

PYTHON

MASTERY

A Comprehensive Dual Guide for Beginners
Accelerated Learning with Hands On Exercises
and Real-World Applications

2 BOOK
IN 1



CHLOE ANNABLE

MASTERING PYTHON PROGRAMMING

A Comprehensive Beginner's Guide with Step-by-Step Instructions and Practical Exercises

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

CHLOE ANNABLE

First edition. January 10, 2024.

Copyright © 2024 Chloe Annable.

Written by Chloe Annable.

TABLE OF CONTENTS

[Title Page](#)

[Copyright Page](#)

[Introduction](#)

[chapter one: | Algorithm and information processing](#)

[chapter two: | Working with python strings](#)

[other supported samples include: | Unicode string](#)

[chapter three: | Data sequences, dictionaries, and sets](#)

[chapter four: | Math functions in python](#)

[chapter five: | Date and time functions](#)

[chapter seven: | Exception handling](#)

[chapter nine:](#)

[Final words](#)

[References](#)

INTRoDUcTIoN

This course introduces the core programming basics in handling algorithm development and program design with functions via the python programming language. At the end of the course, readers will have an in depth knowledge on data and information processing techniques.

Readers will be able to use various python features and modules to solve problems, create practical and contemporary python projects, and explore various software development challenges.

Readers will learn how to import the random module and use it to secure data by implementing a cryptographical method to securely generate a pseudorandom number such as a password or pin. They will also be able to use other modules such as the datetime module, time module, os module, and os.path module to perform various functionalities.

In this book, readers will learn how and why socket programming is the backbone of communication. (it makes the transfer of information between different devices and platforms possible.) The reader will be able to write a python code that allows him or her to connect to the server machine and access a web resource or connect to a client device and execute a client-server program.

He or she will also have in-depth knowledge on how communication between a client server network works and how cgi programming is applied in various chat bots, including those in facebook and whatsapp.

cHAPTER oNE:

ALGoRITHM AND INFoRMATIoN PRoCESSING

conceptual introduction

computers are versatile devices that perform various tasks and solve problems. You can take advantage of a computer's performance power by using programming languages to solve various problems or perform certain tasks.

In this book, we are going to use python, a powerful and easy-to-learn programming language. Python is an object oriented programming language and provides a platform for advanced programming. By the end of this module, you will be able to use python to analyze a problem and come up with a suitable solution to the problem.

In the context of computer science, a computer is a general purpose device that functions on a set of provided instructions and data. The instructions are executed in order to process data. computers receive instructions in the form of inputs, process the instructions one by one, and display the results in the form of an output.

computers focus on two interrelated concepts: algorithms and information processing.

Algorithms

An algorithm is a set of instructions that must be followed in order to perform a particular task or to solve a certain problem. Algorithms are not only applied to computers but to other areas as well. For example, if you followed instructions to cook a particular recipe, this set of instructions is the recipe's algorithms.

A programming algorithm acts as a recipe that describes the steps required for the computer to solve a certain problem or achieve set goals. In the cooking analogy above, the recipe acts as the procedure while the ingredients used in cooking act as the inputs. When the program follows the predetermined procedure, then it produces results in the form of outputs.

A programming algorithm gives instructions on how to do something and the computer will do exactly that. computer programmers write these instructions via a programming language, which is typically english-based, but the computers accept data in binary form. Therefore, the instructions have to be converted into a form the computers will understand (computer language).

Some programmers use pseudocode (a semi-programming language) to describe various steps of the program algorithm. Flow charts and diagrams can also be used to indicate the flow of data and the processes to be performed up to the end of the program.

Features of an algorithm

- contains a finite number of instructions
- consists of well-defined instructions
- Gives detailed description of the process that eventually terminates after arriving at the desired solution to the problem.
- Solves general problems

Example: simple algorithm example to request a user's email address through a web-based program.

Program pseudocode:

Step 1: start

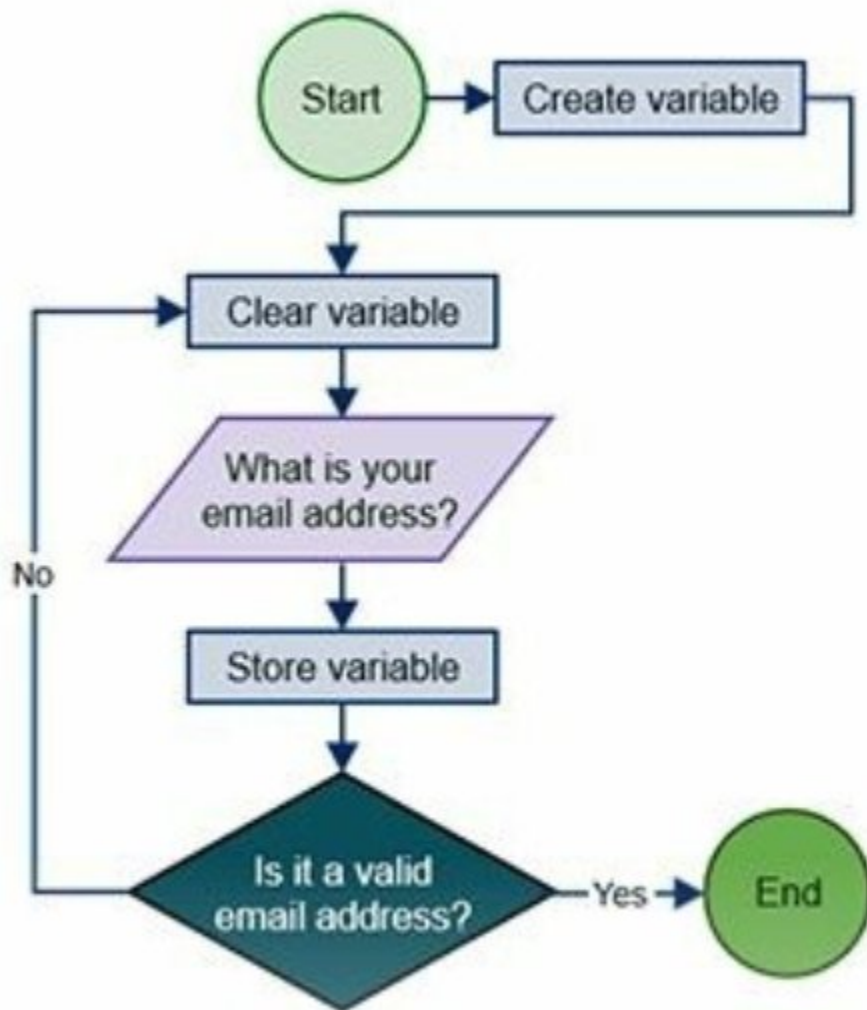
Step 2: enter a variable to store the user's email address step 3: if the variable is empty, clear it

Step 4: request the user to enter an email address step 5: the entered data is stored in the variable

Step 6: confirm whether the entered email is a valid email address step 7: not valid? Take the user back to step 3

Step 8: end

Flowchart to represent the above program



Every program procedure has a starting point and an ending point as shown above. When creating a program, you have to declare variables, which will create a memory space in the computer to store what the user types in as an input (shown in step 2).

once a variable is created, you have to clear it and free the memory to be used again. You don't want to have the old variable values mixed with the new content (step 3).

The user is requested to enter his or her email address in step 4. The email entered is stored in the variable created earlier.

In step 6, we prompt the computer to check whether the email address entered is a valid email or not, then make a decision based on the information returned.

If the email address entered is not valid, the computer takes the user back to step 3. otherwise, if it's a valid email address, then the program should terminate.

Using algorithms in python programming

To use the algorithm, you have to convert it from pseudocode to a programming language. computers accept data and instructions using binary or machine language. This makes it impossible to write computer instructions directly as it will be difficult for humans to read. Therefore, to simplify the programming process, programmers in the early 1940s introduced the use of assembly languages.

Assembly languages are computer instructions easily understood by humans that correspond to the machine language. The computer can't directly understand all programs written in assembly languages, however, so translation programs were created to supplement the assembly languages and translate assembly language into machine language.

Although the introduction of the assembly language led to improvements in the machine language, it remained a low level language that required several instructions to perform even simple tasks. To improve programming tasks, high-level languages were developed to make it easy to read and write program instructions.

Python programming is a good example of a high-level language. For example, writing a simple code to add numbers in python is shown below.

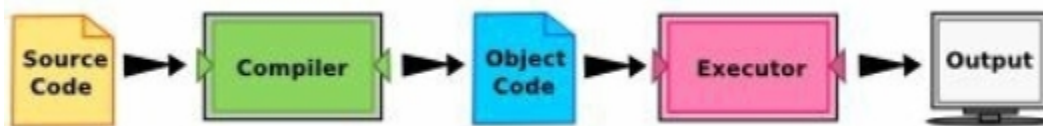
Sum = 15+25

This is a simple python code that is easy to read and write. The high-level languages makes it easy to read and write program code.

compilers and interpreters

High-level languages need translators to translate instructions into machine language prior to execution. Translation is done either one line at a time (line-by-line) or by translating the whole program at once. The translated instructions are stored in another file which is then executed.

Program instructions translated and executed line-by-line are associated with compiled or interpreted languages. Python programming is an interpreted high-level language.



A compiled program uses a compiler to compile program source code to an executable binary form. The compiler has to read the program instructions first before the program executes. The high-level language is the source code, while the translated program is known as the object code. once the program is translated, it can be executed repeatedly. No need for further translation.

An interpreted program, on the other hand, uses the interpreter to read instructions in source files and execute them. Programs using interpreters to execute instructions are less efficient and slower compared to compiled programs.



The python language is more suitable for non-resource intensive programs. It allows programs to be developed more easily and rapidly.

All high-level languages need to be processed first before they run.

Advantages of high-level languages

1.) Require less time to write the program code.

2.) Programs are short and easy to read compared to low-level languages 3.)

They are portable; you can run the program in different platforms

information processing

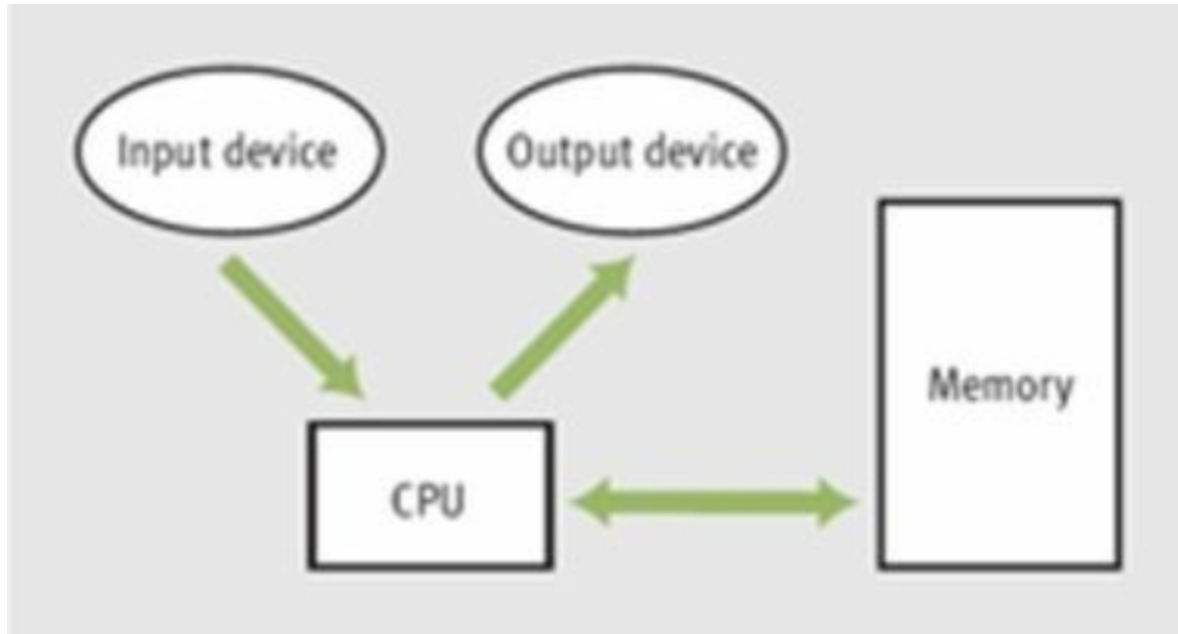
Information processing is the process of describing how user data is processed. The process involves a sequence of events which consists of information input, processing, and information output.

When carrying out the instructions of an algorithm, the computer program manipulates data inputs to produce the desired outputs. An algorithm is used to describe an information processing and it represents information.

Structure of modern computer system

A computer system consists of hardware and computer software. The computer hardware represents the physical devices used in the execution of algorithms while the computer software is a set of instructions or algorithms that represent a program written in any high-level programming language.

computer hardware



A computer's hardware consists of components that communicate with each other and exchange information.

A computer can also exchange information with the external world through the use of ports connected to other devices via a network.

A computer system consists of four components, namely:

1. *Input device*

An input device is any device that allows the computer to receive data from the user. The commonly used input devices include the keyboard, mouse, scanner, or microphone.

2. *Processing unit*

This is used to process user data. It converts the computer's raw data into meaningful information. The main processing unit is the central processing unit (cpu). There are other processing systems based on the function of the computer. For example, a graphics card system uses a graphics processing

unit (gpu). Gpu not only processes graphics but also processes general purpose programs.

3. Memory

This is where the processed data is stored. A computer's memory includes both the primary memory (ram and rom) and the secondary memory, which consists of hard drives, flash disks,

Dvds, and cds.

4. output

An output is a device used to display information to the user. Some of common output devices include the monitor, printer, and speakers.

central processing unit (cpu)

The cpu is an important component in the computer system that performs instruction processing functions. The cpu is subdivided into:

- Arithmetic and logic unit (alu): these are used to perform arithmetic calculations and comparison operations.
- control unit: it determine the instructions to be executed. That is, it determines what is to be executed and when. It coordinates activities between the alu, registers, and the main memory.
- Registers: they act as temporary storage areas for the processed data. The cpu is characterized by:
- clock speed: this is the operating speed for the cpu, and it is expressed in terms of cycles per second. The faster the clock, the faster the cpu.
- Instruction set: these are sets of instructions understood by the cpu.

computer memory

The computer's memory is used to store processed information for future reference. computer memory is divided into two parts: primary memory and secondary memory.

The primary memory is the main memory connected to the cpu for processing. It is also referred to as the random access memory (ram). Ram is a temporary type of memory which loses its content once the computer is switched off.

Secondary memory is a non-volatile type of memory which is permanently available. Secondary memory is available in large sizes and is cheaper than the primary memory. A good example of the secondary memory is the hard disk.

The operating system provides a high-level interface to the secondary memory. All executed programs are directly stored into the secondary memory in form of files.

computer software

A software is a set of computer instructions that tells the computer what to do and how to do it. A software is a sequence of instructions which are performed to solve a particular problem. A software is divided into three parts:

- System software
- Application software
- Programming languages

System software

System software is a collection of program instructions meant for the operation, control, and extension of the computer processing capabilities. System software consists of programs that are written in low-level languages, and they interact with the computer hardware in order to accomplish a specific task.

System software is divided into two: operating system and utility software. Translation programs like compilers, interpreters, and assemblers form part of system software.

Features of system software

- Fast in speed
- operate close to the system
- Not easy to design and understand
- Written using low-level language
- Difficult to manipulate

operating system: operating system (os) is a program that translates computer inputs into a form computers can understand. os acts as an interface between the various components of the computer.

The os schedules program tasks and controls the performance of peripheral devices. It manages the computer hardware, software resources, and control services of other computer programs.

Utility software: this is a program designed to help analyze, optimize, and maintain other computer programs or resources. Examples of utility software include antivirus software, backup utility, copy utility, and dump utility.

Utility software perform tasks which ensure the smooth running of the computer system. They extend the functionality of the operating system.

Application software

An application software is a program designed to perform a specific function. The software product is designed to satisfy a specific need. It consists of a collection of software packages that work together to achieve a common goal.

Application software rely on the os for the running of the programs. Application software uses the operating system to make requests for services via the application program interface, or api.

Application software include:

- Accounting software: quickbooks, pastel, sage, payroll software
- Application packages: spreadsheet, word processor, database management, publisher, and presentation package.
- Inventory management system, income tax software, etc.

Application software programs are written in high-level languages, and are easy to manipulate and understand.

Programming languages

These are software programs used exclusively by computer programmers. Programming languages are high level languages that allow programmers to write program instructions to solve a specific task or build other applications.

For example, you can use programming platforms like c++, java, php, python, or other classes of languages for product development.

computer program

A computer program is written to solve a specific problem or to achieve a particular task. Before writing any program, you should understand the

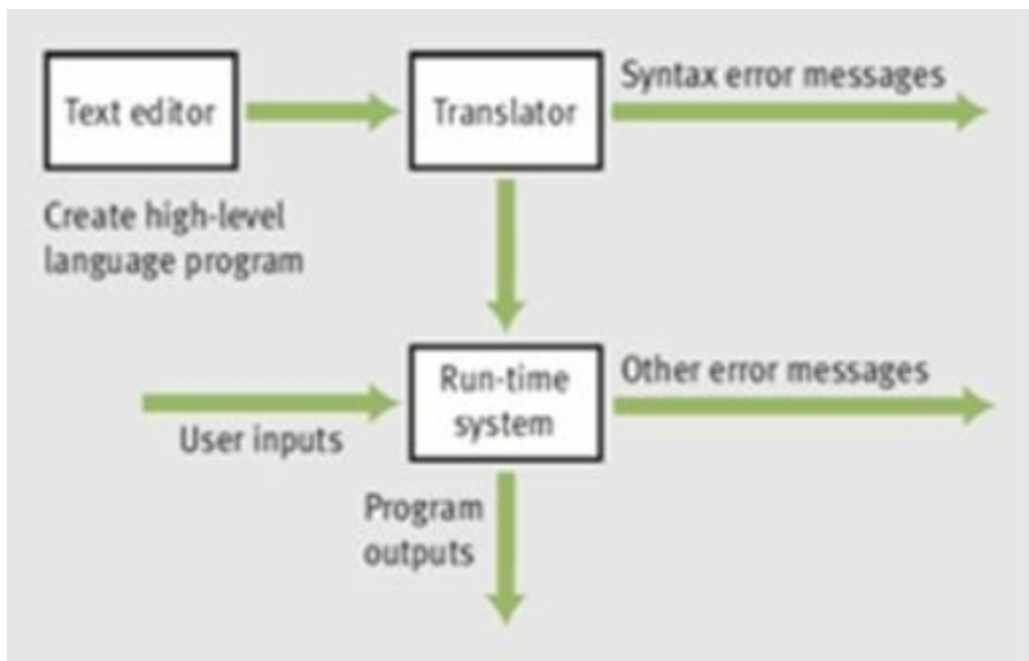
problem thoroughly so as to be in a better position to solve the problem.

