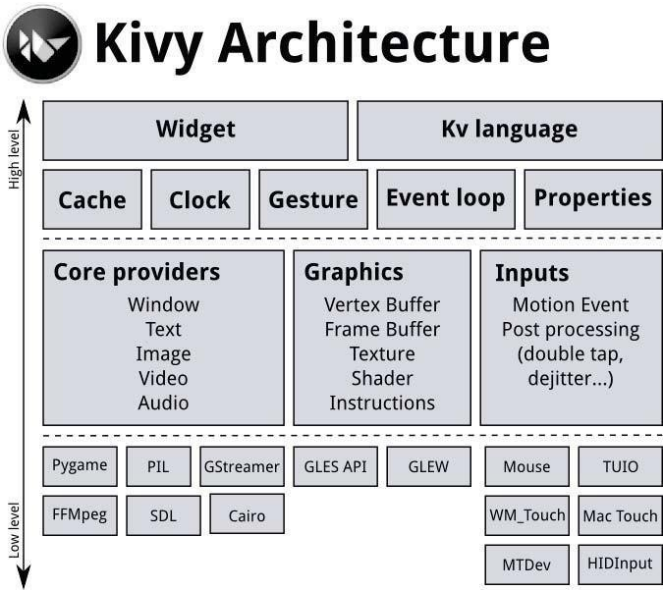


Figure 12.1: Kivy Architecture

Source: https://kivy.org/docs/guide/architecture.html Let's briefly look at the details of following components.

- Core Providers: Core providers are the abstractions of basic tasks or core tasks such as opening a window, displaying text and images, playing audio, getting images from a camera, correcting spelling etc.
- Input Providers: An input provider is a program that facilitate for a specific input device. When a new input device is added, you need to provide a new class that reads the input data from the new device and transforms them into basic events.

- Graphics: Graphics API of Kivy is an abstraction of Open Graphics Library (OpenGL). Within the the software, Kivy issues hardware-accelerated drawing instructions using OpenGL.
- Core: The programs in the core package provides commonly used features, such as calendar, clock, cache, gesture detection, Kivy language and properties. Properties link your widget code with the user interface description. (Kivy language is used to describe user interfaces)
- UIX (Widgets & Layouts): It is the UIX module that contain commonly used widgets and layouts. These can be re-used to create a user interface quickly. You will learn how to import Labels from widgets in the next section while writing your first program in Python.
- Modules: Modules are used to add extra functions into Kivy programs
- Input Events (by touch): Kivy abstracts different input types and sources such as touch screens, mice, and any other Tangible User Interface Objects (TUIO).

Installation of the Kivy environment

To use Kivy you need to install Python first. You may have multiple versions of Python installed side by side, then you have to install Kivy for each version of Python.

1. Before installing Kivy you need to ensure you have the latest pip and wheel. wheel is a packaging format used. For installing them you need to use the following command.

python -m pip install --upgrade pip wheel setuptools

2. Next you need to install the dependencies.

wheels are available for dependencies separately so only necessary dependencies need to be installed. The dependencies are offered as optional sub packages of kivy.deps, e.g. kivy.deps.sdl2

Currently on Windows, the following dependency wheels are given:

- gstreamer for audio and video
- glew for OpenGL, if you are using Python 3.5 angle can be used instead of glew
- sdl2 for control and/or OpenGL

To install the dependencies you need to use the following command:

```
python -m pip install docutils pygments pypiwin32
kivy.deps.sdl2 kivy.deps.glew
python -m pip install kivy.deps.gstreamer
```

3. Once the dependencies are installed, the environment is ready to install Kivy.
   To install Kivy the following command to be used.
   python -m pip install kivy
4. Now we can import kivy to python or run a basic example.
   Let us us a sample Python program given in kivy-examples now.
   python share\kivy-examples\demo\showcase\main.py

You will get the following output.

Figure 12.2 : Output of the main.py program

You will require a basic knowledge of Python to start developing applications using Kivy.

Create an application using Kivy

Creating a kivy application is simple if you are familiar with Python and know how to apply object oriented concepts.

An example of a minimal application is given below.

```
import kivy kivy.require('1.0.6')
# replace with your current kivy version !

from kivy.app import App
from kivy.uix.label import Label

class MyApp(App):
        def build(self):
return Label(text='Hello world')
        if name == ' main ':
        MyApp().run()
```

You can save this to a file, main.py for example, and run it.

If you saved your file inside the Python installation folder, you need to use the following command to run the program.

python main.py

You will find the output of this program as given below.

Figure12.3:Outputoffirst.py

- In order to use Kivy, it's required to import kivy first; line 1 shows how to import kivy.
- The term require can be used to check the minimum version required to run a Kivy application. To run this program you need to have '1.0.6'
- Line 3 is required so that the base class of your App inherits from the App class. It's present in the kivy_installation_dir/kivy/app.py
- In line 4, the uix module is the section that holds the user interface elements like layouts and widgets.

The above program has the following three parts.

- In line 5, sub-classing the App class
- This is where we are defining the Base Class of our Kivy App. You should only need to change the name of your app MyApp in this line.
- In line 6, implementing its build() method so it returns
  a Widget instance. The Label widget is for rendering text. It supports ASCII and unicode strings.
  Here we initialize a Label with text 'Hello World' and return its instance. This Label will be the Root Widget of this App.
- In line 9, instantiating this class, and calling its run() method
- Let us look at the source code for another simple application created to draw circles and lines as shown in the following figure.

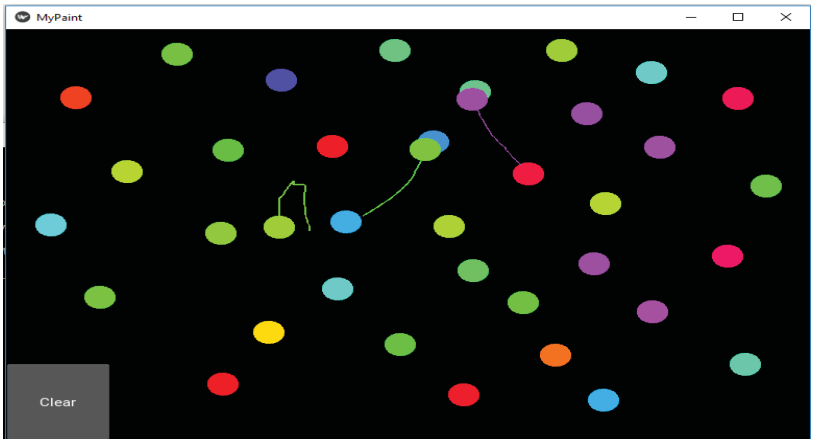


Figure12.4:Outputofpaint.py

```
from random import random                    #Line1
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.button import Button    #Line4
from kivy.graphics import Color, Ellipse, Line


class MyPaintWidget(Widget):
    def on_touch_down(self,touch):
        color = (random(),1,1)          #Line8
        with self.canvas:
            Color(*color, mode='hsv') #Line 10
            d = 30.
            Ellipse(pos=(touch.x-d/2,touch.y-d/2),size=(d,d))
            touch.ud['line']=Line(points=(touch.x,touch.y))

    def on_touch_move(self,touch):
        touch.ud['line'].points+=[touch.x,touch.y]

class MyPaintApp(App):
    def build(self):
        parent = Widget()                                  #Line 18
        self.painter = MyPaintWidget()                     #Line 19
        clearbtn = Button(text='Clear')                    #Line 20
        clearbtn.bind(on_release=self.clear_canvas) #Line 21
        parent.add_widget(self.painter)                    #Line 22
        parent.add_widget(clearbtn)                        #Line 23
        return parent
    def clear_canvas(self,obj):              #Line 25
        self.painter.canvas.clear()          #Line 26

if __name__=='__main__':
    MyPaintApp().run()
```

Source extracted from Kivy documentation in
https://kivy.org/docs/tutorials/firstwidget.html

- Here in Line 1, we import Python's random() function that will give us random values in the range. import statement given as line 4 is required to use the Button class.
- In line 8, we create a new tuple of 3 random float values that will represent a random RGB color. Since we do this in on_touch_down, every new touch will get its own colour.
- In line 10 we set the color for the canvas. We use the random values we generated only at this time and feed them to the colour class using Python's tuple unpacking syntax.

- In Line 18, parent = Widget() is used to create a
- dummy Widget() object as a parent for both our painting widget and the button we're about to add.
- In line 19, we create our MyPaintWidget() as usual, only this time we don't return it directly but bind it to a variable name.
- In line 20, We create a button widget. It will have a label on it that displays the text 'Clear'.

In line 21,we bind the button's on_release event (which is fired when the button is pressed and then released) to the callback function clear_canvas defined on below on lines 25 & 26.

We set up the widget hierarchy in line 22 and 23 by making both the painter and the clearbtn children of the dummy parent widget. That means painter and clearbtn are now siblings in the usual computer science tree terminology.

Up to now, the button did nothing. It was there, visible, and you could press it, but nothing would happen. We change that in lines 25 and 26. We create a small, throw-away function that is going to be our callback function when the button is pressed. The function just clears the painter's canvas' contents, making it black again

Activity	Activity 12.1
	List down and briefly explain the use of five open source Python libraries which are not mentioned in the unit.

Unit summary

Summary	Different types of Python libraries available for rapid application development were explained in this unit. Furthermore the details of Kivy application life cycle and the Kivy architecture were explained. Steps to be followed when writing a Python application using Kivy were explained with examples at the end of the unit. In the next section you will learn how the developed applications using Kivy can be packaged to run on Android devices.

References and Further Reading

Kivy is an Open Source Python Library. https://kivy.org/#home
https://kivy.org/doc/stable/guide/architecture.html
Contributors for Kivy (as given on Kivy.org by March 2019)
- Terje Skjaeveland (bionoid)
- George Sebastian (georgs)
- Gabriel Ortega
- Arnaud Waels (triselectif)
- Thomas Hirsch
- Joakim Gebart
- Rosemary Sebastian
- Jonathan Schemoul

Past core developers
- Thomas Hansen (hansent)
- Christopher Denter (dennda)
- Edwin Marshall (aspidites)
- Jeff Pittman (geojeff)
- Brian Knapp (knappador)
- Ryan Pessa (kived)
- Ben Rousch (brousch)

Special thanks
- Mark Hembrow
- Vincent Autin

Unit 13: Mobile Application Development with Python

13

Unit Structure

13.1 Android Mobile Application Development using Kivy

13.2 Buildozer

13.3 Packaging with Python-for-android

13.4 Packaging your application for the Kivy Launcher

13.5 The Kivy Android Virtual Machine

Introduction

In the previous unit you learnt to set up the development environment for a Kivy application. In this unit you will be learning to develop an application using Kivy and to package it to run on Android devices.

Upon completion of this unit you will be able to:

Outcomes	• Develop a Python application using Kivy • Build the developed application and run it on Android device
Terminology	bootstrap: A bootstrap is a class consisting of few basic components (i.e. with SDL2, Pygame, Webview etc.) Virtual Machine(VM): A virtual machine is an operating system or application environment that is installed on software.

13.1 Android Mobile Application Development using Kivy

A Kivy application can run on Android device. For that we have to compile a Kivy application and to create an Android APK which will run on the device similar to a Java application. It is possible to use different tools which will help to run code on Android devices. We can publish these APKs to Google store where users can download and install in their devices or it is possible to run the apps using a Kivy Launger app. These two methods will be explained further in this section.

- First method is to use the prebuilt Kivy Android VM image, or use the Buildozer tool to automate the entire process.
- The second method is to run the Kivy app without a compilation step with the Kivy Launcher app

13.2 Buildozer

Buildozer is a tool that allows packing of mobiles application easily. It downloads and sets up all the prequisites for Python-for-android, including the android SDK and NDK, then builds an apk that can be automatically pushed to the device.

Buildozer currently works only in Linux. The steps to follow when using Buildozer on Linux are given below.

You can get a clone of Buildozer from https://github.com/kivy/buildozer For that you need to use the following commands in Linux

git clone https://github.com/kivy/buildozer.git cd buildozer sudo python2.7 setup.py install

This will install Buildozer in your system. Afterwards, navigate to your project directory and run:

buildozer init

This creates a buildozer.spec file controlling your build configuration. You should edit it appropriately with your app name etc. You can set variables to control most or all of the parameters passed to Python for android.

You need to install buildozer's dependencies and then you need to plug in your android device and run the application using the following command. This command will build, push and automatically run the apk on your device.

buildozer android debug deploy run

13.3 Packaging with Python-for-android

Python-for-android is also represented as P4A. To install P4A you need to use the following command. This section explains how to do this in Linux environment. The

same process can be done in Windows on top of a Virtual Machine. This process is explained at the later part of this unit.

Python-for-android can be installed using the following command.

Python –m pip install python-for-android

Then you need to install the dependencies you need. Some of the dependencies are given below.

- git
- ant
- python2
- cython
- a Java JDK
- zlib
- libncurses unzip
- virtualenv (can be installed via pip)
- ccache (optional)

You need to download and unpack the Android SDK and NDK to a directory (let's say $HOME/Documents/)

Then, you can edit your ~/.bashrc or other favorite shell to include new environment variables necessary for building on android

```
# Adjust the paths!
export ANDROIDSDK="$HOME/Documents/android-sdk-21"
export ANDROIDNDK="$HOME/Documents/android-ndk-r10e"
export ANDROIDAPI="14" # Minimum API version your application require
export ANDROIDNDKVER="r10e" # Version of the NDK you installed
```

13.4 Packaging your application for the Kivy Launcher

The Kivy launcher can be used to run the Kivy applications on Android devices without compiling them. Kivy launcher runs the Kivy examples stored on the SD Card in the device. To install the Kivy launcher, you must go to the Kivy Launcher page on the Google Play Store. Then click on Install and select your phone. If not you can go to https://kivy.org/#download and install the APK manually.

Once the Kivy launcher is installed, you can put your Kivy applications in the Kivy directory in your external storage directory. Often the application is available at /sdcard even in devices where this memory is internal. For an example /sdcard/kivy/<your application>

<your application> should be a directory containing your main application file(e.g. main.py) and a text file (android.txt) with the contents given below.

title= <Application Title>

author=<Your name>

orientation=<portrait|landscape>

13.5 The Kivy Android Virtual Machine

So far you learnt to build a Kivy Android application in a Linux environment configured with Python-for-android, the Android SDK and the Android NDK. It is possible to use a fully configured VirtualBox disk image in Windows and OS X operating system, because using the previous methods were limited only for Linux based developers.

Setting up the environment

Step 1 : Download the disc image from https://kivy.org/#download, in the Virtual Machine section. A Virtual Machine with Android SDK and NDK and all other pre-requisites pre installed to ease apk generation.

The size of the download is more than 2GB and around 6GB after extracted. Extract the file and remember the location of the extracted vdi file.

Step 2 : Download the version of VirtualBox for your machine from the VirtualBox download area and install it.

Step 3 : Start VirtualBox, click on “New” in the left top. Then select “linux” and “Ubuntu 64-bit”. You need to check the available Linux distribution and select appropriately.

Step 4 : Under “Hard drive”, choose “Use an existing virtual hard drive file”. Search for your vdi file and select it. Assigning insufficient memory may result in the compile failing with cryptic errors. Therefore it’s required to assign sufficient memory when creating the virtual machine.

Step 5 : Go to the “Settings” for your virtual machine. In the “Display -> Video” section, increase video ram to 32MB or above. Enable 3D acceleration to improve the user experience.

Step 6 : Start the Virtual machine and follow the instructions in the
readme file on the desktop.

The instructions given in the readme file are given below for your reference. These instructions can be changed based on the virtualbox disk image that you are downloading.

To use it go into your project directory, and use:

buildozer init

you can then edit the created buildozer.spec, to suit your project. Then use

buildozer android debug

to build your project, once you have it built, you can use the

buildozer android deploy run logcat

command to start the app on your device and collect the error log. (use ctrl-c to stop the logcat)

Commands can be combined as given below.

buildozer android debug deploy run logcat

This command will do the whole process.

To update Buildozer you need to use the following command.

sudo pip install -U buildozer

To update Kivy and other modules the simplest way is to remove the buildozer cache before building your distribution. You can do this by using the following command.

rm -rf ~/.buildozer/android/packages

Building the APK

Once the VM is loaded, you can follow the instructions from Packaging with Python-for-android given above.

Generally, your development environment and toolset are set up on your host machine but the APK is build in your guest. VirtualBox has a feature called ‘Shared folders’ which allows your guest direct access to a folder on your host.

Activity

Activity 13.1

- Download the Kivy demos for Android by visiting https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/kivy/kivydemo-for-android.zip
- Unzip the contents and go to the folder `kivydemo-for-android`
- Copy all the subfolders here to /sdcard/kivy
- Run the Kivy launcher and select one of the Pictures, Showcase, Touchtracer, Cymunk or other demos

Unit summary

Summary	In this unit you learned how to package your Python application to run on Android devices. In the next section let's learn about how to design the Graphical User Interfaces(GUI) for mobile applications using Python libraries

References and Further Reading

- Python for Android https://python-for-android.readthedocs.io/en/latest/quickstart/
- Create a package for Android http://kivy.readthedocs.io/en/latest/guide/packaging-android.html

Unit 14: Python Graphical User Interface development

14

Unit Structure

14.1. Graphical User interface (GUI)

14.2. Different types of packages for GUI development in Python

14.3. Tkinter (GUI toolkit that comes with Python)

14.4. GUI with wxPython

Introduction

In this unit you will learn components of GUI and Tkinter standard GUI library that is bundled with python and wxPython libraries. You will also learn different types of packages available for GUI development. There are eight (8) examples to study Tkinter standard library and three (3) examples to study wxPython.

Line numbers before the programming code are included for easy references. But do not to include line numbers when you code in the text editor. Essential lines in the code are described after the program code.

Upon completion of this unit you will be able to:

Outcomes	• Describe Graphical User Interface (GUI) and its components. • Idetify different types of packages for python GUI implement. • Demonstrate the skills of programming in Tkinter GUI standard library. • Demonstrate GUI programming with wxPython cross platform libraries. • Aply event driven programming for GUI using Tkinter and wxPython libraries
Terminology	Layouts: layouts to arrange widgets import: Import the necessary modules to python code Tkinter: Standard python GUI library wxPython: Free and open source GUI package for python

14.1. Graphical User interface (GUI)

When you work with computer, you need to interact with it in many ways such as clicking on various icons, selecting from menus, right clicking, double clicking, writing text in word processor, inserting pictures, inserting movies etc. All these works are

processed in computer through hardware. User Interface is the space where interctions between you and the computer hardware.

The first user interface that came with computers, was a command-line interface where you could interact with the computer by typing commands on the keyboard. Typing commands was a very difficult task and user needs to have a very good hardware background. Later, graphical representations of such commands were created to interact with any user which are called Graphical User Interface (GUI).

Visual indications of GUI which can be implemented by python language are illustrated in fig. 14.1.

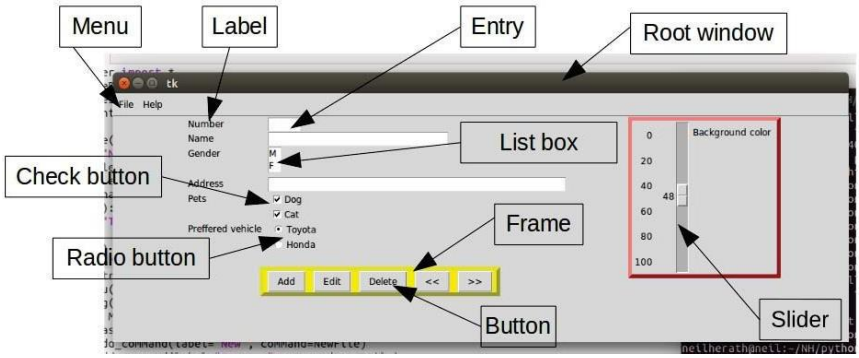


Figure14.1VisualizationofGUI

Before we proceed, let's define some of the common terms.

Window – it is a rectangular zone on computer screen

top-level window - is an independent window within an application. One application may have many top level windows. Many children windows can be created within top level window. When you close a child window it will go back to the parent window

What is shown in figure 14.1 as the root window is a top level window.

widget - the general word for visual building blocks to interact an activity in GUI.

Layouts -The way you arrange the widgets in a GUI.

Frame - Rectangular area which contains other widgets to organize different layouts.

Parent, child – A relationship of parent and child is created for any crated widget. For example, when a check button placed on a frame, frame becomes parent and check button becomes child.

14.2. Different types of packages for GUI development in Python

There are many tool kit options available for developing graphical user interface (GUI) . Most popular are given below:

Tkinter - GUI toolkit given with python is Tkinter. All the other GUI tool kits for python can be used as alternatives to the Tkinter.

PyGTK – permits to write GTK programs in python and has a lot of widegets than Tkinter.

PyQT – is a set of python software libraries for Qt Framework.Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X.

wxPython – is a set of python software libraries to create wxWidgets.

Kivy – is an open source cross platform for developing mobile apps using python. Mobile apps can be developed for andriod, iOS, linux and windows.

GUI program development with Tkinter and wxPython is described next, whereas program development with Kivy GUI is described in unit 15.

14.3. Tkinter (GUI toolkit that comes with Python)

GUI library for python given with the python programming langauage is Tkinter. GUI applications can be created easily and quickly using Tkinter. Strong object oriented widgets to the Tk GUI are given by Tkinter

Creating GUI using Tkinter

Method of programming Tkinter GUI window is described by the example 14.1.

Example 14.1 Tkinter GUI window

```
1. #/user/bin/python
2. import Tkinter as tk1
3. tp=tk1.Tk()
4. # Codes for widgets
5. tp.mainloop()
```

Window created by the above program is illustrated by Figure 14.2.

Figure 14.2. Tkinter window

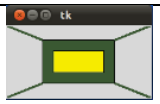
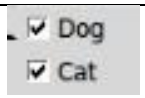
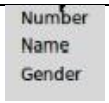
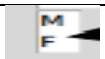
Codelineoftheaboveprogramareexplainedasfollows

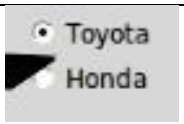
1	Selfexecutingscript,inlinuxyoucanrunby'./'ratherthan typing python <file name>.py
2	ImportTkinterlibrary
3	TkinterTkwindowisassignedto top.
4	Comment
5	Mainloopisstartedandwaitingformouseandkeyboard actoins.