Understanding the problem will help you come up with a useful plan aimed at providing a solution to the problem. This involves coming up with the inputs and outputs required. You can also structure the list of requirements in the form of diagrams like flow charts.

once the above is done, you can come up with an algorithm to solve the problem. An algorithm is a sequence of steps as discussed earlier. once you have come up with the algorithm, the next step is writing the program code to solve the identified problem.

Python programming is one of the most powerful object oriented programs easy to use and understand. Some parts of the algorithm will be straight forward to translate in python.

After writing the program, check for errors. There are different types of errors you may come across when you run the program. These errors are called bugs, and the process used in tracking these errors and correcting them is known as debugging.



Programmers make errors when writing a program code. A python interpreter will be able to detect these errors. Some of these errors include syntax errors, runtime errors, and semantic errors. In order to track these errors down, you have to know the differences between them.

Syntax error

When a python interpreter encounters an error in the program, it terminates the program and displays an error message to the user. Syntax represents the structure of a program and the rules of declaring that structure. If there is a single error, python will quit and you will not be able to run the program.

If you're new to programming, you may spend a few days tracking syntax errors. Once you become familiar with the language, however, you will make fewer errors, and it will be easy to track them.

Runtime error

A runtime error occurs after running the program. That is, the error will not show up until you run the program. Runtime errors are commonly known as exceptions as they indicate something has already happened.

Semantic errors

If a program has a semantic error, the program will run successfully and the python interpreter will not generate the error message. The program will run up to the end but will not perform the task it was meant to do.

To correct semantic errors, you have to look at the program output and work backwards by analyzing what each output was supposed to do.

Python basics

At this point, you already know how to set up a python development framework from *python programming book one*. In this book, we will discuss how to get started using the interactive interpreter.

When you enter *python* on the command line with no parameters, it will automatically launch the python interpreter. The python interpreter is a text console which allows you to enter python code line by line and be interpreted on the fly.

Python interactive shell

While the python interpreter executes program code line by line, the interactive shell, or python shell as commonly known, executes a single command to obtain the results. The interactive shell waits for the user's inputs, and once the user enters the input command, the program is executed immediately and the results displayed.



You can open python shell in a windows computer by opening a command prompt, and then writing *python*, and then pressing enter.

When you run the shell, the three greater than symbols (`>>>`) are shown. From the three symbols, you can enter single-line python statements and get results immediately. For example, enter values to add, multiply, and divide as shown below:

A screenshot of the Python 3.6.3 Shell window. The window has a blue title bar with the text "Python 3.6.3 Shell" and standard window controls. Below the title bar is a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following text:

```
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> 3+2
5
>>> 3*2
6
>>> 9/3
3.0
>>> |
```

The status bar at the bottom right indicates "Ln: 9 Col: 4".

Execution of python script

Python shell is used to execute a single statement. If you have multiple statements to be executed, you have to create a python file and write python scripts to execute (multiple statements) instead.

For example, create a code to output the value of p. The *print ()* function will be able to output the value of p.

P=5;

Print (p)

Interpreters are essential when testing a small piece of code before using it on a complex program. This helps you know how the function works as well as ensure that the syntax of the code is correct.

An interpreter tests code snippet, explore functions, and modules. If you want to save the program, you have to write the program code into a file. All python programs are saved with an extension *.py*. You can run the code anytime using the *python* command followed by the file name as the parameter.

Python example.py
chapter summary

Algorithms and information processing are the fundamental components of computer science. Programmers use these components to come up with program instructions aimed at solving various problems. Software provides a platform where different algorithms can run on a general- purpose hardware device. This chapter discussed:

- How to write a program algorithm.
- How to use flowcharts and pseudocode to write a program.
- How compilers and interpreters work in a program.
- Python programming language is a high-level language. Interpreters translate

Instructions written in python to a lower level form which is then executed on a normal computer system.

- Information processing of data using various computer system components.
- Highlights on various computer architecture devices and how each functions.
- The importance of computer software in handling the hardware components as well as the different types of softwares and examples in each.
- Python basics and the use of the python shell, which provides a command prompt to evaluate and view python expressions and statements.
- When the python program is executed, it is translated to byte code.

In the next chapter you will learn about python strings—how to use slice syntax to represent a substring, string methods, and other string manipulation techniques.

CHAPTER TWO:

WORKING WITH PYTHON STRINGS

Python strings

Python uses built-in strings assigned to variables. Strings are created by enclosing python statements in quotes, either single (') quotes or double (") quotes. creating a string in python is as easy as creating variables and assigning values.

Assign a string to a variable

```
x="Hi!"
```

```
print (x)
```

Assigning strings to variables is accomplished by creating a variable, followed by an equal sign and then the string.

A string can span multiple lines with each line separated by a backslash (\) at the end. String literals can use three double quotes or three single quotes to represent multiple lines of text.

Strings are immutable. That is, once you create strings in python, you can't change them. Therefore, you have to create new strings to represent computed values. For example, ('hello' + 'world') has two strings, 'hello' and 'world', which are used to build a new string 'helloworld'.

Accessing characters in a string

The characters in a string are accessed with the standard [] syntax just like in arrays. Python uses zero indexing to initialize the string. If the string index is out of bounds, an error is generated. Python stops executions if it can't tell what to do, in which case it sends an error message.

To extract characters from the string, enter the string name followed by the index number. For example, to retrieve character 'e' from string 'hello', you write `hello[1]`. The `len(string)` function returns the string length.

The `[]` syntax and `len()` function works in any sequence in strings or lists. Python uses the '+' operator to concatenate two strings.

Example:

```
A="hello"
```

```
Print a[3]
```

```
Print len(a)
```

```
Print a + 'there'
```

```
// output l
```

```
5
```

```
Hello there
```

In some cases, the '+' operator doesn't convert numbers into string type. To do so, use the `str()`

Function that converts values to string form so as to combine them with other strings. Example:

```
Pi=3.14
```

```
confirm= 'the value of pi is' + str(pi)
```

```
// output yes
```

String slicing

The slice syntax is a great way to return sub-parts of a string sequence. If you want to return a range of characters, you can use the slice syntax. This allows you to specify the start index and the end index separated by a colon.

To return part of a string in 'hello world', write:

```
X='hello' print (x[1:4])
```

```
//output 'ello'
```

A diagram illustrating string indexing for the word 'Hello'. The word is written in a large, black, serif font. Below each letter, there are two rows of indices. The first row shows positive indices starting from 0: 'H' is at 0, 'e' at 1, 'l' at 2, 'l' at 3, and 'o' at 4. The second row shows negative indices starting from -5: 'H' is at -5, 'e' at -4, 'l' at -3, 'l' at -2, and 'o' at -1. The indices are in a smaller, black, sans-serif font.

In this case, the characters start at index 1 and extend to others characters but doesn't include the index 4.

`X[1: ]` will return 'ello' – it omits the index defaults at the end (or beginning for `x[:4]`) of the string.

`X[:]` will return 'hello' – since no string index is specified, the code returns the whole string (this is commonly used when you want to copy the entire string or list).

`X[2:100]` will return 'llo'. If you're working with a big index (in this case 100), it will be truncated down to a string length.

Negative indexing

You can use negative indexing to write a slice syntax from the end of the string. The `x[-1]` is the last char 'o' and so on.

X[-1] returns 'o' – last character

X[-4] returns 'e' – the fourth character from the end.

X[:3] returns 'he' – represents characters from the start of the string but doesn't include the last 3 characters.

X[-3:] returns 'llo' – represents characters starting with the third character from the end of the string and extends up to the end of the string.

Note: python doesn't have any separate scalar 'char' type. If you have a string x[10], python will return string_length-1 characters. The operators ==, <=, or >= all works the same way. That is, for any index n, the $x[:n] + x[n:] = x$. This also applies to the negative indexing, -n. The $x[:n]$ and $x[n:]$ partition a python string into two, thus conserving all the characters.

String methods

Python has built-in functions or methods that work well with strings. The string method runs 'on' an object. If a created variable is a string, then the *lower()* or *upper()* functions run on the string object and return the results. The concept of running methods on objects is why python is considered an object oriented programming language.

Some of these string methods include:

- *Strip()* which is used to eliminate the white spaces from the beginning or the end of a string.
- *Lower()* and *upper()* methods returns the string in lower case and upper case respectively.
- *Startswith(other)* and *endswith(other)* methods measure the start and end of the string.

- *Isalpha()*, *isspace()*, and *isdigit()* are used to test whether the string characters belong to a certain class character.
 - *Find(other)* is used to search for a particular string within the created variable and return the first index where it begins or a -1 if it's found.
 - *Join(list)* is used to join together elements in a given list using the string as the delimiter. For example, `join('xxx', 'yyy', 'zzz')` -> `['xxx', 'yyy', 'zzz']`.
 - *Split('delim')* returns substring list separated by a given delimiter. Delimiter in this case acts as text and not a regular expression that is, `'xxx,yyy,zzz.split(',')` -> `['xxx', 'yyy', 'zzz']`. The *split()* with no arguments splits characters with white spaces.
 - *Replace('old', 'new')* returns a string where the old string characters are replaced with new characters.
 - *Format()* is used to format specified values within a string.
 - *count()* returns the number of times a certain character occurs in the string.
- Example 1:

```
x=Hello World!
```

```
Print (x.lower())
```

This code returns the string in lowercase letters. Example 2:

```
x=Hello World!
```

```
Print (x.replace('H', 'M'))
```

When you run the code, the 'hello' string will be replaced with 'mello'.

check string

check string is used to find out whether a certain character is present in a string. It uses the keyword *in* and *not in* to check.

Example:

```
txt= "Welcome toPython programming for advanced users"
```

```
x="ing" in txt
```

```
print(x)
```

To check if the phrase “ing” is not in the text:

```
txt= "Welcome toPython programming for advanced users"
```

```
x="ing" not in txt
```

```
print(x)
```

Special string operators

Strings can be manipulated using one of the following operators.

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n

String formatting operator (%)

Python consists of the string formatting operator %. The % operator is a unique string feature that uses functions from the c language family's printf() function. Python accepts a printf-type string format with %d for integer values, %s for string, and %f for floating point numbers on the left and matching values on the right in a tuple. A tuple consists of values grouped within parentheses and separated by commas.

Example:

Print ("my name is %s and my weight is %d kg!" %(‘faith’, 50)) when the above code is executed, it will display:

My name is faith and my weight is 50kg!

The list below is a set of samples that can be used together with the % operator.

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

oTHER SUPPoRTED SAMPLES INcLUDE: UNiCoDE STRING

Normal python strings are stored using the 8-bit ascii code. Unicode strings are stored using

The 16-bit unicode. This enables the string to accommodate a varied set of characters.

chapter summary

In this chapter, you learned:

- How to create strings in python
- How to access characters in a string
- How to use slicing feature to return sub-parts of a string
- How to apply various methods to manipulate strings like *isalpha()*, *isspace()*, *isdigit()*, *strip()*, and *find()* among other methods.
- How to format strings using the % operator.

In the next chapter you will learn about data sequences, basic list operators, and working with dictionaries and sets.

cHAPTER THREE:

DATA SEQUENcES, DIcTIoNARIES, AND SETS

Python sequence

A sequence is a built-in data type that consists of a series of values bound together by a container. It enables storage of multiple values in an organized and efficient manner. All streams created in python form a sequence.

The order in which the objects are organized is similar to the order in content is retrieved from the objects.

Types of sequences include:

- String
- List
- Tuple
- Xrange objects
- Unicode
- Buffers
- Byte arrays

Strings

In the previous chapter, we covered strings in which an array element or string character is accessed with square brackets or by using the subscript operator. The arrays use indexes which are numbered from 0 to n-1, where n is the number of characters in the variable.

Example:

```
>>>"tuple" [0]
```

output 't'

```
>>>"tuple" [1]
```

output 'u'

```
>>>"tuple" [2]
```

output 'p'

```
>>>"tuple" [3]
```

output 'l'

```
>>>"Tuple" [4]
```

Output 'e'

You can also create negative indexes by counting the string from the end.
You can declare an empty string using the str() function. For example:

```
>>>type (varname)
```

```
>>>varname=str()
```

```
>>>varname
```

Lists

A list is a group of values arranged in a specific order. It acts as a container that holds various objects in this particular order. The list type is essential for

the implementation of sequence protocol. It also makes it easy to add and remove objects from a sequence.

The list elements are enclosed in square brackets. For example, you can create an empty list that is initiated as follows:

```
>>>demo= []
```

You can pass values to the list. Use commas to separate the values.

```
>>>demo ["book", "pen", "10"]
```

In this case, the list hold values of different data types. The above example holds strings “book” and “pen” as well as the number 10.

Just like string characters, the items in a list are accessed by the use of indexes starting at 0 up to n-1. If you want to access a specific item in the list, you can refer to it by using the list name followed by the item’s number inside the square brackets.

```
>>> demo ["book", "pen", "10"]
```

```
>>>demo [0]
```

output “book”

```
>>> demo [1]
```

output “pen”

```
>>>demo [2]
```

output “10”

If you use negative numbers, it will count backwards:

```
>>>demo [-1]
```

```
10
```

```
>>>demo [-2]
```

```
“pen”
```

```
>>>demo [-3]
```

```
“book”
```

Unlike other programming languages, python uses the same syntax and function name to execute sequential data. For example, you can use the *len()* function to determine the length of the string, list, or tuple. It returns the number of items in the array.

```
>>>len(demo)
```

Example 1:

```
3
```

Example 2:

```
>>>subjects= ["English", "Maths", "chemistry", "Biology"]
```

```
>>>len(subjects)
```

```
// output 4
```

Just like in strings, lists can also be sliced.

```
>>>demo [1:]
```

```
['pen', 10]
```

```
>>>demo[:-1]
```

```
['book', 'pen']
```

```
>>>demo.append(20)
```

```
>>>demo['book', 'pen', 10, 20]
```

You can also add more items to the list by using the *append()* function, although there are various ways to add more items to a list:

If you specify the index outside the range, the code will generate an error message. Therefore, all items should be added inside the range. Alternatively, you can use *insert()* if you want to add items or elements in a particular index.

```
>>>demo.insert(1, 'and')
```

```
>>>demo['book', 'and', 'pen', 10, 20]
```

Items can be deleted from the list using the *del()* function.

```
>>>demo['book', 'and', 'pen', 10, 20]
```

```
>>>del demo[1]
```

```
>>>demo['book', 'pen', 10, 20]
```

In this example, the *del ()* function deletes the string 'and' from the list. The list then re-orders

itself to ensure there is no space left between the items.

Lists also portray an unusual characteristic in that, if you have two items in the list: x and y, then set y to x and change the value of x, y will change

automatically as well.

```
>>>x=[5,6,7, 8]
```

```
>>>y=x
```

```
>>>del x[3]
```

```
>>>print x [3, 4, 5]
```

```
>>>print y [3, 4, 5]
```

Properties of a list

- They're represented in an ordered manner
- They have arbitrary objects
- List elements can be changed hence, they're mutable
- They have variable size
- Lists use index to access elements in the list

Tuples

Tuples works the same as list, except they are immutable. If you have a set of tuples, you can't change it. That is, you can't add or delete elements or change them once created. Tuple elements are enclosed in parentheses instead of square brackets, and they use the same rules of indexes as in lists.

Advantages of using tuples:

- They're faster when compared to list execution
- Prevents any accidental changes to the data. If you have data that doesn't require changes, use tuples instead of lists.

- Used as keys in a data dictionary. This is one of the major benefits of using tuples compared to the use of lists.

Tuples are declared as follows:

```
>>>tup= ("tuples", "can't", "be", "changed")
```

Bytes object

This is a sequence of integer number elements with bytes objects in a range of 0 to 255. These numbers correspond to the ascii characters and printed as that.

Bytes arrays

A byte array is the same as byte object except that it is mutable. The byte array returns an array of a particular byte size.

```
>>>x=bytearray(5)
```

```
>>>x
```

Bytearray(y'\x00\x00\x00\x00\x00') to change byte array values:

```
>>>x=bytearray([4,5,6,7,8])
```

```
>>>x
```

```
Bytearray(b'\x04\x05\x06\x07\x08')
```

Python sequence operations

These are operations carried on a sequence. They include:

concatenation

To combine two sequences is as easy as adding two numbers. You can combine lists or strings together. For example:

```
>>>fname= "robert"
```

```
>>>lname="mogare"
```

```
>>>name= fname + " " + lname
```

```
>>>print (name)
```

The augmented assignment operator += can also work well in sequences.

```
a+=b is syntactically equivalent to a=a+b
```

This is syntactically the same but the implementation of the two is different. In a+=b, the left side is only evaluated once. The += operator is used on mutable objects as an optimization.

Repetitions

```
abc*4
```

In the examples above, we have used '+' operator for sequences. You can also use '*' operator to point to values. The '*' represent a sequence or an integer value. For example, a*n or n*a represent an object that concatenate n-times.

Is concatenated to

```
abc+ abc+ abc+ abc
```

Membership

This is used to check whether a certain element is included in the string or not. That is, whether an element is a member of the sequence. It uses the 'in'

or 'not in' operator.

```
>>>'ten' in 'concatenated'
```

```
True
```

```
>>>'t' not in concatenated
```

```
False
```

Slicing

When you want only a part of the sequence and not the entire string, you use the slicing operator.

```
>>>'slicing' [1:4] 'lic'
```

The slice operator allow you to extract a specific part of the string by using the index (the start and the end index with the index numbers separated by a colon).

Python sequence functions len()

The len() function is an essential function used in passing a sequence. The function returns the length of a python sequence.

```
>>>len (sequence) 8
```

Min () and max ()

The *min()* and *max()* functions return the lowest value and the highest value in a python sequence.

```
>>> min(1,2,3) '1'
```

```
>>>max (1,2,3)
```

'3'

The comparison of numbers is based on the ascii values.

Python index ()

This is a python sequence method that returns the index on the first occurrence of a value.

```
>>> world.index('r') 2
```

Python count()

The count () function returns the number of times a value occurs in a string.

```
>>>banana.index('an') 2
```

```
>>>sequence.index('e') 3
```

Python collections

Unlike the sequences, python collections do not have a deterministic order. When dealing with collections, the ordering of items is arbitrary and they have a physical order.

Data dictionaries and sets are containers to sequential data.

Python sets

A set is a python element that holds a sequence of data values. It consists of a group of items that do not hold duplicates. Sets are arranged in a sequential manner but do not support indexing.

creating a python set

A set is declared by typing a sequence of values separated by commas inside curly braces, then assigning the values to a python variable.

They are declared in two ways: assigning to python variables and using the `set()` function. Assigning to python variable:

```
>>> num1={3,2,1,2}
```

```
>>>num1 1,2,3
```

Using the `set()` function:

```
>>> num1= set([3,2,1])
```

```
>>>num1 1,2,3
```

`>>>num1= set ()` creates a set with an empty object. If you create an empty set, it creates an empty dictionary.

Python sets do not support indexing. Therefore, you have to access the entire set at once. From the above example, you can access the entire set at once.

```
>>>num1
```

```
{1,2,3}
```

When you call variable `num1`, the elements are reordered and represented in an ascending order. Since sets don't support indexing, you can't use slicing operator on them. You also can't delete a set element using indexing. If you want to delete an element from the list, you have to call the `discard()` or `remove ()` method. Each method accepts the item to be deleted as an argument.

```
>>> num1= set([3,2,1])
```

```
>>>num1.discard (2)
```

```
>>>num1
```

```
{1,3}
```

```
>>>num1.remove(1)
```

```
>>>num1
```

```
{2,3}
```

The two methods achieve the same results but work differently. If you delete an element that doesn't exist in the dictionary, the *discard()* method ignores it while the *remove()* returns a *keyerror* message.

The pop () method

You can call the *pop()* method in sets to pop out arbitrary items. The method doesn't take an argument since it doesn't support indexing and prints out the item that was popped.

```
>>>num1.pop () 1
```

Updating a set

Sets are mutable but do not have mutable items like lists. As we have seen, you can't reassign a set using indexes. Therefore, to update the set, you can use the *add()* and *update()* methods.

Add ()

The *add ()* function accepts the argument to be added to the set.

```
>>>values= {2, 3, 5, 5.5}
```

```
>>>values.add (6.5)
```

```
>>>values
```

```
{2, 3, 5, 5.5, 6.5}
```

If you add an already existing item in the set, the set will remain the same. The new added item will not be reflected in the set.

Update ()

The *update ()* method adds multiple items into the set. It allows you to pass several arguments into the set at once.

```
>>>values.update ([6, 7, 8, 2, 8.2])
```

```
{2, 3, 5, 5.5, 6, 7, 8, 8.2 }
```

Function used on sets

Sets use functions to perform a certain action or operation and return a value. The commonly used functions in sets include:

1. Len ()

The *len()* function returns the number of elements in a set.

```
>>>len (days) 7
```

1. Max ()

This function returns the highest value in a group of elements.

```
>>>max ({2, 3, 5, 4, 6})
```

```
6
```

You can also use *max ()* on strings. For example,

```
>>>days = ({'mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'})
```

```
>>>max (days) wed
```

This function returns wednesday as the maximum item as it uses ascii values and looks at the alphabet for each day, m t w t f s t, with letter w being the highest value. Thus, the code returns wednesday as the highest value in the set of days

1. *Min()*

The *min ()* indicates the lowest value in a set. To get the lowest value in a set of days use:-

```
>>>min (days) fri
```

Letter f is the lowest ascii value in the set.

1. *Sum()*

The *sum ()* function is used to add numbers together. It performs arithmetic sum in a set and returns a value.

```
>>>sum ({2, 3, 5})
```

```
10
```

The *sum ()* function cannot be applied on a string.

1. *All()*

This function returns true if the values in the set have a boolean expression with a true value. otherwise, it will return false.

```
>>>all (days) true
```

1. *Sorted()*

The function sorts a number of items in a set of lists. The items are sorted in ascending order without modifying the original values.

```
>>>values = {1, 2, 4, 7, 6, 5.2, 8, 9}
```

```
>>>sorted (values) [1, 2, 4, 5.2, 6, 7, 8, 9]
```

Methods on sets

Set methods are used to alter a set unlike functions, which perform actions on sets. Methods perform a sequence on any operation on a set. The methods are called on a set and they include:

1. Union()

This method is used to perform union operation on two or more sets. It returns the items in all the sets. For example:

```
>>>num1, num2, num3 = {2, 3, 4}, {4, 5, 6}, {6, 7, 8}
```

```
>>>num1.union (num2, num3) [2, 3, 4, 5, 6, 7, 8]
```

The above method didn't alter the sets. Therefore, not all methods called will alter the set.

1. Intersection()

This method accepts an argument in the set and then returns all common items in the selected set.

```
>>> num2.intersection (num1) 4
```

```
>>>num2.intersection (num1, num3) set ()
```

Intersecting the three sets returned an empty set since there is nothing common in the three sets.

1. Difference()

This method is used to return the difference between two or more sets. It returns values in the form of a set.

```
>>>num1.difference (num 2)
```

```
{2,3}
```

1. Symmetric_difference ()

This returns all elements which are unique to each set.

```
>>>num1.symmetric_difference (num2)
```

```
{2, 3 5, 6}
```

2 and 3 are available in num1 while 5 and 6 are in num2. It omitted 4 because is available on both sets.

1. Intersection_update ()

This function returns an item which is common to both sets. The method doesn't update the set

on which a call is placed.

```
>>>num1.intersection_update (num2)
```

```
>>>num 1
```

```
{4}
```

1. copy()

This method is used to create a shallow copy of the set

```
>>>num4= num1. copy()
```

```
>>>num1, num4
```

```
{2, 3, 4}, {2, 3, 4}
```

1. *Isdisjoint()*

If two sets have a null intersection, then this method will return true.

```
>>> {2, 3, 4}.isdisjoint({5, 6, 8}) true
```

The `isdisjoint ()` only accepts one set of arguments. If you have more than one set, it will return an error.

1. *Issubset ()*

It returns true if a particular set in an arguments contain another set.

```
>>> {4, 5}. Issubset ({4, 5, 6})
```

True

```
>>> {1, 3} .issubset ({1, 3}) true
```

1. *Issuperset()*

Just like `issubset ()`, `issuperset ()` returns true if one set has arguments in the set.

```
>>> {2, 3, 4}.issuperset({1,2}) false
```

```
>>> {2, 3, 4}.issuperset ({2}) true
```

Python boolean

Booleans are a type of data that returns a value as either true or false. Set methods like *isalpha ()*, *issubset ()*, and *issuperset ()* return a boolean value true or false.

Declare boolean data types as you would declare a normal integer.

```
>>>days= true
```

If you add quotes to the value, it becomes a string instead of boolean.

The bool () function

The *bool ()* function is used to convert a value into boolean type.

```
>>>bool ('data') true
```

```
>>>bool ([]) false
```

Different data values have different boolean values equivalent to them. You can use the *bool ()* set function to find out these values. For example, the boolean value for 0 is false while the boolean value for 1 is true. Any value greater than 0 like 0.00001 has a true boolean value.

```
>>>bool (0) false
```

```
>>>bool (0.0000001)
```

True

A string with values has a true boolean value while an empty string has a false boolean value.

```
>>>bool (' ') true
```

```
>>>bool ('') false
```

An empty construct has a false boolean value while a non-empty construct has a true boolean value.

```
>>>bool (()) false
```

```
>>>bool ((2, 5, 3))
```

True

Boolean operations

You can perform arithmetic calculations on sets, in which a 0 value returns false and a 1 returns true, and then apply arithmetic operators to them. Some of these arithmetic operators include:

1. Addition

This is used to add two or more boolean expressions.

```
>>> true +false //equivalent to 1+0 1
```

```
>>>true+true //equivalent to 1+1 2
```

```
>>> false+false //0+0 0
```

```
>>>>false+true //0+1 1
```

1. Subtraction and multiplication

You can also adopt the same mechanism when performing subtraction and multiplication operations.

```
>>>>false – true //0-1
```

-1

1. Division

When you divide boolean numbers, it results in a floating number value.

```
>>>>false/true 0.0
```

```
>>>>true/false zero division error.
```

This returns an error message since you can't divide a value by zero.

1. Modulus, exponential and floor division.

All rules applied in addition, multiplication, and division also apply to modulus operators as well as in exponential and floor division operations.

```
>>> false % true
```

```
>>>true ** false 1
```

```
>>>>false **false 1
```

```
>>>false // true //equivalent to 0//1
```

Relational operators

The following relational operators are also applied to boolean values.

operator Description

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

!= Not equal to

== Equal to

Example:

```
>>>0<=1
```

True

0 is less than or equal to 1.

```
>>>>false>>true false
```

The value of false is 0 while the value of true is 1. As 0 is not greater than 1, false is returned.

Bitwise operator

Bitwise operators use bit-by-bit operation. Assume you have or code with 2(010) bits and 2(020) bits. This will result in 4(030) bits.

```
>>>2/3 5
```

common bitwise operators used in boolean expressions include:

1. Bitwise and (&)

This operator only returns true if both boolean expressions are true.

```
>>> true & false false
```

```
>>> true & true true
```

1. Bitwise or(|)

This operator only returns a false value if both of the values are false.

```
>>> false | true true
```

1. Bitwise xor (^)

This operator returns a true value if one of the values is true and the other value is false. It returns false if the two values are the same

```
>>> true ^ true false
```

```
>>> true ^ false true
```

3. Binary 1's complement

The binary 1's complement calculates a value to either true indicated by 1 or false indicated by 0.

Python dictionary

A dictionary holds real, mutable values. A dictionary allows you to add, change, or remove elements in the same manner as in lists, although numbers do not bind elements in the dictionary.

Each element in a dictionary has a key and a value in a (key:value) pair. If you call the key, it returns the corresponding value linked to that key. Lists represents a unique type of dictionary in which the key in each element is a number in a particular order.

Dictionaries are optimized to retrieve data elements using keys. Data values can be of any data type within the dictionary.

Dictionary definition and declaration

You can create a dictionary using the key and placing the values inside the curly braces ({}) separated by a colon. It can also be created using the built-in *dict()* function. You can create an empty dictionary by using the curly braces only.

```
>>>sample_dict= {'name':1, 'dob': 2} # using integer keys
```

```
>>>sample_dict= dict({1: 'name', 2: 'dob'}) # using dict()
```

```
>>>sample_dict = {} # empty dictionary
```

```
>>>sample_dict ={'name': 'dob', 1: [3, 5, 7]} # dictionary with mixed keys
```

```
>>>sample_dict = dict([(1, 'name'), (2, 'dob')]) # dictionary with each of the items as a pair.
```

Adding values to a dictionary

Adding values to a dictionary can be done in various ways. You can add one value at a time to the dictionary by assigning the value to be used to the dictionary key. For example, `sample_dict [key] = value`. You can add more items to the dictionary using the *update()* built-in method. You can also update the dictionary using nested values.

When updating a dictionary with a pre-existing key, the key is updated with the new value. otherwise, a new key is created to store the value.

Example: program to add values to dictionary

```
# create an empty dictionary
```

```
New_dict = {}
```

```
Print ('this is an empty dictionary: ')
```

```
Print (new_dict)
```

```
#adding values one at a time
```

```
New_dict [0]= 'adding'
```

```
New_dict [1]= 'values'
```

```
New_dict [2]= 'to'
```

```
New_dict [3]= 'dictionary'
```

```
Print ('\nupdating dictionary with 4 elements: ')
```

```
Print (new_dict)
```

```
#adding a set of values using a single key
```

```
New_dict['value_set'] =3, 5, 6
```

```
Print ('\nadding set of values to the dictionary: ')
```

```
Print (new_dict)
```

```
#using existing key value to update dictionary
```

```
New_dict[0]= 'update'
```

```
Print ('\nupdate dictionary key value: ')
```

```
Print (new_dict)
```

```
# updating dictionary with nested key elements
```

```
New_dict [7]= {'nested' :{'1' : 'data', '2': 'elements'}}
```

```
Print ('\n adding a nested key to the dictionary: ')
```

Print (new_dict)

```
// output
```

This is an empty dictionary:

```
{}
```

Updating dictionary with 4 elements:

```
{0: 'adding', 1: 'values', 2: 'to', 3: 'dictionary'}
```

Adding set of values to the dictionary:

```
{0: 'adding', 1: 'values', 2: 'to', 3: 'dictionary', 'value_set': (3, 5, 6)}
```

Update dictionary key value:

```
{0: 'update', 1: 'values', 2: 'to', 3: 'dictionary', 'value_set': (3, 5, 6)}
```

```
Adding a nested key to the dictionary:
```

```
{0: 'Update', 1: 'values', 2: 'to', 3: 'dictionary', 'value_set': (3, 5, 6), 7: {'Nested': {'1': 'Data', '2': 'elements'}}}
```

Accessing elements from a dictionary

To access items in the dictionary, you have to call the key name to the dictionary. The key item is put inside the square brackets. You can also use the *get()* function to access the contents of the dictionary.

Example: access dictionary elements

```
#create new dictionary
```

```
New_dict= {1: 'new', 2: 'dictionary', 3: 'elements'}
```

```
#accessing elements using key
```

```
Print ("\naccessing dictionary elements with a key: ")
```

```
Print (new_dict ['1'])
```

```
#accessing elements using get () method
```

```
Print ("\naccessing dictionary elements using the get: ")
```

```
Print (new_dict.get(2))
```

```
// output
```

Accessing dictionary elements with a key:

New

Accessing dictionary elements using the get:

Dictionary

Removing elements from a dictionary

You can delete elements from the dictionary using the *del* keyword. This keyword deletes specific data values from the dictionary or the entire content. Functions like *pop()* and *popitem()* delete specific data elements or arbitrary elements from the dictionary.

You can also delete all elements at once using the *clear()* function. In the case of nested values, you can delete specific nested data elements using the *del* keyword supplied with the specific nested key to be deleted.

Note: using *del dict* deletes the entire dictionary, and if you try to print it, it will generate an error.

Example: deleting elements from a dictionary

```
#creating a dictionary:
```

```
New_dict= {1: 'welcome', 2: 'to', 3: 'python', 4: 'programming',
```

```
'X':{5: 'containers', 6: 'in', 7: 'sequence'},
```

```
'Y':{5: 'dictionary', 6: 'containers'}}}
```

```
Print ("\ndictionary creation: ")
```

```
Print (new_dict)
```

```
#delete elements using the key value
```

```
Del new_dict [4]
```

```
Print ("\ndeleting a specific key element: ")
```

```
Print (new_dict)
```

```
#deleting nested elements
```

```
Del new_dict ['y'][5]
```

```
Print ("\ndeleting nested key element from dictionary: ")
```

```
Print (new_dict)
```

```
# deleting using pop ()
```

```
New_dict.pop (1)
```

```
Print ('\ndeleting items using pop: ')
```

```
Print (new_dict)
```

```
#delete items using arbitrary key value (('y', {6: 'containers'}))
```

```
New_dict.popitem()
```

```
Print ('\npopping out arbitrary key value pair: ')
```

```
Print (new_dict)
```

```
#delete the whole dictionary
```

```
New_dict.clear ()
```

```
Print ('deleting the entire dictionary: ')
```

```
Print (new_dict)
```

```
Dictionary Creation
```

```
{1: 'Welcome', 2: 'to', 3: 'Python', 4: 'Programming', 'x': {5: 'containers', 6: 'in', 7: 'sequence'}, 'y':  
{5: 'dictionary', 6: 'containers'}}
```

```
Deleting a specific key element:
```

```
{1: 'Welcome', 2: 'to', 3: 'Python', 'x': {5: 'containers', 6: 'in', 7: 'sequence'}, 'y': {5: 'dictionary', 6:  
'containers'}}
```

```
Deleting nested key element from dictionary:
```

```
// output
```

```
{1: 'welcome', 2: 'to', 3: 'python', 'x': {5: 'containers', 6: 'in', 7: 'sequence'}, 'y': {6: 'containers'}} deleting items using pop:
```

```
{2: 'to', 3: 'python', 'x': {5: 'containers', 6: 'in', 7: 'sequence'}, 'y': {6: 'containers'}}
```

Popping out arbitrary key value pair:

```
{2: 'to', 3: 'python', 'x': {5: 'containers', 6: 'in', 7: 'sequence'}}
```

Deleting the entire dictionary

```
{}
```

Method	Description
<code>clear()</code>	Removes all the elements from the dictionary
<code>copy()</code>	Returns a copy of the dictionary
<code>fromkeys()</code>	Returns a dictionary with the specified keys and values
<code>get()</code>	Returns the value of the specified key
<code>items()</code>	Returns a list containing a tuple for each key value pair
<code>keys()</code>	Returns a list containing the dictionary's keys
<code>pop()</code>	Removes the element with the specified key
<code>popitem()</code>	Removes the last inserted key-value pair
<code>setdefault()</code>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<code>update()</code>	Updates the dictionary with the specified key-value pairs
<code>values()</code>	Returns a list of all the values in the dictionary

Dictionary built-in methods

chapter summary

Data sequences enable you to develop a series of objects using various built-in datatypes. A sequence ensures that all data objects and values are organized in an efficient manner. In the chapter, you learned about the different components of a sequence, which includes lists, strings, tuples, and buffers among other elements. You also learned:

- How to use lists in the implementation of python programs
- How to develop objects using various built-in data types
- Byte objects and its use of ascii characters, including how byte arrays work to return an array of a particular byte size
- carry out various operations carried out in python
- Use concatenation operator to join strings
- How to create sets and use various functions in sets
- How to use boolean expressions to return values as true or false
- Performing calculations using booleans operators
- How to use bitwise operators
- How to retrieve data from the dictionary or add values to the dictionary

In the next chapter you will learn about the math module in python, as well as the random module and seeding function.

cHAPTER FoUR:

MATH FUNcTIoNS IN PYTHoN

Introduction

Python programming has a special predefined math module. The module library provides a variety of functions and properties or constants that help you perform mathematical tasks. Library properties and functions such as logarithmic functions, representation functions, trigonometric functions, and angle conversion functions among others have static objects.