Video 15: **Creating GUI for Python with**	
	This video will demostrate how to create a GUI with Tkinter. You may watch this Video carry out the tasks listed below in creating widgets, buttons etc. URL : https://youtu.be/jcgyfeZsaTc

Tkinter Widgets

Widget is a graphical user interface controls to interact between user and computer. Tkinter gives different types of controls such as buttons, labels and text boxes used in a GUI application. Tkinter widgets are described in the following table. Most widegts can be seen in figure 14.1.

Widget	Widget image	Description
Canvas		isarectangleareawhichusedto display and edit graphics.For example–lines,polygen,rectangle
Checkbutton		Number of options can be selected usingcheckbuttons.Morethanone choice can be selected in grouped options
Entry		A line of text can be entered using entry widget
Frame		A container to insert and organize other widgets.
Label		Caption of single line can be displayed using label. Images may also insert to label.
Listbox		list of options can be provided listbox
Menubutton		Menu options can be displayed in an application.
Menu		Various commands contained in Menu buttons are given to user.
Message		For displaying mulitline text fields to take values from user

Widget	Widget image	Description
Radiobutton		Only one option at a time can be selected by Radio button.
Scale		A number value can be changed by moving knob of a slider.
Scrollbar		Scrolling facility can be added for various widegets.
Text		Multiple line of text can be displayed.
Toplevel		Separate window container can be provided.
LabelFrame		Works as a container for complex window layouts.
tkMessageBox		Message box can be inserted to application.

Using Button widget in window

Function or a method can be attached to a button which is called automatically when the button is clicked.

Here is the simple syntax to create this widget:

w = Button (master, option=value, ...) Parameters:

- master: Parent window.
- options: Options are listed in the Following table.

Option	Description
Active background	Backgroundcolorwhenthecursorisonthebutton.
Active foreground	Foregroundcolorwhenthecursorisonthebutton.
Bd	widthoftheborder.Defaultis2.
Bg	Normalbackgroundcolor.
command	Callingfunctionormethodwhenthebuttonisclicked.
Fg	Colourofthetext(foreground)
Font	Typeofthefontusedforlabelofthe button.
Height	Heightofthetextor image
Highlight color	Thecolorofthefocushighlightwhenthewidgethasfocus.
Image	Imagetobeattachedanddisplay

Option	Description
Justify	Alignmentofthetextline-justify
CENTER	Alignmentofthetextline-center
Padx	Horizontallengthofbuttonthroughxaxiscanbesat.
Pady	verticallengthofbuttonthroughxaxiscanbesat.
Relief	Borderstyle-buttonappearasSUNKEN,RAISED, GROOVE, or RIDGE
State	CansetthebuttonasDISABLEDorACTIVE.Itactivewhen the button is over.
underline	Underlinetherelevantcharacter,Defaultis-1,whichmeans no character is underlined.
Width	Setthewidthofthebutton.
wraplength	Ifthisvalueissettoapositivenumber,thetextlineswillbe wrapped to fit within this length.

Example 14.2. This python program illustrates how to work with button.

```
1  #!/usr/bin/python
2
3  importTkinterastk
4  importtkMessageBoxasbx
5  tp=tk.Tk()
6  defexbutton():
7   bx.showinfo("Pythonbuttonexample","HelloWorld")
8  B=tk.Button(tp,text="Pressthisbutton",command=exbutton)
9  B.pack()
10 tp.mainloop()tp.mainloop()
```

Figure14.3.Message“HelloWorld”willappearwhen"Prssthisbutton" is clicked

The output of the above program is shown in fig 14.3. When the button called ‘Press this button’ is clicked, the message box ‘Hello World’ will appear.

Description of codes in line numbers of the program

9. import tkMessageBox module

10. a function called exbutton

11. Text of the button is “Press this button” and button is assigned to “B”. Function exbutton is invoked by clicking button.

Using Entry widget in window

Single line text strings can be entered using entry widget. Text widget can be used to enter multiple lines and label widget is used to display one or more lines of text but cannot edit.

Syntax

Here is the simple syntax to create this widget −

```
w = Entry( master, option, ... )
```

Parameters:

- master: the parent window.
- options: : List of ususally used options are given below

Option	Description
Bg	backgroundcolor
Bd	Sizeoftheborder Thesizeoftheborderaroundtheindicator.Defaultis2 pixels.
command	Invokingfunctionwhenuserchangecontent
cursor	Cursorofthemousewillchangeaccordingtothegivencursor name (arrow, dot etc).
font	Type of the font
exportsel ection	By default, if you select text within an Entry widget, it is automatically exported to the
clipboard	To avoid this exportation, use exportselection=0.
Fg	The color used to render the text.

Option	Description
highlight color	The color of the focus highlight when the checkbutton has the focus.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles .
Selectbac kground	Background color of the selected text
selectbor derwidth	The width of the border to use around selected text. The default is one pixel.
selectfore ground	The foreground (text) color of selected text.
show	Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
Text variable	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
Width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
Xscroll command	If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

Example 14.3. This program describes how to use a Tkinter entry in a program.

```
1.  from Tkinter import *
2.  tp=Tk()
3.  la1=Label(tp,text="UserID")
4.  la1.pack(side=LEFT)
5.  en1=Entry(tp,bd=6)
6.  en1.pack(side=RIGHT)
7.  tp.mainloop()
```

When the above code is executed, it produces the output as show in the Figure 14.4:

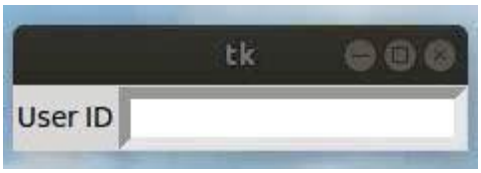


Figure14.4Textentry

Using Label widget in window

This widget implements a display box where you can place text or images.
The text displayed by this widget can be updated at any time you want.

Syntax

```
w = Label ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
anchor	Thisoptionscontrolswherethetextispositionedifthewidgethas more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
Bg	Thenormalbackgroundcolordisplayedbehindthelabeland indicator.
Bitmap	Setthisoptionequaltoabitmaporimageobjectandthelabelwill display that graphic.
Bd	Thesizeoftheborderaroundtheindicator.Defaultis2 pixels.
Cursor	Ifyousetthisoptiontoacursorname(arrow,dotetc.),themouse cursor willchange to thatpattern when itis overthecheckbutton.
Font	If youaredisplayingtextinthislabel(withthetextortextvariable option, the font option specifies in what font that text will be displayed.
Fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
Height	The vertical dimension of the new frame.

Option	Description
Image	To display a static image in the label widget, set this option to an image object.
Justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
Padx	Extra space added to the left and right of the text within the widget. Default is 1. pady Extra space added above and below the text within the widget. Default is 1.
Relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values. text To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
Textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
Under Line	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.

Example 14.4. This program illustrates how to program using label and grid.

```
1  import Tkinter as tk
2  m=tk.Tk()
3  tk.Label(m,text='First',bg="red").grid(row=0,column=0)
   tk.Label(m,text='Second',bg="blue").grid(row=0,column=1)
4  tk.Label(m,text='Third',bg="green").grid(row=1,column=0)
5  m.mainloop()
```

The output of the above program is illustrated in the figure 14.5. Remember you should write line 5 in one line.

Figure 14.5 output of the above program

Setting rows and columns of grid are shown in following table. Any widget like entry, image can be arranged in grids.

Row=0,coloumn=0	Row=0,coloumn=1
Row=1,coloumn=0	Row=1,coloumn=1

Using Checkbutton widget

Checkbutton for selection

Output of running the code in example 14.5 is shown in the Figure 14.6

Figure14.6Checkbuttonentry

Example14.5Programtodescribeshowtoprogramusinglabeland checkbutton in Tkinter GUI.

```
1  from Tkinter import * #import Tkinter module
2  m=Tk() # Tk window assigned to m
3  Label(m,text="Your sex: ") .grid(row=0)
4  Checkbutton(m,text="male").grid(row=1)
5  Checkbutton(m,text="female").grid(row=2)
6  mainloop() #starts main loop- waiting for mouse
   and #key
   board
```

Line number 3 of the above code is a Label and “Your sex:” is displayed. There are two checkbuttons one for “male” and other for “female”. Mainloop() waits for mouse and keyboard.

Grid property set label and check buttons 3 top rows of the “m” window.

Checkbutton for selection with align to the west

Checkbutton can be aligned using stick parameter in grid object. The program to explain the grid sticky option of checkbutton is given in example 14.6 and the output screen is shown in figure 14.7.

Example 14.6 This program explains how to program using checkbutton with grid sticky option

```
1 from Tkinter import * #import Tkinter module m=Tk() # Tk
2 window assigned to m Label(m,text="Your sex: ")
3 .grid(row=0,sticky=W)
4 Checkbutton(m,text="male",variable=var1).grid(row=1,sticky=
5 W)
6 Checkbutton(m,text="female",variable=var2).grid(row=2,stick
  y=W)
  mainloop() #starts main loop- waiting for mouse and key
  board
```

Figure14.7Leftalignmentsofcheckboxes

Male and female checkbuttons of figure 14.7 (a) are not aligned and figure 11.7 (b) shows the aligned checkbuttons. This can be done with sticky option of grid manager. The above example has sticky=W i.e. aligned to left side. Likewise, following options also can be used.

N, E, S, W, NE, NW, SE, and SW

Assigning checkbutton selection to variable

Example 14.7 This program describes how selections of checkbuttons are assigned to variables.

```
1 from Tkinter import * #import Tkinter module
2 m=Tk() # Tk window assigned to m
3 def var_states():
4 print var1.get()
5 print var2.get()
6 Label(m,text="Your sex: ") .grid(row=0,sticky=W)
  var1=IntVar()
  Checkbutton(m,text="male",variable=var1).grid(row=1,sti
  cky=W)
```

```
var2=IntVar()
Checkbutton(m,text="female",variable=var2).grid(row=2,s
ticky=W)
Button(m, text='Show',
command=var_states).grid(row=4,
sticky=W, pady=4)
mainloop() #starts main loop- waiting for mouse and key
board
```

Command of the button is binded to the var_states function. Variable values in three different marks of checkbuttons are displayed in figure 14.8.

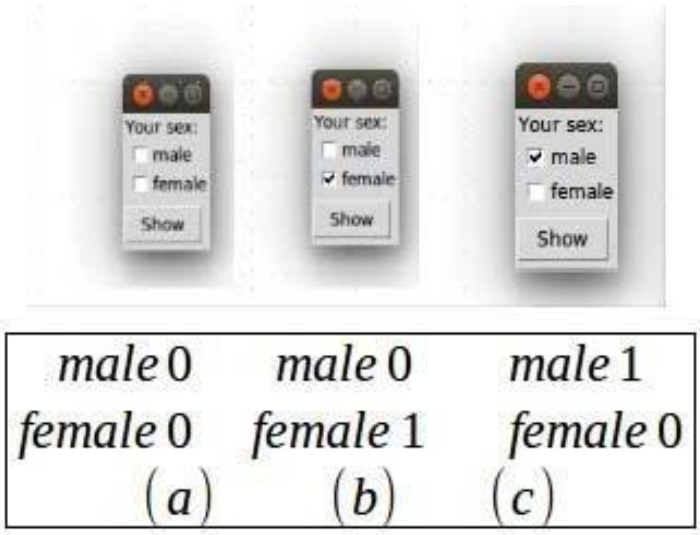


Figure 14.8 Three different values in checkbutton

Listbox

Listbox is the Tkinter standard list box to facilitate a list. List of provinces in Sri Lanka created by using tkinter GUI library is shown in the figure

14.9. User can select a province from the list of provinces.

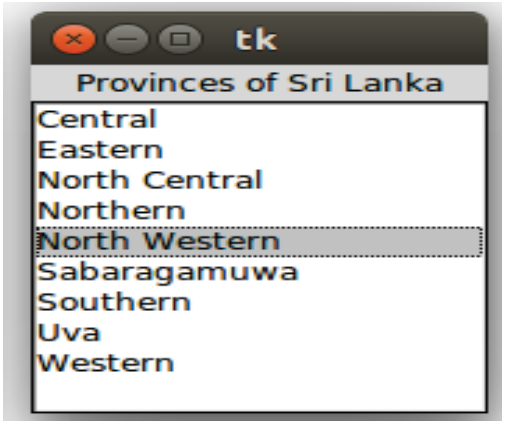


Figure 14.9.list of provinces, user can select either province from the list

Python code for the output of the Figure 14.9 is given in example 14.8. Example 14.8 program to describe listbox.