The module defines two mathematical constants: pi and euler's number. Python properties are accessed through *math.pi* while the functions are accessed through *math.abs (number)*.

Pi (n) is a constant defined as the ratio between a circle's circumference and diameter. It has a value of 3.141592653589793. Euler's number (e) is another mathematical constant defined in the math's module. It is the base of a natural logarithm with a value of 2.718281828459045.

```
>>> import math
```

```
>>> math.pi
```

```
3.141592653589793
```

```
>>> math.e
```

```
2.718281828459045
```

other constants include inf and nan. *Math.inf* is a math property that returns a positive infinity. It can also be used to return a negative infinity. *Math.nan* returns not a number as the output.

To use the math module, you have to first import it to a python program. Math modules have functions to calculate trigonometric ratios in a specific angle. Functions like sin, cos, and tan pass angles in radians as the function arguments. We normally present angles in degree form, but the math module represents angles in the form of radians. Therefore, the module has *degrees()* and *radians()* conversion functions used to convert degrees into radians and vice versa.

Example:

```
>>> math.radians(20) 0.3490658503988659
```

```
>>> math.degrees (math.e/6) 25.957679382967953
```

```
>>> math.radians (45)
```

```
0.7853981633974483
```

```
>>> math.degrees (math.pi/4) 45.0
```

```
>>> math.cos (0.526890)
```

```
0.8643751040967125
```

```
>>> math.tan (0.583589)
```

```
0.6603101237742569
```

Python math functions

A python interpreter has a wide range of built-in functions for performing various tasks. For example, the *print ()* function is used to print a particular object in a standard output device or use a text stream file.

The different classes of math functions supported by python library include:

Numbers and numeric representation

Sr.No.	Function & Description
1	ceil(x) Return the Ceiling value. It is the smallest integer, greater or equal to the number x.
2	copysign(x, y) It returns the number x and copy the sign of y to x.
3	fabs(x) Returns the absolute value of x.
4	factorial(x) Returns factorial of x. where $x \geq 0$
5	floor(x) Return the Floor value. It is the largest integer, less or equal to the number x.
6	fsum(iterable) Find sum of the elements in an iterable object

Some of the functions used to represent numbers include

7	gcd(x, y) Returns the Greatest Common Divisor of x and y
8	isfinite(x) Checks whether x is neither an infinity nor nan.
9	isinf(x) Checks whether x is infinity
10	isnan(x) Checks whether x is not a number.
11	remainder(x, y) Find remainder after dividing x by y.

Power and logarithmic functions

Sr.No.	Function & Description
1	pow(x, y) Return the x to the power y value.
2	sqrt(x) Finds the square root of x
3	exp(x) Finds xe, where e = 2.718281
4	log(x[, base]) Returns the Log of x, where base is given. The default base is e
5	log2(x) Returns the Log of x, where base is 2
6	log10(x) Returns the Log of x, where base is 10

These functions calculate power and logarithmic tasks. They include:

Example:

The *math.log()* function returns the logarithm of a specific number. Natural logarithms are always calculated to the base e.

```
>>> math.log (5)
```

```
1.6094379124341003
```

```
>>> math.log10 (5)
```

```
0.6989700043360189
```

The *math.log10()* returns the logarithm to the base of 10 for a given number.

```
>>> math.sqrt(81)
```

```
9.0
```

Trigonometric and angular conversion functions

Sr.No.	Function & Description
1	sin(x) Return the sine of x in radians
2	cos(x) Return the cosine of x in radians
3	tan(x) Return the tangent of x in radians
4	asin(x) This is the inverse operation of the sine, there are acos, atan also.
5	degrees(x) Convert angle x from radian to degrees
6	radians(x) Convert angle x from degrees to radian

These functions calculate different trigonometric functions.

Random module in python

```
>>> import random
```

```
>>> random.random()
```

```
0.0952389196671356
```

Functions in the random module use a pseudo-random number generator tool to generate floating point numbers between 0.0 and 1.0.

Random() is the basic function in the module and almost all other functions in the random module depend on it. The *random()* returns a floating point number in the range of 0.0 to 1.0.

The *random.random()* function doesn't need arguments or no arguments. To generate random integers, you can use:

- `Randint ()`
- `Randrange ()`
- `choice()`
- `Shuffle()`
- `Sample()`

Randint ()

Randint() is an inbuilt function in the random module, which provides access to various functions used in generating random numbers. The *randint()* function should have both start and end arguments, which must be integer type values. The syntax for this function is *randint(start,end)*.

Example:

```
>>> random.randint(2, 10)
```

```
6
```

```
>>> random.randint(25, 70)
```

70

```
>>> random.randint (30, 50)
```

49

Random.randint() returns an integer value within a range of specified integer numbers.

Errors and exceptions

Randint () returns two types of errors: *valueerror* and *typeerror*.

Valueerror: this is an error returned after passing floating point values as arguments to the function.

TypeError: the error occurs when any parameter is passed as argument to the function that is not an integer value.

Example: program to print random numbers

```
# generate random numbers of range 10 to 40
```

```
Rm1= random.randint (10, 40)
```

```
Print ('random numbers in the range between 10 and 40 is %d' %(rm1))
```

```
# random number for a range of negative values
```

```
Rm2= random.randint (-5, -1)
```

```
Print ('random numbers between -5 and -1 is %d' %(rm2))
```

```
// output
```

Random numbers in the range between 10 and 40 is 17 random numbers between -5 and -1 is -3

Example: program to determine a lucky draw

The *randint()* function can be used in a lucky draw simulation to select a random number. The user has a chance to guess the winning number and if the guess is right, he wins. otherwise, he will lose the competition.

```
Import random
```

```
From random import randint
```

```
Def generator(): # function to generate a random number on each execution
```

```
Return randint(1,10)
```

```
Def rand_guess(): # function to return true or false based on user lucky draw
```

```
Random_number = generator() # call to generator()
```

```
Guess_left = 3 # define the number of guesses the user gets
```

```
Flag = 0 # define flag variable to check the win condition
```

```
While guess_left > 0:
```

```
Guess = int(input("pick your number to enter the lucky draw\n"))
```

```
If guess == random_number:
```

```
Flag = 1 # set flag as 1 if user guesses correctly. Then break loop
```

Break

Else:

```
Print("wrong guess!!")
```

```
Guess_left -= 1 # decrease number of remaining guesses by 1
```

```
If flag is 1: # if win condition is satisfied, then rand_guess returns true
```

```
Return true
```

```
Else:
```

```
Return false
```

```
# driver code
```

```
If name == 'main':
```

```
If rand_guess() is true:
```

```
Print("congrats!! You win.")
```

```
Else:
```

```
Print("sorry, you lost!")
```

```
// output
```

```
Pick your number to enter the lucky draw 8
```

Wrong guess!!

Pick your number to enter the lucky draw 9

Wrong guess!!

Pick your number to enter the lucky draw 0

congrats!! You win.

Randrange () method

The *random.randrange ()* function returns a random element from the range given. The random element is selected from the start, stop, and step arguments. The start value is always 0 by default while the step value is 1 by default.

The step value is optional and it indicates the difference between each of the numbers in the sequence.

```
>>> import random
```

```
>>> random.randrange (2, 8)
```

```
6
```

```
>>> random.randrange (1, 15, 10)
```

```
11
```

```
>>> random.randrange (0, 80, 20)
```

```
20
```

choice() method

Random.choice () is used to return a random value selected from an empty sequence. When you use an empty sequence as argument, an *indexerror* is generated.

```
>>> import random
```

```
>>> random.choice([10, 20, 25, 32, 38,42])
```

```
32
```

```
>>> random.choice([5, 7, 8, 9, 11, 13])
```

```
13
```

```
>>> random.choice('user')
```

```
'R'
```

Example 2: randomly select a city from a list

```
Import random
```

```
city= ['nairobi', 'mombasa', 'kisumu', 'eldoret', 'nakuru']
```

```
Print ('select your favorite city from the list: ', random.choice(city))
```

```
Print ('select your favorite city from the list: ', random.choice(city))
```

```
// output
```

```
Select your favorite city from the list: nakuru select your favorite city from  
the list: mombasa
```

Shuffle() method

The *random.shuffle()* function randomly reorder elements within a list.

```
>>> import random
```

```
>>> values= [12, 32, 23, 30, 45, 32, 48, 38, 50]
```

```
>>> random.shuffle(values)
```

```
>>> values
```

```
[32, 32, 30, 48, 23, 38, 12, 50, 45]
```

```
>>> random.shuffle(values)
```

```
>>> values
```

```
[23, 45, 48, 50, 12, 38, 32, 30, 32]
```

Sample() method

The *random.sample(population, k)* method is used when you want to select several random

Elements from a population. This method returns a list of unique elements selected from a population. The size of the population, *k*, depends on the total number of elements you want to select. A population can be a set of numbers, a list, or a sequence.

Example:

```
>>> import random
```

```
>>> new_list= [2,5,7,8,10,12]
```

```
>>> print ('random.sample() ', random.sample(new_list,3))
```

```
Random.sample() [8, 7, 5]
```

Example: program to play dice

```
#dice game
```

```
Import random
```

```
Firstplayer= 'moses'
```

```
Secondplayer= 'joseph'
```

```
Mosesscore=0
```

```
Josephscore=0
```

```
Firstdice= [2, 3, 5, 6, 7, 8]
```

```
Seconddice= [2, 3, 5, 6, 7, 8]
```

```
Def dicegame(): #players will roll the dice using the shuffle method
```

```
For x in range(4):
```

```
Random.shuffle(firstdice)
```

```
Random.shuffle(seconddice)
```

```
choose_first_number=random.choice(firstdice)#picking a single random  
number
```

```
choose_second_number=random.choice(seconddice)
```

```
Return choose_first_number + choose_second_number
```

```
Print ('this is a dice game using the random module\n')
```

```
Def dicegame(): #players will roll the dice using the shuffle method.
```

```
For x in range(4):
```

```
Random.shuffle(firstdice)
```

```
Random.shuffle(seconddice)
```

```
choose_first_number=random.choice(firstdice)#picking a single random  
number
```

```
choose_second_number=random.choice(seconddice)
```

```
Return choose_first_number + choose_second_number
```

```
Print ('this is a dice game using the random module\n')
```

```
For x in range (3):
```

```
Moses_toss_number=random.randint(1,70)
```

```
Joseph_toss_number=random.randrange(1, 71, 1)
```

```
If(moses_toss_number>joseph_toss_number):
```

```
Print('moses won the toss')
```

```
Mosesscore=dicegame()
```

```
Josephscore=dicegame()
```

```
Else:
```

```
Print('joseph won the toss')
```

```
Mosesscore=dicegame()
```

```
Josephscore=dicegame()
```

```
If(mosesscore>josephscore):
```

```
Print ('moses won the game. Moses score is:', mosesscore, and 'joseph's  
score is:',
```

```
Josephscore, '\n')
```

```
Else:
```

```
Print('joseph won the game. Joseph's score is:', josephscore, and 'moses  
score
```

```
Is:', mosesscore, '\n')
```

```
// output
```

```
Joseph won the toss
```

```
Moses won the game. Moses score is: 9 and joseph's score is: 6
```

```
Joseph won the toss
```

```
Moses won the game. Moses score is: 11 and joseph's score is: 9
```

Moses won the toss

Joseph won the game. Joseph's score is: 12 and moses score is: 6

The seed () function in python

Not all the random numbers generated in python are fully random. They are pseudo-random generated using a pseudorandom number generator (prng). Prng is an algorithm that not only generates a seemingly random number but also is used in reproducing data.

Prng uses a software tool to generate random numbers. This tool works by getting a random number known as the seed and then using an algorithm that generates a pseudo-random sequence of bits based on the seed.

The pseudo-random generator relies on the previous generated value to perform its operation on the value. However, when you use the generator for the first time, there is no previous value stored in the memory. Therefore, the seeding pseudo-random generator generates the first previous value. The seeding value is equivalent to the generated value in any particular random generator number. If you pass the same parameters to the seed twice, it will generate the same sequence of numbers twice.

The *seed()* method initiates the pseudorandom number generator while the random module uses the seeding value as the base when generating random numbers. If the seeding value is unavailable, the current date will be used. If the same value is passed to the seed before generating a random number, then it will generate the same number. To generate a seed, use *seed([n])* where n is the random number. If omitted, then the system time will be automatically used in generating the next random number

The *seed ()* function cannot be accessed directly; therefore, you have to import the random module and call the function with a static random object.

Example:

```
Import random
```

```
From random import randint
```

```
Random.seed (40)
```

```
Print ('num1 -', random.randint(20, 45))
```

```
Random.seed (20)
```

```
Print ('num2 -', random.randint (10, 30))
```

```
// output num1 - 34
```

```
Num2 - 14
```

How seed() works

The *seed()* method is a random module that initializes a pseudorandom number generator. It saves the state of a random function to enable multiple execution of random numbers on the code on either the same machine or different machines.

The seed value indicates the previous random value generated by random generator. If no previous value, the current system time will be used. call the seed first before calling the random number generator.

Example 2:

```
Import random
```

```
Random.seed (4)
```

```
Print (random.randint (1, 200))
```

Print (random.randint (1, 200))

Print (random.randint (1,200))

Print (random.randint (1,200))

/ output 61

78

27

185

Uses of random.seed()

1. It generates encryption key using the pseudorandom generator. An encryption key is an important tool in maintaining computer security. Seed() generates secret keys or codes used in protecting data from unauthorized access over the internet.
2. Random testing of numbers making it easy to optimize the codes. The codes output depends on the input data. The seed produces the same random number if you pass the same arguments to the seed () function.
3. It simplifies algorithm testing process.

cryptographically secure random generator in python

Random numbers and any data generated using the random module is not secure. You can implement cryptography to securely generate pseudo-random numbers. The pseudo random number generator will have properties that will make data more secure.

A cryptographically secure random number generator returns a random byte. This random byte generated depends on os random sources. A secure random number can be achieved through:

- Use of secret module to ensure secure generation of random data
- Using the `os.urandom ()` function.
- Using the `random.systemrandom` class example:

```
>>> import random
```

```
>>> import secrets
```

```
>>> num1= random.systemrandom().random ()
```

```
>>> print ('a cryptographic secure number is', num1)
```

```
>>> print ('the secure bytes token is', secrets.token_bytes (16))
```

```
// output
```

A cryptographic secure number is 0.922173569673084

The secure bytes token is b'\xd0r\xcd\xe5k\xacf\x13\xf3\x80\xae"9oqw'

Get and set random generator state

A random module uses *getstate()* and *setstate ()* functions to determine the current internal state of a random number generator. This information is essential in generating a sequence of data or generating the same random numbers.

Random.getstate()

The *getstate* function records the current internal state of a random number generator. The recorded state is then passed to the *setstate* function to restore the state to the current state. If you change the state to its previous one, you will get the same random data.

Random.setstate ()

This function restores the internal state to its previous state. It restores random generator to state object that is, same state again. To obtain the state object, a call to *getstate()* function is made.

If you restore the previous state, it will enable you to reproduce or obtain the same random values again. If you use a different random function or pass different parameter lists, then this alters the state of the random generator.

Example:

```
Import random
```

```
New_list= [2,5,7,8,10,12]
```

```
Print ("sample list", random.sample (new_list, k=4))
```

```
Status= random.getstate() #storing the current state in status object
```

```
Print ("another sample list", random.sample(new_list, 4))
```

```
Random.setstate (status)
```

```
Print ("printing restored state", random.sample (new_list, 4))
```

```
Random.setstate (status)
```

```
Print ("printing another restored state", random.sample (new_list, 4))
```

```
// output
```

```
Sample list [7, 8, 5, 10]
```

```
Another sample list [2, 10, 12, 8]
```

```
Printing restored state [2, 10, 12, 8]
```

```
Printing another restored state [2, 10, 12, 8]
```

Numpy.random (using prngs for arrays)

Prng stands for pseudo-random number generator. It uses the python random module to help generate data or scalar random numbers. The *numpy.random* method enables you to generate an array of random numbers. The numpy

package has multiple functions which generates n- dimensional array of random numbers.

The numpy package includes:

- *Numpy.random.rand ()* which generates an n-dimensional array of floating point random numbers in the range of 0.0 and 1.0.
- *Numpy.random.uniform* which generates an n-dimensional array of floating point numbers within a given range.

Example:

```
Import numpy
```

```
Array_list =[10, 20, 40, 30, 50, 20, 50]
```

```
Random_choice = numpy.random.choice (array_list, k=1)
```

```
Print ('selecting random choice for single array:', random_choice)
```

```
Multiple_array = numpy.random.choice (array_list, k=3, replace=false)
```

```
Print ('selecting multiple random choice array without replacement:',  
multiple_array)
```

```
multiple_array = numpy.random.choice (array_list, k=3, replace=True)
```

```
print ('selecting multiple random choice array with replacement:', multiple_array)
```

```
// output
```

```
Selecting random choice for single array: [10]
```

Selecting multiple random choices array without any replacement: [20 20 10]
selecting multiple random choices array with a replacement: [10 50 50]

Generating random numbers with unique ids

The uuid (universally unique identifier) module in python provides you with immutable uuid objects to enable you generate a unique id. The uuid package consists of various functions generate universal ids. It allows you to generate up to 128 bit long unique random id and a cryptographic safe.

These unique ids help in identifying users, documents, resources, or any information exchanged within a computing system.

To use the uuid object, you have to import the module in your development framework.

```
import uuid
```

```
getid= uuid.uuid4()
```

```
print ('Your secret id is', getid)
```

// output

Your secret id is 3430e137-4f94-4528-9b64-222d77bd6cd9

chapter summary

Python consists of a list of libraries commonly known as modules for use with other programs. The math module allow you to perform various tasks. In this chapter, you learned how to:

- Import the math module into your program
- Refer functions in the module as module_name.function_name. The dot (.) operator means part of.

- Use the math module in the standard library
- Use python constants
- Use various math functions to perform various mathematical calculations. Some of these functions include: numbers and numeric representation, power and logarithmic functions, and trigonometric and angular conversion functions.
- Import the random module and import *randint()* from the random module
- Use python random module to generate random numbers and how players can play dice game using both shuffle and choice methods.
- Use the seed algorithm to generate a pseudo random number. It uses the seeding value as the base when generating random numbers. The seeding method relies on the previous state of the random number and if no previous number recorded, it uses the current date system.
- Use and apply the *seed()* and *random.seed()* functions
- cryptographically secure random generator in python. This ensures you're able to secure your own data and information being exchanged from one machine to another.
- Use the *getstate()* and *setstate()* random generator functions to determine the current internal state of a random number.
- Use the numpy python package when working with arrays.
- Generate unique ids using the uuid module.

In the next chapter you will learn about the date and time module.

CHAPTER FIVE:

DATE AND TIME FUNCTIONS

Introduction

```
>>> import datetime
```

```
>>> date = datetime.datetime.now()
```

```
>>> print (date)
```

Dates in python programming do not have a data type on their own. To use the date's objects or the date function, you have to import the *datetime* module into your program.

// output

2020-01-05 05:50:08.673261

The *datetime* method returns the year, month, day, and the system time during execution displayed in hours, minutes, seconds, and microsecond. The *datetime* module has various functions that return information on the date object.

In the above example, we defined *datetime* class inside the *datetime* module and then used the

Now() function to create an object with the current date and time.

Python keeps track of date and time in different ways. The *datetime* python module helps in tracking dates and time as well as converting the date formats. Time formats are expressed as floating point numbers in units of seconds.

The *time* module has multiple functions for working with the time format and conversion between representations.

Date and time manipulation

Date, time, and datetime in python are classes that provide multiple functions to handle date, time, time intervals, and date. Both date and datetime are python objects, and once you manipulate them, you're actually manipulating the object itself, not its string or timestamp.

To manipulate date and time, you have to import the datetime module in your python shell. The datetime class is classified into:

- Date class: this manipulates dates in terms of month, day, and year.
- Time class: it handles time formats in the form of hours, minutes, seconds, and microsecond. Time is independent of the day.
- Timedate class: it handles both date and time formats
- Timedelta class: it is used in the manipulation of dates and shows time duration.
- Tzinfo class: it handles time zones.

Using the date class

Before running datetime code, you have to import the module from the python library, then make calls to specific date and time functions. The date class allows you to instantiate the date objects which represent the year, month, and day. For example,

```
>>> import datetime
```

The line above tells python interpreter to import date class from the datetime module. once you imported the date class, the next step is to create an instance of the date object.

To get the current date, you can run the following code:

```
>>>import datetime
```

```
>>>today_date=datetime.date.today()
```

```
>>>print (today_date)
```

```
// output 2020-01-05
```

The *today()* method is part of date class functions and it allows you to use the date object to extract the current system date. You can also use *date.today()* method to return the current date.

You can also use the *date()* method, which acts as a constructor of the date class. In this case, the

Date () constructor accepts three arguments: year, month, and day. That is, `today_date= datetime.date (2020, 01, 05)`.

```
from datetime import date
```

```
d= date (2020, 01, 05)
```

```
print (d)
```

You can also import the date class from the datetime object. This can be done through the following code:

Using timestamp to retrieve date

A date object can also be created from a timestamp. A unix timestamp is used to determine seconds in a particular date range. You can easily convert the timestamp format to date form using *fromtimestamp ()* method.

Example:

```
from datetime import date
```

```
d = date.fromtimestamp(1582145574)
```

```
print("Date ", d)
```

```
// output
```

Date 2020-02-19

Using the time class

A time object can be instantiated from a time class. From the datetime object, import time class. To get the current time, assign the time class to *datetime.now()* method.

Example: using time class to get the current local time.

```
from datetime import time
```

```
tt=datetime.time (datetime.now())
```

```
print ('The Current time is:', tt)
```

```
// output
```

The current time is: 17:27:08, 384466

The tt variable is used to store the time value. When you call the *datetime.now()* method, it returns the current system time.

Example 2:

From datetime import time

T=time()

Print (t)

T1= time(11, 34, 10)

Print (t1)

```
// output 00:00:00
```

```
11:34:10
```

Datetime class

The datetime module has a datetime class with properties of both the date and time classes. The

Datetime() class accepts time and timezone arguments.

```
import datetime
```

```
date = datetime.datetime.now()
```

```
print (date)
```

Strftime () method

The datetime object consists of formatting functions used to format the date objects into a string. The *strftime()* method is used to format a readable string and accepts a single parameter (format) to specify how the date string is returned

Example: displaying the date based on the month

```
import datetime
```

```
d=datetime. datetime(2020, 1, 3)
```

```
print (d.strftime("%b"))
```

```
// output jan
```

The *strftime()* method uses different control codes to produce different outputs. The different control codes represents different date formats like %y/%y for years, %a/%a for weekdays, and

%b/%b for months. If you use %y, it will print the full year (i.e. 2021), but if you use a lowercase %y, it will print out (21) instead.

Example:

```
import datetime
```

```
d=datetime.datetime(2019)
```

```
print (d.strftime("%y"))
```

// output 19

Example 2:

```
from datetime import time
```

```
now= datetime.now()
```

```
print (now.strftime('%A, %d %B, %Y'))
```

// output

Sunday, 5 january, 2020

Some commonly used formats for the date class include:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100
%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%

Using timedelta methods

A `timedelta` object is used to determine any special day, date, or time. It can predict past and future periods. It doesn't print date or time, only calculates them.

To use the `timedelta` class, you have to import it from the `datetime` module.

```
>>> from datetime import timedelta
```

```
>>> print (timedelta(days=20, hours=4, minutes=45))
```

```
20 days, 4:45:00
```

```
>>> # display today's date
```

```
>>> print ('today is on:' + str (datetime.now()))
```

```
Today is on: 2020-01-05 24:41:15.528684
```

```
>>> # retrieve future date: one year from now
```

```
>>> print ('a year from now will be:' + str (datetime.now() +timedelta(days=365))
```

```
A year from now will be: 2021-01-05 24:42:20. 407012
```

```
>>> # calculate future date from current time
```

```
>>> print ('in one week and 2 days the date will be:' +str(datetime.now() +  
timedelta(weeks=1,
```

```
Days=2) ))
```

```
In one week and 2 days the date will be: 2021-01-14 24:44:18.582465
```

You can make the `timedelta()` method more complex by adding arithmetic calculations to it. For example, use `timedelta()` to determine how many days passed after the new year or calculate the number of days within a particular date range.

Example: calculate how many days after the new year

```
From datetime import timedelta
```

```
#obtain today's date
```

```
Today=date.today()
```

```
New_year= date.year,1,1) #first day of the year
```

```
If new_year<today:
```

```
Print ('the new year has already passed with %d days' %((today-new_year).days))
```

```
// output
```

The new year has already passed with 5 days

Python strptime() method

The *strptime()* method is a string for changing the datetime format. The method creates the datetime object from any date and time string.

```
From datetime import datetime date-string = "3 august, 2019"
```

```
Print("the date string is:", date-string)
```

```
Dateobject = datetime.strptime(date-string, "%d %b, %y") print("the date object is:", dateobject)
```

```
// output
```

The date string is: 3 august, 2019

The date object is: 2019-08-3 00:00:00

The above string represents date and time format and the format code equivalent in the form of

%d, %b, and %y to represent day, month, and year in full names.

Working with different time zones

```
from datetime import datetime
import pytz
local = datetime.now()
print("Local:", local.strftime("%m/%d/%Y, %H:%M:%S"))

tz_NY = pytz.timezone('America/New_York')
datetime_NY = datetime.now(tz_NY)
print("NY:", datetime_NY.strftime("%m/%d/%Y, %H:%M:%S"))

tz_London = pytz.timezone('Europe/London')
datetime_London = datetime.now(tz_London)
print("London:", datetime_London.strftime("%m/%d/%Y, %H:%M:%S"))
```

If you're working on a program that needs to display different time zones, you have to call the pytz module into your program. The pytz module allows you to display the date or time based on a particular time zone.

// output

```
Local time: 2020-1-5 15:20:44.260462
America/New_York time: 2020-1-5 15:20:44.260462
Europe/London time: 2020-1-5 15:20:44.260462
```

The datetime objects created include the datetime_ny object and datetime_london object. These objects obtain the date and time of the respective areas based on the time zone.

Python time module

The python time module provides functions used for manipulating time and converting time representations. You can use objects, numbers, and strings to represent the time class. To use these functions, you have to import the module in your program.

```
>>> import time
```

At the end of the chapter, you will be able to understand how to create time using epochs and time zones and represent time using floating points or tuples.

Python time using floating point numbers

one way to manage time in python is to use floating point numbers that indicate the number of seconds that have passed since the epoch era. That is, a specific starting point. The epoch is very essential when implementing time in python project.

The epoch era

Epoch is used to manage time using floating point numbers. It represents an elapsed time since the starting of a particular era. An era is a fixed point in time which records a particular series of years.

The starting point identifies/ measure the passage of time of an epoch. For example, if epoch is defined on january 1, 1970 utc at midnight (standard date format for windows and unix systems), then since epoch, the midnight date on january 2, 1970 utc is represented as 86400 seconds.

This is simply so because a day has 24 hours, 60 minutes in an hour and 60 seconds in a minute. Therefore, january 2, 1970 is a single day after epoch:

```
>>>24*60*60 86400
```

Time can also be represented before epoch by indicating the number of seconds to be negative. Therefore, the epoch on midnight of december 31, 1969 utc is -86400 seconds.

Though january 1, 1970 utc is the standard epoch, there are other epochs used in computing. Different filesystems, operating systems and apis use different epochs. For example, unix operating system use january 1, 1970 epoch while win32 api use january 1, 1601 epoch.

```
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=0)
```

To know your operating system epoch, use *time.gmtime()* method.

Python time in seconds

```
>>> from time import time
```

```
>>> time()
```

```
1578308718.2490938
```

The *time.time()* python function will show the number of seconds that have passed since an epoch.

Time is measured in seconds in order to:

- Determine the difference between the two floating points in time.
- Use floats for data transfer. Floating point numbers are serializable and can easily be stored to enable data transfer from one end to the other. For example, if you want to

Have time represented in form of a string, pass the number of seconds from the

Time() function into the *time.ctime()* function.

Example: time conversion in seconds into a string

```
>>> from time import time, ctime
```

```
>>> t_seconds=time()
```

```
>>> ctime(t_seconds)
```

'Mon jan 6 14:15:50 2020'

The above code records the current time in seconds to the variable 't_seconds' and then passes the variable t_seconds as the argument to the *ctime()* function.

The *ctime()* function returns a string in form of a timestamp which has the following attributes. current day of the week: mon

The month: jan day of the month: 6

current time using 24-hour clock notation: 14:15:50 (hours, minutes, seconds) the year: 2020

The timestamp above is based on your current time zone. If you run the same code in a computer in different states, you will get a different timestamp. Therefore, the timestamp returned by the call to *ctime()* is based on your geographical location.

The representation of time based on your physical location is known as local time and the concept of time representation is known as time zones.

Time zones

A time zone allows us to conform to a standardized time within our region in the world. A time zone is defined using the coordinated universal time (utc) and takes into account daylight savings time.

Utc and time zones

Utc is a standard for synchronizing or coordinating the world's time keeping. It acts as a transcendent standard that determines which time zone we are in. Utc is measured using astronomical time, which determines the earth's rotation and its atomic clock.

Different time zones are determined by their offset from the utc. For example, the north and south america use the central time zone (ct) which is usually behind the utc time by either five or six hours. Therefore, the notation utc -5:00 or utc - 6:00 is used to represent the two states.

If you're in australia or sydney, the eastern time zone (aet) is usually ahead of utc zone with 10 or 11 hours. Therefore, the utc notation is utc + 10:00 or utc + 11:00.

The reason behind why some parts of the world are either six hours behind or eleven hours ahead of the utc is because they observe daylight saving time.

Daylight savings time

Summer months have longer days than nights while winter months have longer nights than days in some parts of the world. This is because some places observe daylight savings time (dst) during the spring and summer periods to make better use of these long daylight hours.

In areas where dst is observed, the clock always jumps one hour ahead at the start of spring. Then, during fall, the clock will be reset to its standard time.

When representing time as a timestamp, you have to factor in the dst. The *ctime ()* function accounts for the dst.

Sr.No.	Attribute with Description
1	time.timezone Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).
2	time.tzname Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

Every function in the time module has the following attributes.

Working with python time using data structures

In the previous subtopics, we have used numbers, epochs, utc, and strings to represent python time. There are other ways you can incorporate the python time module. This includes the use of primitive data structures like tuples to represent time.

Using tuples to represent python time

Tuple data structures are very important in the representation of time modules. Tuples allow you to easily represent time data. It abstracts some data items not

needed to make it more readable. When you represent time using tuples, every tuple element will represent a specific element of the time.

Some of these elements include:

- Date: represents the format (year, month represented as an integer value from 1 to 12, the day of the month).
- Time: in the form of hours (24 hour format), minutes, and seconds
- Day of the week: this is an integer value to represent any day of the week where 0 corresponds to monday and 6 represent sunday.
- Day of the year
- Daylight savings time which is an integer value with
 - 1 to indicate daylight saving time
 - 0 to indicate the standard time
 - -1 unknown value

Let's look at an example on how to represent time module using tuples.

#time representation with tuple

```
>>> from time import time, ctime
```

```
>>> t_seconds=time ()
```

```
>>> t_seconds
```

```
1578321168.5871239
```

```
>>> ctime(t_seconds)
```

```
'Mon Jan 6 17:32:48 2020'
```

```
>>> time_tuple= (2020, 1, 6, 17, 32, 48, 50, 57, 0)
```

The `t_seconds` and `time_tuple` show the same time, but the tuple indicates a more readable interface to deal with components.

Using python time as an object

When you use tuples, it just looks like a list of numbers, albeit more organized. The *struct_time*

Solves this tuple construct problem by calling the *namedtuple* from python collection library.

This makes it easy to associate the tuple's sequence numbers using useful identifiers.

```
>>> from time import struct_time
```

```
>>> time_tuple= (2020, 1, 6, 17, 32, 48, 50, 57, 0)
```

```
>>> time_obj= struct_time (time_tuple)
```

```
>>> time_obj
```

```
Time.struct_time(tm_year=2020, tm_mon=1, tm_mday=6, tm_hour=17,  
tm_min=32, tm_sec=48,
```

```
Tm_wday=50, tm_yday=57, tm_isdst=0)
```

In python programming, no *struct* data type is required since everything is represented as objects unlike other object oriented programs, where keyword *struct* and *object* are in opposition to each other. The *struct_time* is derived from the c programming language where the keyword *struct* is the data type.

When python time module is implemented in c language, the struct type is called directly into the program by including the *time.h* header file.

Example:

```
#accessing elements of time_obj using the attributes names
```

```
>>> current_day_of_year= time_obj.tm_yday
```

```
>>> current_day_of_year
```

```
57
```

```
>>> current_day_of_month= time_obj.tm_mday
```

```
>>> current_day_of_month
```

```
6
```

converting python time from seconds to an object

So far, we have seen the various ways you can work with time. In this section, you will learn how to convert different time data types. converting different time data types depends on whether the current time is a local time or utc.

coordinated universal time (utc) conversion

Epochs rely on utc for time presentation, not on time zone. Hence, seconds pass since epochs are not variables based on your geographical location.