```
from Tkinter import *
master = Tk()
l1=Label(master,text="Provinces of Sri Lanka")
l1.pack()
listbox = Listbox(master)
listbox.pack() listbox.insert(END, "Central")
for item in ["Eastern",
             "North Central",
             "Northern",
             "North Western",
             "Sabaragamuwa",
             "Southern",
             "Uva",
             "Western"]:
    listbox.insert(END, item)
mainloop()
```

Line 3: l1 instance is derived from the Label class. Text of the label “l1” is "Provinces of Sri Lanka".

```
l1=Label(master,text="Provinces of Sri Lanka")
```

Line 5: listbox instance is derived from the Listbox class.

```
listbox = Listbox(master)
```

Line 7: Append an item to the list.

```
listbox.insert(END, "Central")
```

Line 8 to 16: Append all other items to list.

```
for item in ["Eastern",
                "North Central",
                "Northern",
                "North Western",
                "Sabaragamuwa",
                "Southern",
                "Uva",
                "Western"]:
        listbox.insert(END, item)
mainloop()
```

14.4. GUI with wxPython

WxPython is a free and open source cross platform GUI toolkit package. It can be downloaded from the official website http://wxpython.org. It consists of wxObject class, which is the base for all classes in the API. Control module contains all the widgets used in GUI application development. For example, wx.Button, wx.StaticText (analogous to a label), wx.TextCtrl (editable text control), etc.

“wxPyton” contains five basic modules, windows, Graphics Device Interface (GDI), Misc, Core and Controls. Basic modules and some of sub modules are depicted in figure 14.10.

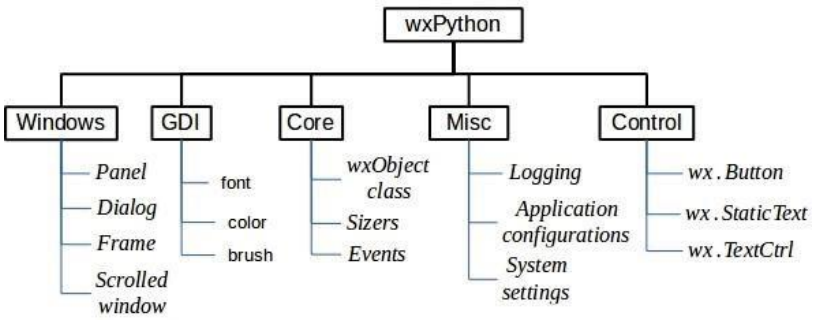


Figure14.10wxPythonbasicadsubmodules

BasicwxPythonGUIdevelopment

GUIinFigure14.10hasaframeanda“Helloworld”text.

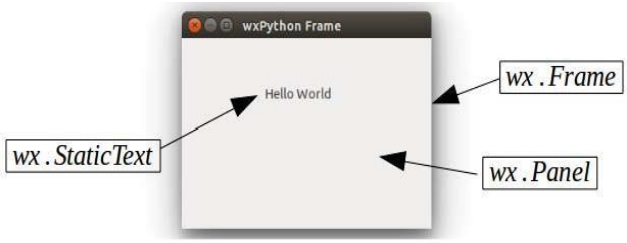


Figure 14.11.basic wxPython program output

OutputinFigure14.11isproducedbythecodeinexample14.9.

Example14.9ProgramtoexaplaintocreatewxPythonGUIwindow

```
1  importwx
2  app=wx.App()
3  window=wx.Frame(None,title="wxPythonFrame",size=(300,200))
4  panel=wx.Panel(window)
5  label=wx.StaticText(panel,label="HelloWorld",pos=(100,50))
6  window.Show(True)
7  app.MainLoop()
```

Letusanaysetheabovecodetounderstandwhatitdoes. Line 1:

Import the wx module.

```
importwx
```

Line2:DefineanobjectofApplicationclass.

```
app= wx.App()
```

Line3:Createatoplevelwindowasobjectofwx.Frameclass.Captionand size parameters are given in constructor.

```
window=wx.Frame(None,title="wxPythonFrame",size=(300,200))
```

Line4:AlthoughothercontrolscanbeaddedinFrameobject,theirlayout cannot be managed. Hence, put a Panel object into the Frame.

```
panel=wx.Panel(window)
```

Line5:AddaStaticTextobjecttodisplay‘HelloWorld’atadesired position inside the window.

```
label=wx.StaticText(panel,label="HelloWorld",pos=(100,50))
```

Line6:Activatetheframewindowbyshow()method.

```
window.Show(True)
```

Line7:EnterthemaineventloopofApplicationobject.

```
app.MainLoop()
```

Line 3,4,5 of the above program include wx classes calledwx.Frame, wx.Panel andwx.StaticText.

Top level window classes (wx.Frame and wx.Panel) wx.Frame

wx.Frame class is a top level window class. Higher level to this class is wx.Window. You can change its size and position. It is a container widget. It means that it can contain any window that is not a frame or dialog. wx.Frame has a title bar, borders and a central container area. The title bar and borders are optional. They can be removed by various flags.

wx.Frame Class has a default constructor with no arguments. It also has an overloaded constructor with the following parameters:

wx.Frame (parent, id, title, pos, size, style, name)

```
window=wx.Frame(None,-1,“Hello”,pos=(10,10),size=(300,200),style=
```

```
wxDEFAULT_FRAME_STYLE,name="frame")
```

wx.Panel

wx.Panel class is derived from the wx.Window class. This class is inserted to the wx.Frame class. Widgets such as button, text box etc. can be placed in this class.

Sub classing the Frame -"Hello World" program with class

The GUI output in Figure 14.12 has a frame with the title "Hello World".

Figure 14.12.basic GUI of wxPython

Output of the Figure 14.12 is produced by running the program in example 14.10.

Example 14.10 Program to create frame class called "Frame".

```
1  importwx
2  classFrame(wx.Frame):
3      definit(self,title):
4          wx.Frame.init(self,None,title=title,size=(350,200))
5
6  app=wx.App(redirect=True)
7  top=Frame("HelloWorld")
8  top.Show()
9  app.MainLoop()
```

Let us analyse code lines of the example 14.10 to understand creation of frame class.

Line 2: class Frame is derived from the wx.Frame.

```
classFrame(wx.Frame):
```

Line7:classFrameiscalledwiththetitle“HelloWorld”

```
top=Frame("HelloWorld")
```

Line3:TheabovetitleisreceivedtothefollowingclassFrameandinitiated with the title.

```
definit(self,title):
```

Line 4: title of the line 3 is received to the wx.Frame and it initiated with this title.

```
wx.Frame.init(self,None,title=title,size=(350,200))
```

Video 16: Python GUI with WxFrame	
	This video will demostrate how to create a GUI with WxFramework. You may watch this Video to create the calculator example. URL : https://youtu.be/je7l6OMdjFU

Adding content to the frame with Calculator example

Calculator GUI in Figure 14.13 has 3 text boxes for inputting 2 numbers and for displaying result. There are three (3) buttons for calculating addition, subtraction and multiplication.

Figure 14.13 Calculation of 2 variables

Output of the Figure 14.13 is produced by running the code of the example 14.11.

Example 14.11 program code describes how to use textcontrols and buttons of wxPython

```
1  importwx
2  classwxcal(wx.Frame):
3          definit(self,title):
4                  wx.Frame.init(self,None,title=title,
   pos=(150,150),size=(350,200))
5                  panel=wx.Panel(self)
6                  self.fno=wx.TextCtrl(panel,value="",pos=(10,2),
   size=(50,-1))
7                  self.sno=wx.TextCtrl(panel,value="",pos=(70,2),
   size=(50,-1))
8                          self.result=wx.TextCtrl(panel,-1,"",pos=(130,2),
   size=(50,-1))
9                  self.addbtn=wx.Button(panel,wx.ID_NONE,"+",pos=
   (10,50),size=(50,-1))
10                 self.addbtn.Bind(wx.EVT_BUTTON,self.addbtnclick)
11                 self.subbtn=wx.Button(panel,wx.ID_NONE,"-",pos=
   (70,50),size=(50,-1))
12                 self.subbtn.Bind(wx.EVT_BUTTON,self.subbtnclick)
13                 self.mulbtn=wx.Button(panel,wx.ID_NONE,"*",pos=
   (130,50),size=(50,-1))
14                 self.mulbtn.Bind(wx.EVT_BUTTON,self.mulbtnclick)
15         defaddbtnclick(self,event):
16                         self.result.SetValue(str(float(self.fno.GetValue())+
   float(self.sno.GetValue()))
17         defsubbtnclick(self,event):
18                         self.result.SetValue(str(float(self.fno.GetValue())-
   float(self.sno.GetValue())))
19         defmulbtnclick(self,event):
20                         self.result.SetValue(str(float(self.fno.GetValue())*
   float(self.sno.GetValue())))
21 app=wx.App(redirect=True)
22 top=wxcal("WXCalulator")
23 top.Show()
24 app.MainLoop()
```

Line2:Classwxcaliscreatedandtsbaseclassiswx.Frame.

```
classwxcal(wx.Frame):
```

Line5:“panel”isatype of“wx.Panel”class.Allthewidgetssuchastext, buttons are placed in panel.

```
panel= wx.Panel(self)
```

Line 6 to 8: Two (2) text boxes (wx. TextCtrl) named “fno” and “sno” are placed in panel and their positions and sizes are specified as follows. There is another text box named “result” is also placed in specified position and size.

```
self.fno=wx.TextCtrl(panel,value="",pos=(10,2),size=(50,-1))
self.sno=wx.TextCtrl(panel,value="",pos=(70,2),size=(50,-1))
self.result=wx.TextCtrl(panel,-1,"",pos=(130,2),size=(50,-1))
```

Line 10 to 11: Button for addition named “addbtn” is a wx.Button class type. Text of the button is “+” and position and size is as follows. This button has no identification (wx.ID_NONE). This button is bind to the method “addbtnclick” on the click event of the button (wx.EVT_BUTTON).