When using a struct_time object, the time module may take the time zone into account or not. There are two ways to convert the floating point numbers representing time in seconds: utc and local time.

```
>>> import time
```

```
>>> time.gmtime(0)
```

```
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=0)
```

To convert floating point seconds from python time to utc based struct_time, use *gmtime()*. The *gmtime()* function is a part of the python time module. When you make a call to this function, it will return the system epoch as we have seen earlier in our epoch discussion.

The *gmtime()* function converts the elapsed seconds to a struct_time in utc. The argument 0 used as number of seconds finds the epoch itself.

If you call the functions with no arguments then it will return the current time in utc.

```
>>> import time
```

```
>>> time.gmtime()
```

```
time.struct_time(tm_year=2020, tm_mon=1, tm_mday=6, tm_hour=16, tm_min=12, tm_sec=7,
tm_wday=0, tm_yday=6, tm_isdst=0)
```

When using utc in python projects, you don't have to worry about dst, the time zone, or even location information.

Local time

In most python programs, you will use the local time rather than the utc. There are various functions you can use to obtain local time from the number of seconds since the epoch era.

The *localtime()* works the same way as the *gmtime()* function and accepts options *secs* arguments to enable creation of struct_time using the local time zone.

```
>>> import time
```

```
>>> time.time()
```

```
1578328017.015449
```

```
>>> time.localtime(1578328017.015449)
```

```
time.struct_time(tm_year=2020, tm_mon=1, tm_mday=6, tm_hour=19, tm_min=26, tm_sec=57,
tm_wday=0, tm_yday=6, tm_isdst=0)
```

The '*tm_isdst=0*' since dst is very important when calculating local time object. The value can be set to either 0 or 1 based on whether the dst of a given time is applicable or not.

For example, in the us, daylight saving time is applicable nine days later on march 10, instead of on march 1 where dst is not applicable. Therefore, to test whether the dst flag will change correctly, you have to add nine days into the number of seconds within a day (86400×9) to obtain the *secs* argument.

```
>>> new_localtime_sec = 1578328017.015449 + (86400*9)
```

```
>>> time.localtime (new_localtime_sec)
```

```
>>> time.struct_time(tm_year=2020, tm_mon=1, tm_mday=15, tm_hour=19,
tm_min=26, tm_sec=57, tm_wday=2, tm_yday=15, tm_isdst=0)
```

When you run the code, you will notice that the *tm_mday* has jumped ahead to 15, the *tm_wday*

And *tm_yday* have also changed due to sunlight saving time.

other attributes used in determining time zone in *struct_time* include *tm_zone* and *tm_gmoff*. Example:

```
>>> import time
```

```
>>> current_local_time= time.localtime()
```

```
>>> current_local_time.tm_zone
```

```
'E. Africa standard time'
```

The *localtime ()* returns the east africa standard time zone. The *tm_zone* determines the local time zone of your system.

Example 2:

```
>>> import time
```

```
>>> current_local_time= time.localtime()
```

```
>>> current_local_time.tm_gmtoff
```

```
10800
```

```
>>> current_local_time.tm_isdst
```

```
0
```

The current local time is 10800 seconds behind the gmt. Greenwich mean time (gmt) is a time zone without utc offset (utc± 00:00).

You can convert the 10800seconds to gmt by dividing with seconds per hour (3600). Thus the local time gmt +03:00 which corresponds to utc +03:00.

If you ignore the secs argument when calling the *localtime()*, it will return the current local time in struct_time.

converting local time object to seconds

To convert the local time object to seconds, *mktime()* is used. *Mktime()* accepts parameter t which is in the form of a normal tuple with nine elements or a struct_time object.

```
>>> import time
```

```
>>> time_tuple= (2020, 1, 6, 17, 32, 48, 50, 57, 0)
```

```
>>> time.mktime(time_tuple)
```

```
1578321168.0
```

```
>>> t_struct = time.struct_time(time_tuple)
```

```
>>> time.mktime(t_struct)
```

```
1578321168.0
```

converting time object to a string

You can also use strings to manipulate the python time instead of using tuples. String representation of time is its timestamp, which improves time readability and builds an intuitive user interface.

The commonly used functions for converting a `struct_time` object to a string include:

- `Asctime()`
- `Strftime()`

Asctime()

The *asctime()* function converts a `struct_time` object and time tuple to a timestamp.

```
>>> import time
```

```
>>> time.asctime(time.gmtime())
```

```
'Mon Jan 6 18:44:19 2020'
```

```
>>> time.asctime(time.localtime())
```

```
'Mon Jan 6 21:45:01 2020'
```

The *gmtime()* and *localtime()* functions return `struct_time` instances for both utc and local time. These instances are converted to a string timestamp using the *time.asctime()* function. *Time.asctime()* passes tuples as an argument instead of floating point numbers. The parameter passed to the function is optional. If no parameter is passed, then the current local time will be automatically used.

Strftime()

Strftime() works in the same manner as *asctime()*, but it has additional formatting options that allow you to format the string into a more meaningful manner. The string format time (*strftime()*) formats *struct_time* object or the time tuples to a more readable form.

Strftime() accepts two arguments:

- Format: which indicates the order and form in which time elements in a string appear.
- T: which is an optional time format.

Format strings use directives (character sequences) to specify the time element. For example, %d specifies the day of the month, %m dictates the month, and %y represents the year.

Example: output date using iso 8601 local time standard

```
>>> import time
```

```
>>> time.strftime('%y-%m-%d', time.localtime()) '2020-01-06'
```

chapter summary

In this chapter, you learned how to work with the date and time format. To manipulate date and time, a datetime module is needed to provide different classes and categories like date, time, datetime, timedelta, and tzinfo. You also learned how to:

- Use datetime objects by importing the objects before execution of a code
- Use the *date.today* function to print the current local time of the day. You also saw how to manipulate the function by indexing a particular day and print an individual date, month, or year

- Use the `date.time` object to obtain the current time in terms of hours, minutes, and seconds
- Use the `strftime()` function to format time as a string. You saw how to print time and month separately
- Use the `timedelta` object to estimate both future and past dates
- calculate special days, such as the number of days left before your birthday based on the current time
- Use the `strptime()` string method to change the date format
- Work with `pytz` module to change the time zone
- Import the `time` module and manipulate time. You learned how to work with dates and times using epochs, time zone, and daylight savings time
- Represent time using floating point numbers, tuples, and `struct_time`
- convert different time presentations, suspend thread execution, and use

`Perf_counter()` to measure codes performance

In the next chapter you will learn about file processing techniques including how to open a file, read from it, write into the file, and close the files. You will also learn about file methods and how to use `os` and `os.path` modules.

chapter six:

File processing

What is a file

A file consists of a contiguous set of bytes used to store a collection of related information in a disk. The data in the file can be anything and is organized in a specific format, such as a text file or an executable file. Files are stored in non-volatile memory, which stores data permanently. Non-volatile memory structures include hard disks, which are unlike random access memory (ram), which only holds data and instructions temporarily.

components of a file

A file consists of three parts

1. File header: this is the metadata about file contents and it stores information about the file name, size, type, and properties.
2. Data: this represents the contents written in the file by the user.
3. End of file (eof): this is a special character that shows the reader has reached the end of the file

All files are stored with a specific extension based on the type of the file. For example, an image has the extension .gif because it conforms to the specifications of the graphics interchange format. There are different file extensions you can use when operating files.

File paths

When accessing a file, you have to provide the file path, the specific location where the file is stored in the computer memory. This is possible because of the operating system stored in your computer as it prompts file handling by the user.

The file path represents a string that indicates the location of the file. The file path consists of:

1. Folder path: where the file is located in the file system. It can be in subsequent folders which can be accessed using the backslash \ in windows or forward slash / in unix.

2. File name: this is the unique name used in storing the file.
3. Extension: this indicates the file type. It is shown at the end of the file path followed by a period (.) Then the file type. For example, myfile.doc

If you want to access myfile.doc stored in documents folder and your current location is the user folder, you can access the file by providing the file path, for example:

Users/faith/documents/myfile.doc

once you open the file, you can either read from the file or write into the file.

opening files in python

To use any file, you have to open it. This is done by making a call to the *open()* built-in function in python. The *open()* function takes the file path as an argument to the functions. When the function is invoked, it returns a single file object that allows reading or modification of the file.

```
>>> sample_file = open ('test.txt') # opens a file in the current folder
```

```
>>>sample_file = open (c:/users/faith/documents/myfile.doc) # using file path
```

When opening a file, you can specify the file mode, whether to read 'r', write 'w', or append 'a' content into the file. You can also open a file either in a text mode or binary mode. The text mode is the default form for opening files. The binary mode returns files inform of bytes and is mostly used when dealing with non-text files such as executable files or images.

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

File operation modes

Example:

```
sample_file = open ("sample.txt")
```

```
sample_file = open ("sample.txt", 'w')
```

```
sample_file = open ("kaa.bmp", 'r+b') #to read and write data in a binary mode
```

When working with files in text mode, you have to specify the encoding type. otherwise, your code may behave differently when using different platforms.

```
Sample_file = open ("sample.txt", mode='w', encoding='utf-8')
```

Buffered binary files

The buffered binary file type allows reading and writing into binary files written as `open('sample.txt', 'rb')` or `open('sample.txt', 'wb')`.

When you call `open()`, the file object data returned is either a `bufferedReader` or a `bufferedWriter`.

Raw files

A raw file type is used as the low-level building blocks for binary data and a text stream. The file is opened by `open ('sample.txt', 'rb', buffering=0 )`.

closing files in python

After carrying out all file operations, you have to close the file and free up the memory. computer memory is a limited resource and once not in use, you should deallocate the memory by closing all opened files. To free up the memory resources, we use the `close()` method.

file operations

```
Sample_file = open ("sample.txt", mode = 'w', encoding = 'utf-8')
sample_file.close()
```

If an exception occurs while performing a file operation, the program will terminate without closing the file. To prevent this from happening, you can implement python try and finally calls to catch all exceptions in the program.

Try:

```
Sample_file= open ('sample.txt')
```

Finally:

```
Sample_file.close ()
```

```
with open ('sample.txt', 'r') as my_file
```

```
    my_file.read("This code will automatically close your file!")
```

With the try...finally method, you're assured of proper closing of the file even when an exception is raised. You can modify your program code using *with* statements. This ensures the file is closed once the block of statements leaves the *with* statement. A *with* statement is highly recommended as it provides a cleaner code and makes it easy to handle exceptions.

Writing files in python

You can write new content or update python file with new data. To write to a file, you have to first open the file the call the write mode ('w'). You can also append 'a' data or exclusively create a file using the 'x' mode.

The 'w' mode can overwrite the previous content in the file if you're not careful. You can also use the *write()* method to write a string or sequence of bytes . The *write()* method can also return the number of characters written in a single file.

Example:

```
with open ('sample.txt', 'w' encoding = 'utf-8) as fl:
```

```
    fl.write ('\nThis is a sample file')
```

```
    fl.write ('Python files')
```

If the sample.txt is not available, python will create a new file named sample.txt. If the file exists, it is overwritten.

Reading files in python

For you to read a file, you have to open it in reading mode. You can use the 'r' mode to read a file character or use the *read()* method to read a file. When using the *read()* method you can pass the number of characters you want to read as an argument *read (size)*. If the read size is not specified, the function will call to read up to the end of the file.

Example:

```
>>> fl = open ("sample.txt", 'r', encoding = 'utf-8')
```

```
>>> fl.read(4) # reading the first 4 data elements 'this'
```

```
>>> fl.read(4) # reading the next 4 characters including space ' is '
```

```
>>> fl.read() # read up to the end of file 'a sample file \nthe file \nhas three lines\n'
```

```
>>> fl.read() # reading past end-of-file returns an empty string "
```

As seen in the above program, the *read()* method returns a newline in the form of `'\n'`. If you reach the end-of-file and invoke the read method again, it returns an empty string.

You can also use for loop to read a file line by line.

```
>>>for line in fl:
```

```
>>>print (line, end= '')
```

This is a sample file

The file

Has three lines

The `line` keyword is equivalent to `'\n'` and prints data into a new line. Passing `line` as a parameter in the *print ()* function ensures you don't print two lines when printing data.

```
>>> fl.readline()
```

Alternatively, the *readline()* function can be used to read each file separately. The function reads the file up to the newline. It also includes the newline character.

‘this is a sample file\n’

```
>>> fl.readline()
```

The file\n

```
>>> fl.readline()
```

‘has three lines\n’

```
>>> fl.readline()
```

‘,’

Appending to a file

```
sample_file = open('sample.txt', 'a')
```

```
sample_file.write("Add this data to a new line")
```

```
sample_file.close()
```

You can append to a file or write at the end of an existing file using the ‘a’ as the argument mode.

The append functions updates an existing file with new data without overwriting it.

Python file methods

Method	Description
<code>close()</code>	Close an open file. It has no effect if the file is already closed.
<code>detach()</code>	Separate the underlying binary buffer from the <code>TextIOBase</code> and return it.
<code>fileno()</code>	Return an integer number (file descriptor) of the file.
<code>flush()</code>	Flush the write buffer of the file stream.
<code>isatty()</code>	Return <code>True</code> if the file stream is interactive.
<code>read(<i>n</i>)</code>	Read at most <i>n</i> characters from the file. Reads till end of file if it is negative or <code>None</code> .
<code>readable()</code>	Returns <code>True</code> if the file stream can be read from.
<code>readline(<i>n</i>=-1)</code>	Read and return one line from the file. Reads in at most <i>n</i> bytes if specified.
<code>readlines(<i>n</i>=-1)</code>	Read and return a list of lines from the file. Reads in at most <i>n</i> bytes/characters if specified.
<code>seek(<i>offset</i>, <i>from</i>=<code>SEEK_SET</code>)</code>	Change the file position to <i>offset</i> bytes, in reference to <i>from</i> (start, current, end).
<code>seekable()</code>	Returns <code>True</code> if the file stream supports random access.
<code>tell()</code>	Returns the current file location.
<code>truncate(<i>size</i>=<code>None</code>)</code>	Resize the file stream to <i>size</i> bytes. If <i>size</i> is not specified, resize to current location.
<code>writable()</code>	Returns <code>True</code> if the file stream can be written to.
<code>write(<i>s</i>)</code>	Write string <i>s</i> to the file and return the number of characters written.
<code>writelines(<i>lines</i>)</code>	Write a list of <i>lines</i> to the file.

Python has various methods used in file manipulation. Some of the commonly used methods include:

Python input/output operations

A python module offers various built-in functions to enable us to manipulate files in the python prompt. The *input()* and *print()* functions are the standard input and output operations widely used. The io python module provides facilities necessary to handle i/o operations. The i/o operations are categorized into three:

1. Text i/o
2. Binary i/o
3. Raw i/o

Each of the categories has a concrete object or stream. Each of the stream object can either be a read-only, write-only, or support read-write options. The stream

object enable arbitrary random access both forward and backward in any location or allow sequential access especially when using sockets or pipes.

The i/o streams are dependent on the type of data passed as arguments. For example, if a *str*

object is passed to the *write()* method, the binary stream will generate a *TypeError*.

Text i/o

Text i/o streams receives and sends *str* objects. That is, if the secondary memory or backing store is made of bytes, then there is transparency in the encoding and decoding of the data. A text stream is created using the *open()* method. You can also specify the encoding, although its optional.

```
My_file= open ('file.txt', 'r', encoding= 'utf-8')
```

Binary i/o

A binary stream accepts bytes-like objects and generates byte objects. It doesn't require any encoding or decoding. The binary stream is applied on non-text data and in manual controls of text data.

A binary stream is represented with *open()*, which accepts mode string 'b'.

```
My_file= open ('file.txt', 'rb')
```

Binary streams are also available as *bytesio* objects in the system memory.

```
My_file=io.BytesIO(b 'this is binary data: \x00\')
```

Raw i/o/ unbuffered i/o

This acts as a low-level building block of both binary numbers and texts streams. The raw stream is rarely used and it's created by opening a file in binary mode and disable buffering.

```
My_file= open ('file.txt', 'rb', buffering= 0)
```

The *io* module has specific built-in functions for handling data. The *input()* and *print()* functions are the standard input output operations widely used.

Fileio has os level files with bytes of data and implements both the rawiobase and iobase interfaces.

Python output using print()

The *print()* function is used to display output data on a standard output device or file.

```
>>>print('this is our first output to the screen')
```

This is our first output to the screen

Num= 15

```
>>>print('the num value is:', num)
```

The num value is: 15

The actual syntax of the *print()* function is `print(*object, sep= ' ', end=;\n', file=sys.stdout, flush=false)`.

The object is the value to be printed, and the sep is a separator with an empty space character. The end ensures all characters are printed. The file is the object where all data values are printed with the default value being sys.stdout.

```
>>> print (2, 3, 5, 6, 7)
```

2, 3, 5, 6, 7

```
>>> print (2, 3, 5, 6, 7, sep='*') 2*3*5*6*7
```

```
>>>print (2, 3, 5, 6, 7,sep='#', end='&')
```

2#3#5#6#7

output formatting

If you want to make the output more attractive, you can use *str.format ()* method on any string.

```
>>> a=10; b=15
```

```
>>> print ('printing value of a is {} and b{}'.format(a, b)) printing value of a is  
10 and b is 15
```

The curly braces are placeholders. You can specify how the output is printed by using numbers or tuple index.

```
>>> print ('welcome to {0}{1}'.format('python', 'programming'))
```

Welcome to python programming

```
>>> print ('welcome to {1} in {0}'.format('python', 'programming'))
```

Welcome to programming in python

The str or bytes-like object can also be used in a file to both read and write to the file. String files opened in text mode can use stringio in-memory stream. Bytesio is used to open files in binary mode.

Python input

Input operation allows us to take input data from the user. To do this, an *input()* function is used followed by the string you want to display on the screen. The syntax is `input([user data])` where user data is the input string you want displayed on your screen. The user data can be optional.

Example:

```
>>> name= input ("enter your name:")
```

Enter your name: faith

```
>>> name
```

Faith

In the above example, you have entered your name as a string.

class hierarchy

I/o streams are organized as a hierarchy of class during the implementation process. The streams have an abstract base class (abc), which specifies the various categories of the stream, and a concrete stream class, which provides standard stream implementation.

The abstract base class supports the implementation of the concrete stream class by providing various implementation methods. For example, *bufferediosbase* supports unoptimized implementation of *readinto()* and *readline()* methods.

At the top of i/o hierarchy is the *iosbase* base class ,which defines the basic interface to the stream. The class doesn't separate operation for reading and writing to a stream. Implementation of abstract base class methods are allowed to raise unsupported operation.

Rawiosbase abc extends the functionality of *iosbase*. It handles reading and writing bytes into the i/o stream. Subclasses of file i/o rely on *rawiosbase* to provide a file interface in the machine file system.

Bufferediosbase is an abstract base class which handles buffering on a raw byte stream. The *bufferedReader* and *bufferedWriter* subclasses are used to buffer readable, writeable, and both read/write streams. Random access streams uses *bufferedRandom* as an interface buffer.

Textiosbase abc is a subclass of *iosbase* and it handles text streams and both encoding and decoding of text streams.

Importing python os module

When writing complex programs in python, it is essential to break the program into several modules. A module is simply a file with various definitions and statements. For example, a python file has filename followed by file extension. All python files end with the extension *.py*. A python definition inside a particular module can be imported into another python module. You can also import the module into an interactive python interpreter.

You can import modules into python interface by using the keyword *import*. For example, to import the *math* module, we use *import math*.

```
>>> import math
```

```
>>> from math import pi 3.141592653589793
```

When you import the math module, all the definitions inside the module are available for our disposal. This allows you to import specific attributes and functions using the '*from*' keyword.

When importing modules, python locates the module definition from different places defined in the *sys.path* which contains a list of directories.

The os module in python allows you to perform various operating system functions. It provides you with functions for creating a new directory (folder), deleting a directory, renaming the directory, or accessing the contents of the directory.

These os functions allows you to interface with the underlying operating system running on your machine and perform various tasks.

creating a director with the os module

To create a new directory in python, the *mkdir()* function from the os module is used.

```
>>> import os
```

```
>>> os.mkdir("d:\\newdir")
```

This created a new directory in the path specified by the string in the argument of the function. Now open your drive d in windows explorer and locate 'newdir' folder created.

Switching directories

```
>>> import os
```

```
>>> os.chdir("d:\\newdir")
```

You can switch from the current working directory to the new directory created and do all your operations there. This can be done through *chdir()* function.

This will change your directory path to the new directory created. To know whether the current working directory has changed, use *getcwd()*.

```
'd:\\newdir'
```

```
>>> os.getcwd ()
```

The *getcwd()* doesn't take any argument and it confirms directory path has changed by returning the new directory path. Any file operation carried will be saved in this new directory.

Note: directory paths are always relative. If you set the current directory to drive d and then to newdir without indicating the preceding path, then the current working directory will change to d:\\newdir.

You can change the current directory to its original path by passing “..” As argument in the *chdir ()* function.

```
>>> os.chdir ('d:\\newdir')
```

```
>>> os.getcwd()
```

```
'd:\\newdir'
```

```
>>> os.chdir('..')
```

```
>>> os.getcwd()
```

```
'd:\\'
```

Removing a directory

```

>>>os.chdir('newdir')
>>>os.getcwd()
'd:\\newdir'
>>>os.rmdir('d:\\newdir')
PermissionError: [WinError 32] The process cannot access the file because it is being used by another process: 'd:\\newdir'
>>>os.chdir('.')
>>>os.rmdir('newdir')

```

To remove an existing directory, the *rmdir()* function is used. This function deletes the specified directory. For a directory to be removed, it must be empty. You cannot remove a current working directory. In this case, newdir cannot be removed since it's the current working directory. To delete it, you have to change it from working directory, then delete it.

You can also list all the files and sub-directories within a particular directory using the *listdir ()* function. If no directory is specified, then the files and directories in the current working directory will be displayed.

The os modules have a variety of functions you can use with the most common functions on how to work with directories. other functions include:

1. *os.name*

This function is used to retrieve the name and other credentials of the operating system your python program is running on.

```
>>> import os
```

```
>>> print (os.name)
```

1. *os.popen()*

This is a file object manipulation method that opens to a pipe or from a command. You can either read or write to the file based on the specified file mode.

Import os

File=mku.doc

popen() is the same as open() function

```
Fl = open(file, 'w')
```

```
Fl.write("python programming os functions")
```

```
Fl.close()
```

```
Fl = open(file, 'r')
```

```
Text = fl.read()
```

```
Print(text)
```

popen() allows you to access the file directly through its pipe

```
Fl = os.popen(file, 'w')
```

```
Fl.write("your first lesson")
```

```
// output
```

Your first lesson

1. *os.close()*

All files opened with *os.popen()* are closed with *os.close()* or *close()*, just like the normal *open()*

Function. If you use *os.close()* on a file opened with *open()*, python will throw *TypeError*. Therefore we can use *os.close* or *close()* to close the above program.

```
os.close(fl)
```

1. *os.error*

This function raise an oserror in case of an invalid file or files that cannot be accessed by the operating system despite having the correct path, name, and argument type.

1. *os.rename()*

It is used to change the name of a file. The function only works if the file exists and you have the rights to change the file.

Import os file=mku.doc

os.rename(file, mkunew.doc)

os path module in python

This module has functions used to handle python pathnames. There are different functions supported by the module like retrieving path names, merging files, and normalizing functions among others.

These functions take only strings and bytes as arguments. That is, you can only pass parameters of string objects or bytes to the os path functions. The returned file name or path is of the same type.

Functions of os path module

1. *os.path.basename (path)*: this function returns the file basename. That is, it returns the file name based on the path to its location.

```
>>>#using basename ()
```

```
>>>import os
```

```
>>>output= os.path.basename ('/users/dir.py)
```

```
>>>print (output)
```

```
'dir.py'
```

```
1. >>>Output= os.path.basename ('/users/dir.py')
>>>print (Output)
'/users'
```

os.path.dirname(path): this function returns the directory name from the specified path. It doesn't indicate the path name, only the name of the directory is returned.

1. *os.path.isdir(path)*: this tests whether a particular path is an existing directory or not.

```
>>> Output= os.path.isdir ("C:\\windows")

>>> print (Output)

yes
```

```
1. >>>Output= os.path.basename ('/users/dir.py')
```

os.path.isabs(path): a function to confirm whether the specified path is absolute or not. An absolute path is one that begins with a backslash in windows and a forward slash in unix system.

2. *os.path.normalcase (path)*: in windows, this function converts the filename path into lower case and the forward slash is converted to backward slash. The path remains the same for unix and mac os systems.

Both os and os.path modules provides you with a portable way of interacting with platform- dependent functions and perform various input and output file operations.

chapter summary

File input and output processes are very important in the operation of i/o streams. In this chapter, you learned:

- What makes up a file, why files are important in python programming and file paths.
- To understand the basics in reading and writing into the file, learn how to open and close files as well as appending to a file.

- Different file processing modes and how to apply them in the manipulation of files.
- Various file functions and how to apply them in various file input/output operations.
- The import python os module and interact with various operating system dependent functionalities.
- How to use python input to prompt you to enter data into the system.
- How to use the *print()* function to display your output and format the output based on your needs.
- How to use os.path module to confirm the existence of a directory, create a new directory , and change the directory path to make it the current working directory.

In the next chapter you will learn exception handling, how to use else clause and finally clause, using exceptions for flow control and exception hierarchy.

cHAPTER SEVEN:

EXcEPTIoN HANDLING

Exception

An exception is any error that occurs during program execution. When an error occurs, a python program generates an exception to handle the error and prevent the exit of the program. Python has a variety of built-in exceptions that generate an error when something goes wrong in your program.

When an exception occurs, it forces the current process to stop and transfer the control to the calling process until the error is handled. If the error is not handled, the program will crash. Therefore, exceptions handle errors and any other special conditions within the program.

For example, function a calls function b while b calls function c . If an exception occurs during execution of function c, function c passes the exception to function b which is then passed to function a. Function c doesn't handle the exception but passes it to other functions. If the error is not handled at function a, an error message will be generated and the program will terminate.

If you suspect a certain code will generate an error, it's necessary to create an exception to handle the error.

common exception errors in python

- **Ioerror:** this error occurs when you cannot open a certain file.
- **Importerror:** an error is generated if python cannot import or find the invoked module.
- **Valueerror:** it occurs when the argument passed to a built-in function or operation has the right data type but an inappropriate value.

- Keyboardinterrupt: this occurs when you press the keyboard interrupt key (ctrl + c or the delete key)
- Eoferror: this error happens when *input()* and *raw_input()* reach the end-of-file (eof) without reading any data.

catching exceptions in python

Exceptions are handled using a try statement. If you have a critical condition or operation that might result in exceptions, you can place the program code inside the try clause and the code that handles the exception in the except clause. The operation carried out after the exception is based on the program needs.

Syntax:

Try:

statements for block operations except:

#if execution occur, execute this block of statements else:

#no exception, execute this block of statement

The else block is executed if no exception is raised in the try block. Example:
python code to open a file and write into it without any exception

```
fl=open("samplefile.doc", "w")
```

```
Fl.write("a sample file for exception handling!!") exceptioerror:
```

```
Print ('error:no file available to read data') else:
```

```
Print ('you have successfully written content in the file') fl.close()
```

```
//output
```

You have successfully written content in the file

Example 2: opening a file where you don't have access permission
`fl=open("samplefile.doc", "r")`

`Fl.write("a sample file for exception handling!!")` exceptioerror:

Print ('error: no file available to read data') else:

Print ('you have successfully written content in the file') `fl.close()`

// output

'Error: no file available to read data

The try-except statements catch all exceptions that occur in the program. This may not be a good programming practice since the method only catches all errors but the programmer doesn't know the root cause of the problem.

Argument of an exception

Exceptions receive arguments which provides additional information the exception. This can be done by passing a variable in the except clause.

Try

Statements

Except (exceptiontype, argument):

Statements based on the value of argument else:

No exception statement

If you're writing code to handle a single exception, you can pass a variable to the except statement. If you have multiple exceptions, the variable is passed

to the tuple of the exception.

The variable received either a single value or multiple values in form of tuple. The tuple will have error number, error string and error location.

Example:

```
#function definition
```

```
Def temp_convert ( var):
```

```
Try:
```

```
Return int (var)
```

```
Except (valueerror, argument):
```

```
Print ('\nthe argument passed is not a number', argument) temp_convert('xyz')
```

Except clause with multiple exceptions

Syntax

```
Try:
```

```
Execute operation statements; except exception 1:
```

```
#block statements to handle exception except exception 2:
```

```
#block statement to handle exception else:
```

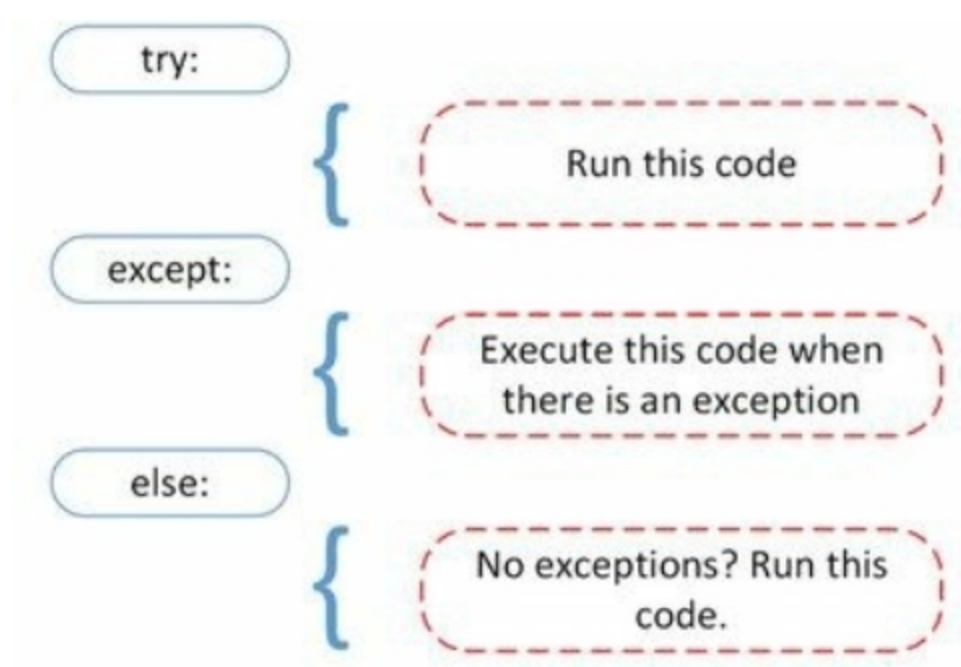
```
# no exception found, execute this code
```

A single try statement may have more than one except statements. This is of great importance if the try block has several statements that throw different types of exceptions. You can also create a generic except clause that allows you to handle different exceptions.

The block of code within the try clause is executed statement by statement. If an exception occurs, the except block is executed and the remaining try code is skipped.

Errors are always caught in the try clause while the except clause is invoked to handle the error. That is, if an error is encountered in try block, the execution of the try block is stopped and the control transferred to the except block to handle the error.

Else clause



Python program allows the use of else statements. These statements are used to instruct a program to execute a specific block of code if no exceptions are found.

Try:

```
codecs.lookup()
```

Except lookuperror as error:

Print (error)

Else:

Print ('no error found: executing the else clause.')

Since there is no exception when the code is run, the else clause is executed.

You can also insert the try block inside the else clause and catch any other exceptions.

Raising exception

You can raise an exception in a program by using 'raise exception statement'. This breaks out the current code block and return exception statements back until the exception is handled.

In python, we create exceptions using the raise keyword followed by the name of the exception with:

Raise exception_name

When you raise an exception, it interrupts the normal flow control of the program. When this happens, the python interpreter looks for the code in its run-time-stack that can handle the message. The search begins from the current program state up to the end of run-time-stack. The execution process follow the order in which functions are called.

A try:except block is used to handle the exception.

The raise statement creates an exception object and leaves the program execution process to look for the try statement with a matching except clause in its run-time-stack. The raise statement can either divert a program

execution if matching except clause is found or exit the program because a matching except clause for handling exception wasn't found.

The exception object has a message string to provide meaningful error messages. It also makes it easy to add more attributes to the exception.

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):

KeyboardInterrupt

>>> raise MemoryError("An error occurred when loading memory")
Traceback (most recent call last):
MemoryError: An error occurred when loading memory

>>> try:
    val = int(input("Enter a positive integer: "))
    if val <= 0:
        raise ValueError("That is not a positive number!")
except ValueError as ve:
    print(ve)

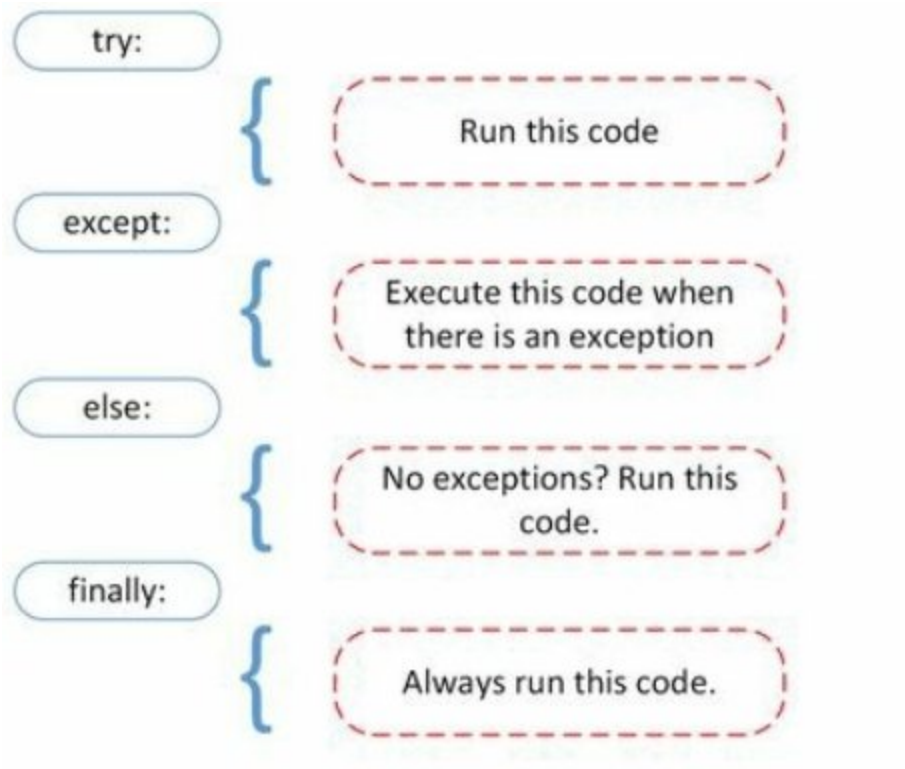
Enter a positive integer: -6
That is not a positive number!
```

Example:

Try...finally

The try statement can also have a finally clause. The clause can be optional and whenever used, it executes the program and free up the external resources. If you're working with files on a remote data center connected via a network or with a graphical user interface, you have to free up the memory after execution.

You must clean up the resources once done no matter what. closing of the file, disconnecting from the network is always done in the finally clauses to guarantee you a safe execution.



Example:

Try:

```
Fd= open("sample.txt",encoding = 'utf-8')
```

```
# performing a file operations
```

Finally:

```
Fd.close()
```

Exception handling for flow control

As we have discussed earlier, exceptions occur due to conditions that can't be handled by the normal flow of the program. In this case, exceptions are normally classified as errors. Although all errors are handled using exceptions in python, not all exceptions are errors.

In normal flow-control of a python program, instructions are executed one after the other using if statements, loops, and function calls. In these three constructs, the program flow-control is as follows:

In if statements, only one statement is executed at a time in a block of statements then the flow control is transferred to the first statement or line after the if statement.

In loops, when the end of the loop is reached, the program's flow-of-control loops back (is transferred) to the start of the loop to test whether the loop needs to be executed again. If no further execution is required inside the loop, then the flow of control is transferred to the first statement after the loop.

If you call functions in python, the flow control is transferred to the first statement of the called function and executes the code. After function execution, the flow control is transferred to the next statement line in the program.

Therefore, the flow-of-control is a sequential process because when a call is made, it always executes the first statement in the program before proceeding to the next statement.