```
self.addbtn=wx.Button(panel,wx.ID_NONE,"+",pos=(10,50),size=(50,-1))
self.addbtn.Bind(wx.EVT_BUTTON,self.addbtnclick)
```

Line 13 to 17: Program codes for buttons for subtraction and multiplication and their bindings.

```
self.subbtn=wx.Button(panel,wx.ID_NONE,"-",pos=(70,50),size=(50,-1))
self.subbtn.Bind(wx.EVT_BUTTON,self.subbtnclick)

self.mulbtn=wx.Button(panel,wx.ID_NONE,"*",pos=(130,50),size=(50,-1))
self.mulbtn.Bind(wx.EVT_BUTTON,self.mulbtnclick)
```

Line 19 to 26: Methods to add, subtract and multiply values of fno and sno and assigned to result are coded here.

```
def addbtnclick(self,event):
        self.result.SetValue(str(float(self.fno.GetValue())+float(self.sno.GetValue())))

defsubbtnclick(self,event):
        self.result.SetValue(str(float(self.fno.GetValue())-float(self.sno.GetValue())))

def mulbtnclick(self,event):
        self.result.SetValue(str(float(self.fno.GetValue())*float(self.sno.GetValue())))
```

WxPython Graphics Drawing Interface (GDI)

GDI in wxPython offers objects required for drawing shape, text and image like Colour, Pen, Brush and Font. It is used to interact with graphic devices such as monitor, printer or a file. It consists of 2D vector graphics, fonts and images. Objects called “device context” (DC) should be created to start drawing graphics. It represents number of devices in a generic way.

wx.DC classes are

wxBufferedDC

wxBufferedDC

wxBufferedPaintDC

wxPostScriptDC

wxMemoryDC

wxPrinterDC

wxScreenDC

wxClientDC

wxPaintDC

wxWindowDC

Drawing a circle with background colour of red in a window titled “Draw circle” is shown in Figure 14.14.

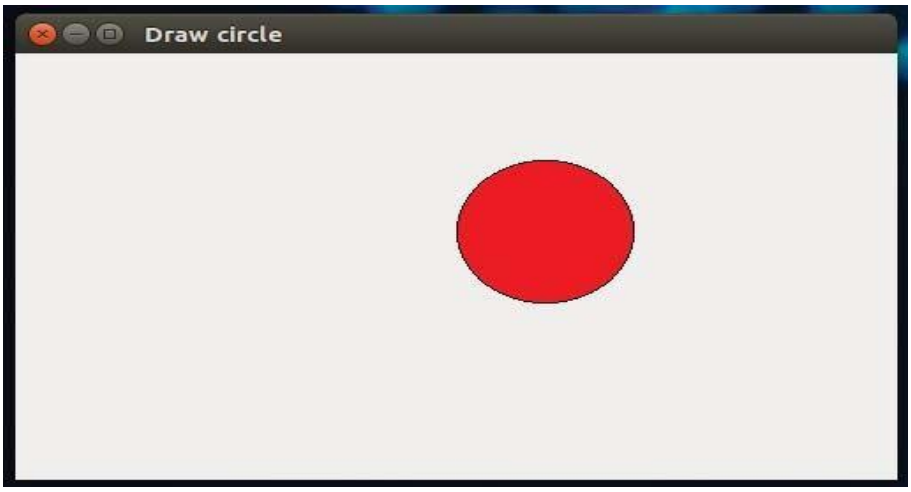


Figure14.14DrawingcircleusingGDI

Example14.12Programtodisplaywindowandcirclewithredbackground	
1	importwx
2	classdrawgeo(wx.Frame):
3	definit(self,parent,title):
4	super(drawgeo,self).init(parent, title=title,size=(500,300))
5	self.InitUI()
6	defInitUI(self):
7	self.Bind(wx.EVT_PAINT,self.drawcle)
8	defdrawcle(self,w):
9	dc=wx.PaintDC(self)
10	color=wx.Colour(255,0,0)
11	b=wx.Brush(color)
12	dc.SetBrush(b)
13	dc.DrawCircle(300,125,50)
14	app1=wx.App()
15	top=drawgeo(None,"Drawcircle")
16	top.Show()
17	app1.MainLoop()

The above window and circle with red background can be programmed by using the following program. In this program, initially, create instance of wx app and activate a wx.frame custom class to draw the circle.

Now we will study program code lines in the above program. For easy reference, program is numbered at the left hand side column to the program.

1	importwx
	ImportwxPythonmodulesintotheprogram

16	app1 = wx.App()
	Createaninstanceofwxappcalled app1

17	top=drawgeo(None,"Drawcircle")
	Createaninstanceofthecustomclassofwx.Framenamed “drawgeo” with the title “Draw circle".

18	top.Show()

	Showtheclass“top”.

2	class drawgeo(wx.Frame):
	Createawx.Frameclassnamed“drawgeo”

3	definit(self,parent,title):
	Definetheinitmethodwhoseparametersaretitleandparameters belongs to supper class “wx.Frame”.

4	super(drawgeo,self).init(parent,title=title,size=(500,300))
	Titleofthesuperclassis"Drawcircle"andsizeis (500,300)

5	self.InitUI()

StartInitUImethod.

7	def InitUI(self):
	DefineInitUI()method

8	self.Bind(wx.EVT_PAINT,self.drawcle)
	“drawcle”methodoftheclassdrawgeoisboundtothe wx.EVT_PAINT event.

10	def drawcle(self,w):
	Definethemethod“drawcle()”

11	dc= wx.PaintDC(self)
	wx.PaintDC(self)isusedtodrawgraphicsinaclient area.

12	color=wx.Colour(255,0,0)

	Createaninstanceofwx.Colourcalled“color”whosecolouris (255,0,0). i.e. red.

13	b=wx.Brush(color)
	Createaninstanceofwx.Brushcalled“b”whosecolouris“color”. wx.Brush is used to fill background colur of the object.

14	dc.SetBrush(b)
	Setthebrushofthewx.PaintDCclasscalled“dc”.

15	dc.DrawCircle(300,125,50)
	Drawthecirclewithxaxispoint,yaxispointandradius

wxPython GUI visual Designer tools

Coding GUI windows is a very difficult and time cosuming effort. Therefore, there are few visual GUI designer tools available for rapid and quick GUI application development. They are wxFormBuilder, wxDesigner, wxGlade, BoaConstructor, gui2py.

Activity

Activity 14.1

1. Write a GUI program using Tkinter libraries to input two boolean values (0 or 1) and select a boolean operation from set of boolean operations AND, OR, NAND, NOR, XOR. Result should be printed in a text box next to the boolean selection box.
2. Write a GUI program using wxPython libraries to input 2 boolean values (0 or 1) and select a boolean operation from set of boolean operations AND, OR, NAND, NOR, XOR. Result should be printed in a text box next to the boolean selection box.

.

Unit summary

Summary	In this unit you learned to identify elements of GUI, different types of python related packages for GUI development, GUI implement using Tkinter standard library and wxPython. Then you learned how to invoke widgets of GUI with methods in user defined classes.z

References and Further Reading

1. kivy.org–officialsite
2. TutorialsPoint(I)Pvt.Ltd., PYTHONprogramminglanguage, https://www.tutorialspoint.com/python/python_tutorial.pdf

Unit 15: GUI programming using kivy libraries

15

Unit Structure

15.1 Basic GUI programming (user password GUI)

15.2 Kivy Layouts and Widgets

15.3 Kv language

15.4 Developing a Calculator using python and kivy

15.5 Develop a Calculator using python and kv language

15.6 GUI with check boxes

Introduction

Developing GUI using kivy is different from Tkinter and wxPython because Kivy has a different architecture. You will learn how Kivy GUI libraries can be used in python programming by studying the given programming codes. Output of each programming code is given before the code snippet. Write and run codes and check whether the outputs are correct.

Python with kivy cross platform is a very useful method of implementing GUI for mobile application. Four different types of GUI programming examples are given in this unit to study different areas when you want to run a python GUI with Kivy libraries using widgets, layouts, running with only python or combined with python and Kv language.

Upon completion of this unit you will be able to:

Outcomes	• List type of layouts and widgets in Kivy libraries. • Describe how to program by arranging and inserting widgets into grid layout. • Describe calling methods when interacting with widgets. • Describe programming with combining python and Kv language

15.1. Basic GUI programming (user password GUI)

A sample user password entry GUI is shown in Figure 15.1 and it includes two labels and two text entry boxes in a two column gridlayout. Only the GUI shown in figure 15.1 is displayed and no other functions are executed when running the programming code.

Figure15.1Userpasswardentryboxusing kivy

Example15.1Whenyouruntheprogramgivenbelow,GUIofthefigure 15.1 canbe obtained.

```
1  fromkivy.appimportApp
2  fromkivy.uix.gridlayoutimportGridLayout
3  fromkivy.uix.labelimportLabel
4  fromkivy.uix.textinputimportTextInput
5  classLoginScreen(GridLayout):
6          definit(self,**kwargs):
7          super(LoginScreen,self).init(**kwargs)
8          self.cols=2
9          self.add_widget(Label(text='UserName'))
10         self.username=TextInput(multiline=False)
11         self.add_widget(self.username)
12         self.add_widget(Label(text='password'))
13         self.password=TextInput(password=True,multiline=False)
14         self.add_widget(self.password)
15 classMyApp(App):
16         defbuild(self):
17                 returnLoginScreen()
18                         ifname=='main':
19 MyApp().run()
```

Now let us analyse the above program code. Line numbers are added for easy referance.

Line 1 to 4: importing the necessary Kivy GUI libraries (App, gridlayout, label and textinput).

```
fromkivy.appimportApp

fromkivy.uix.gridlayoutimportGridLayout
from kivy.uix.label import Label
```

Line6:inbuiltclass“GridLayout”hasusedforthebaseclassfortheclass “LoginScreen”.

```
class LoginScreen(GridLayout):
```

Line9:definethenumberofcolumnsinthegridlayout.Hereitis2.

```
self.cols=2
```

Line 10 to 15: Add widgets to the LoginScreeen gridlayout. First displayed the 'User name' label, then, text input to input user name. These 2 widgets are inserted to first row of the two-column gridlayout. Then 'password' label and text input are inserted into 2nd row of the gridlayout.

```
self.add_widget(Label(text='User Name'))
self.username=TextInput(multiline=False)
self.add_widget(self.username)
self.add_widget(Label(text='password'))

self.password=TextInput(password=True,multiline=False)
self.add_widget(self.password)
```

In kivy, GUI can be implemented with two types of reusable user interfaces, layouts and widgets.

15.2. Kivy Layouts and Widgets

Layouts

There are 5 types of layouts in Kivy.

1. **GridLayout:** used to arrange widgets in a grid. Even if one dimension of the grid is given, Kivy can compute the size of the elements and arrange them.

2. **StackLayout:** used to arrange widgets adjacent to each other. It uses a set size in one dimension and does not try to make them fit within the entire space. This is useful to display widgets of the same size.

3. **AnchorLayout:** this layout considers only about children positions. It allows placing the children at a position relative to a border of the layout. Does not consider size_hint.

4. **FloatLayout:** facilitate placing children with arbitrary locations and size, either absolute or relative to the layout size. Default size_hint (1, 1) will make every child the same size as the whole layout, therefore this value should be changed if there are more than one child. To use absolute size, we can set size_hint to (None, None). This widget considers pos_hint, as a dict setting position relative to layout position.

5. **RelativeLayout**: Behaves similar to FloatLayout, except that children positions are relative to layout position, not to the screen.

Widgets

Widgets are pieces of code (small programs) of user interface elements that provide different functions. Few examples of widgets are: file browser, buttons, sliders, and lists. Widgets may not be visible all the time and they receive MotionEvents.

15.3. Kv language

Mobile GUI applications with kivy libraries can be coded in two different methods. In first method, functions, classes, widgets and their properties are coded in the same python file. This method is described with examples in section 15.1 and 15.4.

In the second method, functions and classes of application are coded in a python file. Extension of this file is .py. Widgets and their properties are coded in a kv file. Extension of this file is .kv. Therefore, logic of the application and its user interface can be separated clearly. GUI applications can be changed easily to suit user's requirements by doing this way. This method is described with examples in sections 15.5 and 15.6.

15.4. Developing a Calculator using python and kivy

In this calculator program, three text boxes are available for entering two numbers and other is used to print result. There are three buttons are available for addition (+), subtracting (-) and multiplication (*). This calculator graphical user interface is shown in figure 2. When you press “+” button, 2 numbers will add result will be displayed in the third text box. Subtraction or multiplication will happen by pressing “-” and “*” buttons.

Figure 15.2 Calculation of 2 variables

Calculator in Figure 15.2 can be obtained by running the program code in example 15.2.

```
1   fromkivy.appimportApp
2   fromkivy.uix.buttonimportButton
3   fromkivy.uix.buttonimportLabel
4   fromkivy.uix.gridlayoutimportGridLayout
5   fromkivy.uix.textinputimportTextInput
6   fromkivy.uix.widgetimportWidget
7
8   classrootwd(GridLayout):
9           definit(self,**kwargs):
10                  super().init(**kwargs)
11                  self.cols=3
12                          self.add_widget(Label(text='FirstNumber'))
13                  self.add_widget(Label(text='SecondNumber'))
14                  self.add_widget(Label(text='Result'))
15                  #
16                  self.fno=TextInput(multiline=False,font_size=30)
17                  self.add_widget(self.fno)
18                  #
19                  self.sno=TextInput(multiline=False,font_size=30)
20                  self.add_widget(self.sno)
21                  #
22                  self.tno=TextInput(multiline=False,font_size=30)
23                  self.add_widget(self.tno)
24                  self.display=self.tno
25                  #
26                  btnadd=Button(text="+",font_size=30)
27                  self.add_widget(btnadd)
28                  btnadd.bind(on_press=self.btnaddcal)
29
30                  btnsub=Button(text="-",font_size=30)
31                  self.add_widget(btnsub)
32                  btnsub.bind(on_press=self.btnsubcal)
33
34                  btnmul=Button(text="*",font_size=30)
35                  self.add_widget(btnmul)
36                  btnmul.bind(on_press=self.btnmulcal)
37
38          defbtnaddcal(self,instance):
39
40                                  self.tno.text=str(float(self.fno.text)+float(self.sno.text)
    )
41          defbtnsubcal(self,instance):
42                          self.tno.text=str(float(self.fno.text)-
    float(self.sno.text))
43
44          defbtnmulcal(self,instance):
45
46                                  self.tno.text=str(float(self.fno.text)*float(self.sno.text)
    )
47
48  classtestapp(App):
49                  defbuild(self):
50                          returnrootwd()

51  ifname=='main':
52          testapp().run()
```

Example 15.2

Let's analyze the program given in example 15.2.

Line 8: Class rootwd belongs to the class GridLayout

Line 11: Gridlayout rootwd has 3 columns.

```
self.cols=3
```

Line 12 to 14: Add labels first number, second number and result to the first row of the rootwd grid layout.

```
self.add_widget(Label(text='First
Number'))
self.add_widget(Label(text='SecondNumb
er')) self.add_widget(Label(text='Result'))
```

Line 16 to 24: Add 3 text inputs named fno, sno and tno to the rootwd grid layout. Font size of each text input is set to 30 and muti line of text inputs are set to false. Line 17 is used to add text input to the rootwd grid layout.

```
self.fno=TextInput(multiline=False,font_si
ze=30) self.add_widget(self.fno)

#

self.sno=TextInput(multiline=False,font_si
ze=30) self.add_widget(self.sno)

#
```

Line 26 to 28: Add button named btnadd to the grid layout and its text is “+” and font size is set to 30. Code in the line 28 bind the on_press (when press on the button) to the method btnaddcal.

```
btnadd=Button(text="+",font_size=30)
self.add_widget(btnadd)
btnadd.bind(on_press=self.btnaddcal)
```

Line 38 to 39: Method btnaddcal, calculates the addition of two input texts “fno” and “sno” then result is assigned to the “tno”.

```
def btnaddcal(self,instance):
    self.tno.text=str(float(self.fno.text)+float(self.sno.text))
```

Lines 41 to 48 are same type of methods used to subtract and multiply two numbers.

15.5. Develop a Calculator using python and kv language

Output of the calculator example which is programed using both Python and Kv language is shown in Figure 15.3.

Figure 15.3 Calculator using kv language

Two programming code files are used for this method. Name of the first file example 15.3(a) is twonum.py and name of the second file example 15.3(b) is twonum.kv. Now consider the Python program code in twonum.py

Example 15.3 (a) twonum.py program code

```
1  importkivy
2  fromkivy.appimportApp
3  fromkivy.uix.gridlayoutimportGridLayout
4
5  classtwonumgrid(GridLayout):
6          defcal(self,fnum,snum,op):
7                  ifop=="+":
8                                  self.display.text=str(float(fnum)+float(snum))
9                  elifop=="-":
10                                  self.display.text=str(float(fnum)-float(snum))
11                 elifop=="*":
12                                  self.display.text=str(float(fnum)*float(snum))
13
14 classtwonumapp(App):
15
16         defbuild(self):
17                 returntwonumgrid()
18
19 ifname=='main':
20         twonumapp().run()
```

Line5:Class“twonumgrid”isa“GridLayout” class.

```
class twonumgrid(GridLayout):
```

Line 6 to 12: method named “cal” to calculate addition, subtraction or multiplication according to the “op” input parameter. Input parameters to the method are fnum, snum and op (op is either + or – or *). Variable values fnum, snum and op are received from the Kv file.

```
    defcal(self,fnum,snum,op):
        if op=="+":
            self.display.text=str(float(fnum)+float(snum)) elif op=="-":
            self.display.text=str(float(fnum)-float(snum)) elif op=="*":
            self.display.text=str(float(fnum)*float(snum))
```

Now consider the Kv language file, twonum.kv file.

Example 15.3 (b) twonum.kv program code

```
1  <CusButton@Button>:
2          font_size:40
3
4  <CusText@TextInput>:
5          font_size:35
6          multiline:False
7
8  <twonumgrid>:
9          id:twonumcal
10         display:result
11         rows:2
12         padding:10
13         spacing:10
14
15         BoxLayout:
16                 CusText:
17                         id: fno
18                 CusText:
19                         id: sno
20                 Label:
21                         text:'='
22                         font_size:40
23                 CusText:
24                         id:result
25
26         BoxLayout:
27                 CusButton:
28                     text:"+"
29                                                 on_press:twonumcal.cal(fno.text,sno.text,self.text)
30                 CusButton:
31                     text:"-"
32                                                 on_press:twonumcal.cal(fno.text,sno.text,self.text)
33
34                 CusButton:
35                     text:"*"
```

```
    on_press:twonumcal.cal(fno.text,sno.text,self.tex
```

Line 1 to 6: Button and TextInput widgets can be customized to suit our requirements and can be assigned to customize widgets. In this example, these two widgets are namd as CusButton and CusText . This assignment is coded by using “@” sign. In this example, it coded as CusButton@Button and CusText@TextInput. An instance of a root widget can be declared within <> and followed by:. For example

<CusButton@Button>: and <twonumgrid>: in line 8.

```
        font_size:40
<CusText@TextInput>:
        font_size: 35 multiline:False
```

Line 8 to 13: class “twonumgrid” is an instance of class GridLayout. In a widget tree, which display information in a hierarchical structure like a tree, it is often needed to access or reference other widgets. The kv Language facilitates this access using ids. Id of the class twonumgrid is declared as “twonumcal”. Result of the two text inputs are displayed in the text input named “result”. Value of the result textinput is transferred to the class twonumgrid by declaring to the display of the class. Number of rows in gridlayout is 2. Length of padding of widgets is declared as 10.

The padding argument tells Kivy how much space there should be between the Layout and its children, whereas the spacing arguments tell it how much spacing there should be between the children.

```
<twonumgrid>:
id: twonumcal display: result rows: 2
padding: 10
spacing: 10
```

Line 15 to 24: Boxlayout for the first row of the gridlayout is declared in these lines. Under the BoxLayout, there are two CusText text inputs which are identified by fno and sno. Another label is included for “=” and another CusText text input is included for displaying result and identified by “result”.

```
        BoxLayout:
                CusText:
                        id:fno CusText:
                        id:sno
                Label:
                        text: '=' font_size:40

                CusText:
                        id: result
```

Line 26 to 35: Second row of the gridlayout has another BoxLayout and it has three (3) CusButtons. Text of each CusButton is +, - and *. Method “cal” of the class “twonumgrid” with its paramters in the twonum.py can be invoked by the on_press event. Input parameters are fno.text, sno.text and text of the Relevant CusButton.

```
BoxLayout:
CusButton:
text: "+"
on_press: twonumcal.cal(fno.text,sno.text,self.text) CusButton:
text: "-"
on_press: twonumcal.cal(fno.text,sno.text,self.text) CusButton:
text: "*"
on_press: twonumcal.cal(fno.text,sno.text,self.text)
```

15.6. GUI with check boxes

Various widgets are included in the Kivy library. Label, TextInput, CheckBox, Button are some common type of widgets.

GUI in figure 15.4 has 2 text entries for entering user's name and age. A group of check boxes is available for selecting title of the user (whether Mr. or Mrs. or Ms). Information in the last line will be printed after typing above information and clicking on the Click button.

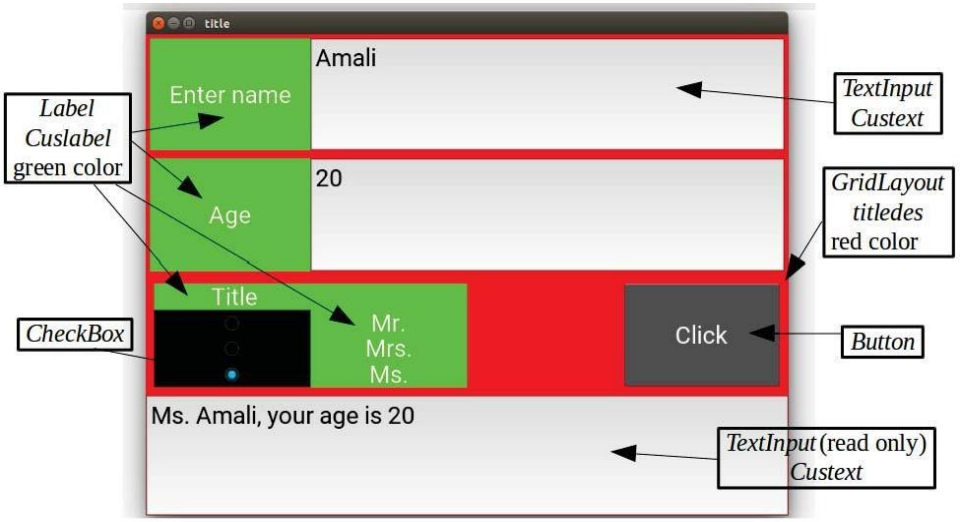


Figure 15.4 Text entry with check boxes

Now, let us analyse the two program codes title.py and title.kv. First, we will analyse the codes of title.py

```
1  importkivy
2  fromkivy.appimportApp
3  fromkivy.uix.gridlayoutimportGridLayout
4  class titledes(GridLayout):
5  definit(self,**kwargs):
6  super().init(**kwargs)
7  self.tt=""
8  def disptitle(self,name,age):
9  self.display.text=self.tt+" "+name+",yourageis"+age
10 defcngtt(self,x):
11 self.tt=x
12 class titleApp(App):
13 defbuild(self):
14 returntitledes()
15 titlapp=titleApp()
16 titlapp.run()
```

Line 1 to 3: import necessary kivy libraries

```
import kivy
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
```

Line 4: class “titledes” is derived from the class “GridLayout” in the kivy

```
class titledes(GridLayout):
```

Line 5 to 6: Initialize the class “titledes”.

```
def __init___(self, **kwargs):
super().__init__ (**kwargs)
```

Line 7: Introduce a variable called “tt” in the class “titledes”. This variable is used to receive title of the user which is selected through the checkboxes.

```
self.tt=""
```

Line 8 to 9: Method to receive name and age form the title.kv file and they combined with the title (self.tt). Output is self.display.text. The display is declared in the .kv file under the class “titledes” gridlayout for the “strtxt” Textinput widget which is the output display place.

```
def disptitle(self,name,age):
        self.display.text=self.tt+" "+name+", your age is "+age
```

Line 10 to 11: Method to receive label text relevant to activated checkbox and assign to the variable “tt” in the “titledes” GridLayout class.

```
def cngtt(self,x):
self.tt=x
```

Example 15.4 (b) title.kv file

```
<Cuslabel@Label>:
        font_size: 30 canvas.before:
                Color:
                        rgb:0,1,0
```

```
                        Rectangle:
                        pos: self.pos
                        size:self.size
<Custext@TextInput>:
        font_size: 30
        multiline:False
<titledes>:
        id: titledisp
        display:strtxt
        canvas.before:
                Color:
                        rgb:
                                1,0,0
                        Rectangle:
                                pos: self.pos
                                size:self.size
        rows: 4
        BoxLayout:
                padding:5
                Cuslabel:
                        text:"Entername"
                        size_hint_x: None
                        width: 200
                Custext:
                        id:nametxt
        BoxLayout:
                padding:5 Cuslabel:
                        text: "Age" size_hint_x:None width: 200
                Custext:
                        id:agetxt
        GridLayout:
                padding:10
                cols: 4
                BoxLayout:
                        width: 2 canvas.before:
                                Color:
                                        rgb:.1,.1,0.1
                                Rectangle:
                                        pos: self.pos
                                        size:self.size
                        orientation:'vertical'
                        Cuslabel:
                                text:"Title"
                                size_hint_x:None
                                width: 200
                        CheckBox:
                                id: c1
                                group:"title"
                                on_active:
titledisp.cngtt(ttmr.text)
```

```
                CheckBox:
                    id: c2
                    group:"title"
                    on_active:
titledisp.cngtt(ttmrs.text)
                CheckBox:
                    id: c3
                    group:"title"
                    on_active:
titledisp.cngtt(ttms.text)
            BoxLayout:
                width:20
                orientation:'vertical' Cuslabel:
                Cuslabel:
                    id: ttmr text:"Mr."
                Cuslabel:
                    id: ttmrs text:"Mrs."
                Cuslabel:
                    id: ttms text:"Ms."
            BoxLayout:
            BoxLayout:
                Button:
                    font_size:30 text:'Click' on_press:
titledisp.disptitle(nametxt.text,agetxt.text) BoxLayout:
            Custext:
                id: strtxt
                readonly:True
```

Second, lets analys the codes of the title.kv file.

Line 1 to 2: Declare the class called the “Cuslabel” which is an instance of the class “Label”. Its font size is 30.

```
<Cuslabel@Label>:
font_size: 30
```

Line 3 to 8: Declare Canvas, which is the root object used for drawing by a widget. These lines are focused to change color of the relevent widget. Here, it is “Cuslabel” Its color is green. The “rgb: 0,1,0” means red is 0, green is 1 and blue is 0.

```
canvas.before:
Color:
rgb: 0, 1, 0 Rectangle:
pos: self.pos size: self.size
```

Line 9 to 11: “Custext” is an instance of “ TextInput” class. Its font size is set to 30 and multiline is off.

```
<Custext@TextInput>:
font_size: 30 multiline: False
```

Line 12 to 21: “titledes” is declared in the title.py which is a “GridLayout” class. This class in the title.kv file is identified by the “titledisp”. “display: strtxt” is the read only textinput to be displayed the output. Background color of the GridLayout is set to the red (rgb: 1,0,0). Number of rows of the GridLayout is 4.

```
<titledes>:
        id: titledisp
        display: strtxt
        canvas.before:
Color:
        rgb: 1, 0, 0
Rectangle:
        pos: self.pos
        size: self.size
rows: 4
```

Line 22 to 29: BoxLayout is inserted to the first row of the GridLayout. It has a Cuslabel to display “Enter am” and a Custext to input the name.

```
BoxLayout:
        padding: 5
        Cuslabel:
                text: "Enter name"
                size_hint_x: None
                width: 200
        Custext:
                id: nametxt
```

Line 30 to 37: BoxLayout is inserted to the second row of the GridLayout. It has a Cuslabel to display “Age” and a Custext to input the age.

```
BoxLayout:
        padding: 5
        Cuslabel:
                text: "Age"
                size_hint_x: None
                width: 200
        Custext:
                id: agetxt
```

Line 38 to 40: Another 4 column GridLayout is inserted to third row to divide into 4 columns. Four columns are needed to insert checkboxes, labels of checkboxes, space and a button.

```
                GridLayout:
                        padding: 10
                        cols: 4
```

Line 41 to 49: BoxLayout is inserted to the first column of the third row. Widgets can be inserted vertically because orientation is 'vertical'. Color of the BoxLayout also changes.

```
BoxLayout:
                width: 2
                canvas.before:
                        Color:
                                rgb: .1, .1, 0.1
                        Rectangle:
                                pos: self.pos
                                size: self.size
                orientation: 'vertical'
```

Line 50 to 65: first row of the first column of the third row is allocated to label the “Title”. Other three rows are filled with three (3) CheckBoxes. Active CheckBox call the “cngtt” method of the ttle.py file with relevant text parameter of the Label of checkbox. All the CheckBoxes are grouped as “title”. When they are grouped, only one checkbox can be selected at a time.

```
Cuslabel:
        text: "Title"
        size_hint_x: None
        width: 200
CheckBox:
        id: c1
        group: "title"
        on_active: titledisp.cngtt(ttmr.text)
CheckBox:
        id: c2
        group: "title"
        on_active: titledisp.cngtt(ttmrs.text)
CheckBox:
        id: c3
        group: "title"
on_active: titledisp.cngtt(ttms.text)
```

Line 66 to 78: BoxLayout is inserted to second column of the third row. Widgets to this BoxLayout can be inserted vertically because orientation is 'vertical'. Cuslabel widgets are inserted to display text of checkboxes.

```
BoxLayout:
        width: 20
        orientation: 'vertical'
        Cuslabel:
        Cuslabel:
                id: ttmr
```

```
            text: "Mr."
        Cuslabel:
            id: ttmrs
            text: "Mrs."
        Cuslabel:
            id: ttms
            text: "Ms."
```

Line 79 to 84: Empty BoxLayout is inserted to the third column of the third row. Then BoxLayout is inserted to fourth column of the third row for inserting the Button for “Click”. When the Button is pressed, method “disptitle” will call with 2 parameters “nametxt.text” and “agetxt.text”.

```
BoxLayout:
BoxLayout:
        Button:
            font_size: 30
            text: 'Click'
            on_press: titledisp.disptitle(nametxt.text,agetxt.text)
```

Line 85 to 88: BoxLayout is inserted to the fourth row of the GridLayout. This BoxLayout has readonly Custext to display output.

```
BoxLayout:
        Custext:
        id: strtxt
        readonly: True
```

Activity	Activity 15.1 Write a GUI program using kivy libraries and kv language to input 2 boolean values (0 or 1) and select a boolean operation from set of boolean operations AND, OR, NAND, NOR, XOR. Result should be printed in a text box next to the boolean selection box.

Unit summary

Summary	In this unit you learned how to create a graphical user interface with kivy libraries using only Python as well as using both Python and Kv language. We also discussed calling methods in Python on actions of Kivy widgets.

References and Further Reading

1. Kivy Documentation Release 1.10.1.dev0, https://media.readthedocs.org/pdf/kivy/latest/kivy.pdf
2. kivy.org – official site

Appendix 1: Answers to Activities given in Python book

<table>
<tr><td>Unit01</td><td>Activity 1.1
1. Python is a programming language which has a very simple and consistent syntax. It allows beginners to concentrate on important programming skills. Python helps to introduce basic concepts such as loops and procedures quickly to students.
2. There is an interactive interpreter in Python which helps students to test language features while they’re programming. Students would be able to see both windows (the interpreter running and their program’s source) at the same time.
3. Good IDEs are available for Python.
4. Python language is intuitive and fun. Since Python is an open source programming language, it reduced up-front project costs as well.

Activity 1.2
Nowadays, Python is used in many application domains to cater for Web and Internet Development, Scientific and Numeric applications, Education applications, Desktop GUIs and Software Development.
Activity 1.3
See HELP documents if you cannot get the PATH setup correctly. Issues vary depending on the directory you installed Python and the operating system.</td></tr>
<tr><td>Unit-02</td><td>Activity 2.1
Typing each statement will give you following results.
>>> 55
55
>>> x=55
>>> x+1
56
In a new file from IDE type; 55</td></tr>
</table>

	x = 55 print(x) x + 1 print(x) Save the file and select 'run module' from Run. Output was; ====== RESTART: C:/Users/sarala/Documents/COL/Activity2-printscript1.py ====== 55 55 >>> **Activity 2.2** >>>width = 42 >>>height = 14 >>>width/3 14.0 >>>width/3.0 14.0 >>>height/3 4.6666666666666665 >>> 13 + 25* 10 263 **Activity 2.3** 1. >>>h=6 >>>r = 2 >>>pi = 3.141592 >>>volume= pi*r*r*h >>>print(volume)

Unit-03	**Activity 3.1**
	
	#Program to print even numbers between 20 and 60
	n = 20
	while n <= 60 :
	n = n + 2 print(n)
	print('All even numbers between 20 to 60 are printed')
	
	Activity 3.2
	# Use of For loop
	import math
	numberlist = [1,2,3,4,5] for no in numberlist:
	print(no, "cube of ", no, " is", (math.pow(no,3)))
	
	Activity 3.3
	>>>car = ('Toyota', 'Aqua', 2015, 'TA-181')
	>>>print(car)
	('Toyota', 'Aqua', 2015, 'TA-181')

Unit-04	**Activity 4.1** def findPrime(no): i = 0

	divisorList=[2,3,5,7,11,13,17,19]
	for j in divisorList:
	if (no != j)and (no % j == 0):
	i =1
	if i == 1:
	print("False")
	else:
	print ("True")
	findPrime(4)
	findPrime(1)
	findPrime(19)
	Required argument type is used
	
	Activity 4.2
	no = 397.234567
	print(round(no,3)) 397.235
	
	Activity 4.3
	>>> g = lambda x : x**2
	>>>print (g(9))
	81

Unit-05	**Activity 5.1**
	
	flower = "jasmine"
	index = len(flower)- 1
	while index >= 0:
	letter = flower[index]
	print (letter)
	index = index - 1
	
	Activity 5.2
	>>>flower[:] 'jasmine'
	
	Activity 5.3
	(i)
	str1 = 'Divya asked, '
	str2 = 'Great! Then can you let me have all the mangos, limes, oranges and apples?'
	
	str3 =str1 + str2
	
	print(str3) index = 0
	while index < len(str3):
	letter = str3[index]
	
	if letter.isalpha():

	print (letter)
	else:
	
	print(' ')
	index = index + 1
	
	(ii)
	str1 = 'Divya asked, '
	str2 = 'Great! Then can you let me have all the mangos, limes, oranges and apples?'
	str3 =str1 + str2
	print(str3)
	index = 0
	while index < len(str3):
	letter = str3[index]
	if letter.isalpha():
	print (letter, end =" ")
	else:
	print(' ', end =" ")
	index = index + 1

Unit-06	**Activity 6.1:** Write a Student class which contains studentID, lastName, courseID. Input values to one object of type student and print the values. class Student: def init (self, studentID, lastName, courseID): self.studentID = studentID self.lastName = lastName self.courseID = courseID >>>s1 = Student(); >>>print(s1.studentID, s1.lastName, s1.courseID) **Activity 6.2:** Write an init method for the Bike class that takes gear and speed and initialize them. # inside class Bike: Def init___ (self, gear =1, speed =0): Self.gear = 1 Self.speed = 0 **Activity 6.3:** Write a str method for the Bike class and print it.

	.# inside class Bike: Def str (self) Return('Gear %d%, speed '%(self.gear, self.speed)) >>>myBike = Bike(2,35) >>>print(myBike)

Unit-07	**Activity 7.1:**
	1. Write a Person Class. Make another class called Student and inherits it from Person class.
	2. Define few attributes that only have with Student class, such as school they are associated with, graduation year, GPA etc.
	3. Create an object called student
	Set some attribute values for the student, that are only coded in the Person class and another set of attribute values for the student, that are only in the Student class.
	4. Print the values for all of these attributes.
	
	Activity 7.2:
	A CEO buys a car. Later on the CEO buys two new cars BMW and a Mercedez. There is a driver for the CEO who chooses a car to drive to the office.
	1. Identify the classes involved in this scenario.
	2. Select appropriate superclass and subclasses
	3. Implement move method inside the superclass.
	4. Invoke the move method in superclass by creating instances of subclasses from a sub class.
	5. Implement the move method inside the subclasses.
	6. Override the move methods by creating instances of sub class.
	
	class Person:
	def init (self, name, surname, idno):
	self.name = name
	self.surname = surname
	self.idno = idno
	
	class CEO(Person):
	def init (self, xxx, *args, **kwargs):
	self.xxx = xxx
	super(CEO, self). init (*args, **kwargs)
	
	class driver(Person):
	PERMANENT, TEMPORARY = range(2)
	
	def init (self, employment_type, *args, **kwargs):
	self.branch = []
	self.employment_type = employment_type

	super(driver, self). init (*args, **kwargs)
	def enrol(self, yyy):
	self.branch.append(yyy)
	
	class vehicle:
	def init (self, model, make,color, gear, *args, **kwargs):
	self.model = model
	self.make = make
	self.idno = color
	self.gear = gear
	def move(self, gear):
	print("gear given is :", gear)
	
	class car(vehicle):
	def init (self, currFuel, miles, *args, **kwargs):
	self.currFuel = currFuel
	self.miles = miles
	super(car, self). init (*args, **kwargs)
	def move(self, miles):
	fuelNeeded = miles/10
	if self.currFuel <= fuelNeeded:
	print("Need re-fueling")
	else:
	print("Has sufficient fuel, can move")
	return {"changeSpeed", self.gear}
	BMW=car(23,34, 'x','xx','brown', 2)
	BMW.move(34)

Unit-08	**Activity 8.1:** List the different types of errors and explain how you can identify them separately. Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Runtime errors are produced by the interpreter if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Semantic errors are problems with a program that runs without producing error messages but doesn't do the right thing. In other words the program does execute the correct logic. **Activity 8.2:** What are exceptions and why is it important to handle them appropriately. State with examples.

	Errors detected during execution are called exceptions and are not unconditionally fatal. Essentially, exceptions are events that modify program's flow, either intentionally or due to errors. They are special events that can occur due to an error, e.g. trying to open a file that doesn't exist, or when the program reaches a marker, such as the completion of a loop. Example: When something goes wrong during the runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback. **Activity 8.3:** Describe what user-defined exceptions are. These are exceptions that Python allows the programmer to create based on his/her requirements. However it is better to check before creating an exception if there is already an existing one.
Unit-09	**Activity 9.1:** Explain the difference between white box testing and black box testing. White Box testing is where the internal structure of the system such as control structures of the program are tested and the Black Box testing is where the functionalities of the system is tested without going into the details of the implementation. **Activity 9.2:** Test suite: A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together. Explain the concepts used in unit testing. unittest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. A few of the concepts that is supported by Python includes: Test fixture: A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. Test case: A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. **Activity 9.3:** Write a test case for a string method that test for a "FOOD". Please refer to the unit testing video.
Unit-10	**Activity 10.1:** What is a debugger framework (bdb) and state each functions it handles with examples. The bdp module handles basic debugger functions, like setting breakpoints or managing execution via the debugger. The following syntax is used to define the exception which would be raised

	by the bdb class for quitting the debugger. exception bdb.BdbQuit **Activity 10.2:** What is a Python debugger (pdb) and what are the debugging functionalities it supports. The module pdb defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control. **Activity 10.3:** What is a profile and by referring to the examples above(in the text book), profile a function that takes in a single argument. It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application. To profile a function that takes a single argument, you can do: import cProfile import re cProfile.run('re.compile("foo|bar")')
Unit-11	**Activity 11.1:** Have to write the database as instructed
Unit-12	**Activity 12.1:** You need to research on the internet to find out the most popular Python libraries or packages available as open source repositories to use in different domains such as data science, image processing, information retrieval, data manipulation etc. Few such libraries are given below. SQLAlchemy: This is used as the Python SQL toolkit and Object Relational Mapper. Pillow : This is a Python Imaging Library(PIL) which adds image processing capabilities to the Python interpreter. Scrapy : A collaborative framework for extracting the data required from websites. NumPy: This is the fundamental package for scientific computing with Python. Matplotlib: This is a Python 2D plotting library which produces quality figures.
Unit-13	Activity 13.1: The Kivy launcher can be used to run the Kivy applications on Android devices without compiling them. Kivy launcher runs your Kivy application when you copy it inside the SD Card of your device.

	To install the Kivy launcher, you must go to the Kivy Launcher page on the Google Play Store. Then click on Install and select your phone. If not you can go to https://kivy.org/#download and install the APK manually. Once the Kivy launcher is installed, you can put your Kivy applications in the Kivy directory in your external storage directory. Often the application is available at /sdcard even in devices where this memory is internal. For an example /sdcard/kivy/<your application> <your application> should be a directory containing your main application file(e.g. main.py) and a text file (android.txt) with the contents given below . title= <Application Title> author=<Your name> orientation=<portrait|landscape>
Unit-14	***Assignment14.1:Answer***

```
from Tkinter import *
root = Tk()
def evaluateand():

        if n0.get()=="1" and n1.get()=="1":
                out.config(text="1")
                out.update_idletasks()
        else:
                out.config(text="0")
                out.update_idletasks()
def evaluateor():
        if n0.get()=="1" or n1.get()=="1":
                out.config(text="1")
                out.update_idletasks()
        else:
                out.config(text="0")
                out.update_idletasks()

root.geometry("300x100")
n0=Entry(root,width=2)
n1=Entry(root,width=2)
result=Entry(root,width=2) n0.delete(0)
n1.delete(0)
n0.place(x=20,y=1)
n1.place(x=20,y=40)
out=Label(root,text="")
out.place(x=150,y=20)
logicand= Button(root, text="AND",command=evaluateand)
logicand.place(x=50,y=20)
```

```
logicand= Button(root, text="OR",command=evaluateor)
logicand.place(x=50,y=50)
root.mainloop()
```

Assignment 14.2 Answer

```
import wx

class logicex(wx.Frame):
        def  init (self, title):
        wx.Frame. init (self,None,title = title,size = (400,200))
        panel = wx.Panel(self)
        self.n0=wx.TextCtrl(panel,value="",pos=(10,2))
        self.n1=wx.TextCtrl(panel,value="",pos=(10,100))
        self.result=wx.TextCtrl(panel,value="",pos=(300,50))
        lg = ['AND', 'OR', 'NAND', 'NOR', 'XOR']
        self.combo = wx.ComboBox(panel,choices =
        lg,pos=(90,50))
        self.combo.Bind(wx.EVT_COMBOBOX, self.oncombo)

    def oncombo(self,event):
        self.lsel = self.combo.GetValue() if self.lsel=="AND":
        if self.n0.GetValue()=="1" and self.n1.GetValue()=="1":
                self.result.SetValue("1")
        else:
                self.result.SetValue("0")
    app = wx.App()

    top=logicex("Logic Example")
    top.Show()
    app.MainLoop()
```

Unit-15

Assignment 15.1: Answer

andk.py file

```
import kivy
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.properties import ObjectProperty
class andkgrid(GridLayout):
                def callog(self,n0,n1,lg):
                if lg=="AND":
                        if n0=="1" and n1=="1":
                                self.display.text=str(1)
                        else:
                                self.display.text=str(0)
class andkapp(App):
                def build(self):
                        return andkgrid()
if  name  == ' main ':
        andkapp().run()
```

andk.kv file

```
<CusButton@Button>:
        font_size: 40
<CusText@TextInput>:
        font_size: 35
        multiline: False
<andkgrid>:
        id: andkcal
        display:result rows:3
        padding: 10
        spacing: 10
        BoxLayout:
                CusText:
                        id: q0
        Label:
```

	text: '' font_size: 40 Label: text: '' font_size: 40 BoxLayout: spacing: 10 Label: text: '' font_size: 40 CusButton: text: "AND" on_press: andkcal.callog (q0.text,q1.text,self.text) CusText: id: result BoxLayout: spacing: 10 CusText: id: q1 Label: text: '' font_size: 40 Label: text: '' font_size: 40
	***-Note that this line code belongs to the upper line**