```
def main()
    A()

def A():
    B()

def B():
    C()

def C():
    D()

def D()
    # processing
```

Example: exceptions in flow control

The above program is executing statements in function d previously called by function c. c is called by b, while b is called by function a, and a called by main (). If d gets an interruption on its current processing task, it has to send a message to main () to try something different. D can't send messages directly to main (), so it has to rely on the normal flow-of-control to send a special message to function c to 'try something different'. When c gets the message, it stops its current processing and passes the message to function b. The same procedure is followed until the 'try something different' message gets to the main() function.

Since program d couldn't communicate with main() directly or with function a and b, it creates an exception using the raise command.

When the exception message is created, it interrupts the normal flow-of-control of the program. Python evaluates its run-time-stack for a code to handle the exception message. When the code is found, a try:except block is thrown to catch the exception.

If function d had a try:except block on the code that raised an exception message, then the flow control would only have been passed to the local except block. This would have enabled d to solve its problems without transferring control to the main().

```

def main()
    A()

def A():
    B()

def B():
    C()

def C():
    D()

def D()
    try:
        # processing code
        if something_special_happened:
            raise MyException
    except MyException:
        # execute if the MyException message happened

```

Exception class hierarchy

A class hierarchy consists of a number of exceptions distributed across different base class types. In any programming application, errors occur when something unexpected happens. The errors can result from improper arithmetic calculations, a full or near-full memory space, formatting errors, or invalid file references that raise an error.

The series of errors raised are considered as exceptions since they're non-fatal and allow the execution of the program to continue. It also enables you to throw an exception to explicitly catch or rescue the exception raised.

The exception hierarchy is determined by the inheritance structure of various exception classes. All raised exceptions are instances of a class derived from the baseexception class. A try:except clause, using a particular class, can handle all exceptions derived from that class.

The hierarchy of the major built-in exceptions include:

Baseexception exception

Arithmeticerror floatingpointerror overflowerror zerodivisionerror
assertionerror

Most classes use keyword *exception* in the *baseexception* and *exception* in its parent classes. Subclasses uses the word *error*. A python program inherits a series of *abcerror* classes.

Baseexception class

The *baseexception* class is the base class of all built-in exceptions in python program. A raised python exception should be inherited from other classes. The *baseexception* class takes in a tuple of arguments passed during the creation of a new instance of the class. In most cases, a single string value is passed as an exception which indicates the specific error message.

The class also includes `with_traceback(tb)` method to explicitly pass the 'tb' argument to the traceback information.

Exception class

This is one of the most inherited exception type classes. All the classes classified as errors are subclass to the *exception* class. Therefore, if you create a custom exception class, it should inherit properties of the *exception* class.

The *exception* class has a variety of subclasses that handle the majority of errors in python. Some of these subclasses include:

1. Arithmeticerror: this acts as a base class for various arithmetic errors, such as when you're trying to divide a number by zero.
2. AssertionError: this is an error generated when a call to `[assert]` statement can't be completed or it fails.
3. Buffererror: in python, applications have access to a low level memory stream in the form of buffers. If the buffer memory fails, an error is generated: `buffererror`.

4. `Eoferror`: this type of error is raised when an *input ()* reaches the end of a file without finding the specified file or any data.
5. `Importerror`: to perform advanced functions or work with complex programs in python, you have to import modules. If the import statement fails, an `importerror` will be raised.
6. `Lookuperror`: just like in `arithmeticerror`, a `lookuperror` acts as the base class from which is inherited by other subclasses. If an improper call is made to the subclass then a `lookuperror` is raised.
7. `Memoryerror`: if a python program runs out of memory or there is not enough memory stream to run the application, a `memoryerror` is raised.
8. `Nameerror`: this is an error that occurs when you try to call an unknown name or use an identifier with an invalid name.
9. `oserror`: an error raised due to the failure of an operating system.
10. `Valueerror`: it occurs if a method receives parameters of correct type, but the actual value passed is invalid due to some reasons.

Keyboardinterrupt

This occurs when the user presses `ctrl+c` key combination. This creates an interrupt to the executing script.

Generatorexit

The *generator* is a particular iterator which simplifies the iteration process with constantly changing values. It allows you to use the *yield* statement inside the generator code block. This type of exception allows python to generate a new value when a call to *next()* function is made. When `generator.close()` method is invoked, a `generatorexit` instance is raised.

Systemexit

When you call `sys.exit()` method, a `systemexit` exception is raised. The `sys.exit()` function call closes the current executing script and then closes the python shell. Since this is an exception, you catch the exception before the script shuts down and exit by responding to the `systemexit` exception immediately.

chapter summary

From this chapter, you learned:

- That an exception happens when a certain condition interrupts the normal flow-of-

control of a program.

- That if an exception is raised, python first searches for a try:except block in the program's runtime stack to handle the exception. When the first statements in the try:except block handle the exception, then the flow control is transferred to the next line in the program.

- About common exception errors in python and how to handle each of them.

- How to use the except clause and finally clause when handling exceptions

- About exception hierarchy

In the next chapter you will learn about graphics and image processing in python as well as how to import a simple 2d drawing module.

chapter eight: graphics and image processing

Python graphics

To make graphics programming more fun, professor john zelle designed a custom simple graphics library to make python graphics as simple as possible for students. This library allows you to generate a graphical output using simple lines of codes.

You have to download the [graphics.py](#) library in your machine before you can use it. Save it in the same folder as the program you're creating.

```
>>> c:\> python
Python 3.6.3 (default, Jan 10 2020, 03:12:16) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from graphics import *
```

|

Test your installation and import graphics:

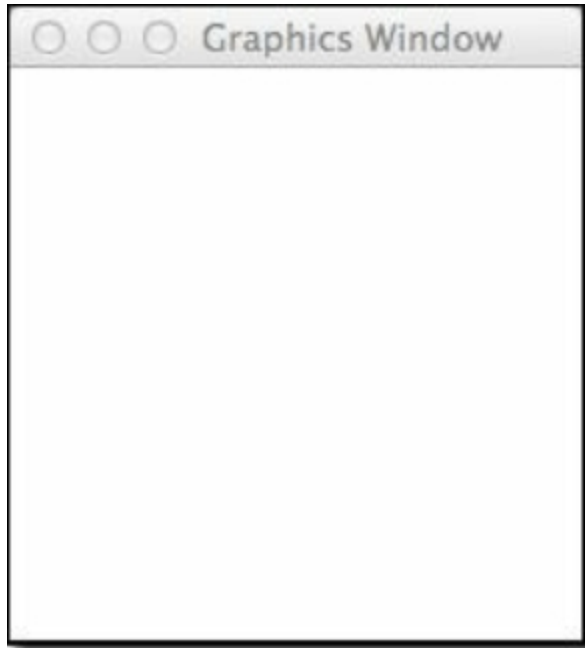
If no error message generates, your installation was successful, and you can go ahead and start using the library.

opening a graphics window

```
>>> from graphics import *
>>> win = GraphWin()
>>> win.getMouse()
```

open the code editor and enter the following code:

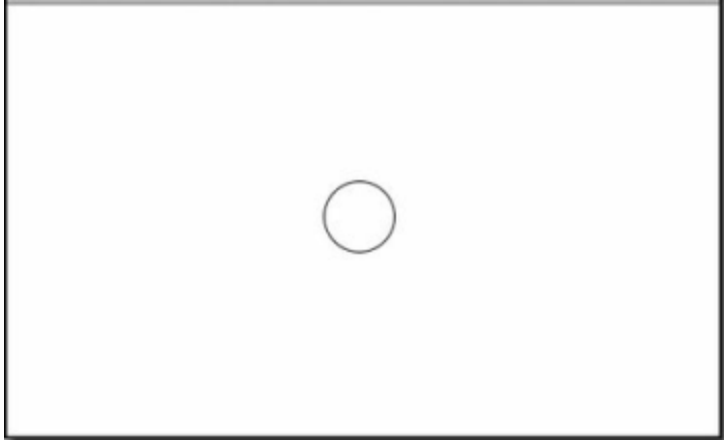
The graphwin is a call to the system to open a new window while getmouse instructs the system to active your mouse button. Save the command and run the code.



```
>>> pt = Point(270, 140)
>>> cir = Circle(pt, 27)
>>> cir.draw(win)
```

Drawing a circle

To draw a circle, you have to call the circle, pass the parameter, and specify the point where the circle should be drawn. The *draw()* function instructs the system to draw the circle in the graphics window.



You can refill the circle with any color of your choice by calling:

```
>>> cir.setfill ('red')
```

You can also build other shapes into your graphics by making the same changes.

Turtle graphics

A turtle graphics is a robotic turtle made of different shapes that mimics simple moves. Turtle graphics is a great way of introducing graphics programming.

```
>>> import turtle
```

This allows you to import the module for handling turtle graphics.

Example: assuming you have a turtle robot that initially has a starting point (0,0) in the x,y axis. You can pass a command that will move it 10 pixels.

```
>>> turtle.forward (10)
```

When you run this code, it will display the robot moving 10 pixels in the direction it currently faces, drawing a line as it moves. Give it another command to draw a circle.

```
>>> turtle.circle(20)
```

This will change the robot image to a circle. The turtle library allows you to combine different commands to draw intricate shapes and pictures.

The turtle module allows you to use various commands, classes and methods interactively when running the module via the idle python shell. The module provide primitive graphics in an object oriented and procedural language simply because it uses *tkinter* for all underlying graphics.

classes of object-oriented interface

1. *Turtlescreen* class uses the graphics window as the playground to draw turtles. The *tkinter.canvas* and *scrolledcanvas* are arguments to the constructor class. *Screen()* returns an object of *turtlescreen* subclass.
2. *Rawturtle (rawpen)* defines an object that draws on a *turtlescreen*. A subclass *turtle* is derived from *rawturtle*.

Turtle methods

There are various methods you can use to manipulate your graphics. They are classified into:

1. Turtle motion (move and draws methods, tell turtle's state, setting and measurement methods)
2. Pen control (drawing control, color control, filing)
3. Turtle state(visibility, appearance)
4. Using events (onclick(), onrelease())
5. Special turtle methods (clone(), get_poly())

Example: moving the turtle to absolute position. If the pen face downwards, draw the line.

```
>>> import turtle
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

```
>>> import turtle
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(40)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180) # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

Example: draw a circle centered at (0,0).

Image processing

Today's world is data driven and different organizations rely on the use of images and video data to simplify processes. Almost 60% of graphics data in an application must be pre-processed. That is, the images must be analyzed, processed, and manipulated before being put to good use.

Python provides you with a plethora of algorithms on how to handle images. Image processing deals with various ways to handle image to image transformation. This involves both input and output image transformation mechanisms.

A computer vision is a field that deals with how computers understand or gain high-level information from a digital image or video.

An image processing algorithm accepts an image or a video as an input, processes the input, and outputs the results as an image or a video. A computer vision algorithm, on the other hand, processes the image or video inputs and constructively obtains meaningful information from processed data.

Working with images in python

An image is any 2-dimensional object with x and y spatial coordinates.

To use images in python, you have to install the standard library package to handle images. Pil is one of the standard python imaging library that provides python interpreter with various image manipulation capabilities.

other image processing libraries include:

1. *opencv*: this focuses on obtaining real-time computer vision with applications using 2d and 3d features, facial expression and recognition, robots, and human- computer interactions among others.
2. *Numpy and scipy library*: used mostly in image manipulation and processing
3. *Scikit*: this library provides multiple algorithms used in image processing.

4. *Python imaging library (pil)*: used when you're performing basic image operations like resizing, rotation of image, using different image formats or creating image thumbnails.

You can choose any python library based on your image needs or what you want to do. In this chapter, we will focus on creating a basic image processing in python.

Install required library

Install the required library to your machine

once you install the pil library, you have to import the image module from pil.

```
From pil import image
```

```
#opening an image
```

```
Img=image.open ('iphone.jpg')
```

```
#rotate image to 90 degrees and show
```

```
Img=img.rotate(90).show
```



```
// output  
convert and save() image
```

You can convert the image format from one form to another.

```
>>>img.save('iphone.png')
```

This code will save a new iphone image in png format on the same folder.

Resizing image thumbnails

```
>>> img.thumbnail (350,350)
```

```
>>> img.show()
```



Graphical user interface (gui)

A graphical user interface is an interface on the screen that allows interaction between the user and the system. The interface presents you with menus, text boxes, labels, buttons, menus, and main window that you can click with your mouse to perform a certain action. These gui components are collectively known as widgets.

There is a wide range of widgets and a programmer can place them logically on the window making it easy for the user to interact with.

Event loops

In any application, gui waits for you to trigger an event or do something. For example, when you type something on the screen via an input box or click on a button with the mouse, it triggers an event.

At the back end, the gui toolkit runs an infinite loop which is commonly known as an event loop. Events loops wait for events to happen, and then take action on the event depending on what the programmer wanted the

application to do. If the developed application doesn't catch the event, then it's ignored.

When you're writing a graphical user interface application, always remember that you have to link up each of the designed widgets with the event handler to do something.

If you block event loops, the graphical user interface becomes unresponsive and it may appear to freeze (taking forever to load). If you launch an application in the gui and it takes a long time to load, you should launch it as a separate thread or process. This prevents freezing of the gui application, giving you a better experience.

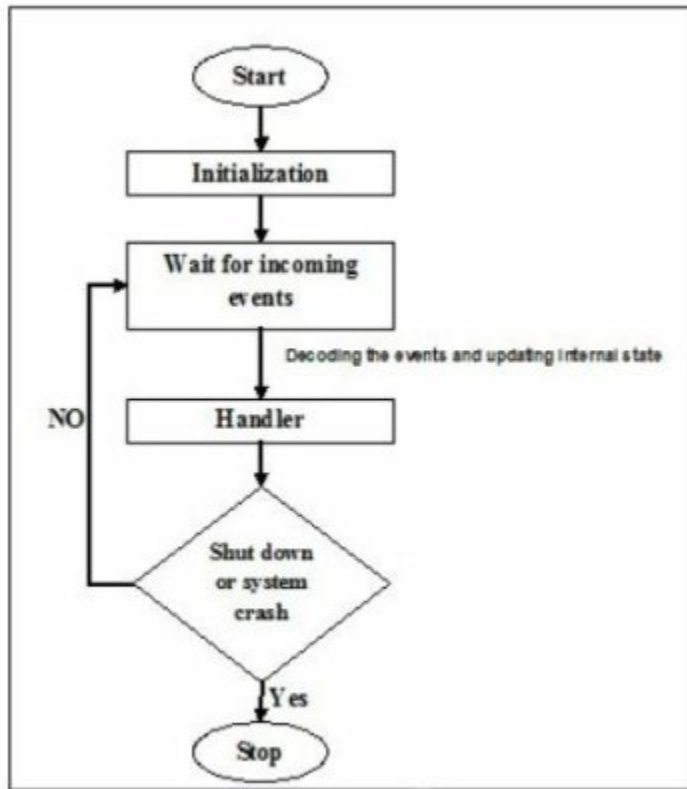
Event driven programs

An event driven programming application is a program that relies on user inputs like clicking on a command button or choosing an item from a drop line list, using text box among others types of events. The program flow depends on the events happening by always listening for new incoming events.

When an event occurs, then an event handler decides what to execute and the specific order of execution.

Events programs are separated from the rest of the program code since an event driven program is an approach not a language. Event driven applications can be created in any programming language. These events leads to the improvement of the program responsiveness, throughput and flexibility.

Event handlers



An event handler consists of functions or methods with the program statements that need to be executed once an event occurs. The event handler is a software routine for processing event actions like mouse movement, or keystrokes. Python event-driven programs rely on the event loop that listens to any new incoming event.

Binding and events

The tkinter python module relies on the event sequences to determine which event to bind to the handlers. The first argument passed to the event in the bind method determines the event to bind. An event sequence is represented as a string <modifier-type-detail> where the type field is used

To specify the event. The modifier and detail provide further information to the 'type' but are not necessary required. Therefore, they are left out in most cases.

The 'type' determines which event to bind. For example, a mouse click event or an event as a result of key press has an input focus.

`widget.bind(event, handler)`

The tkinter application is always inside an event loop in which it enters the `mainloop` method. Tkinter is a powerful graphical user interface tool that allows you to easily handle events by binding python functions and methods to events in widgets.

If event occurs and it matches the event description in the widget, then the associated handler is called with the old object that describes that event.

```
from tkinter import *

def single_button_click(event):
    '''
    Prints Single Click
    '''
    print("Single Click")

def double_button_click(event):
    '''
    Prints Double Click then exits
    '''
    print("Double Click")
    import sys; sys.exit()

widget = Button(None, text='Mouse Clicks')
widget.pack()
widget.bind('<Button-1>', single_button_click)
widget.bind('<Double-1>', double_button_click)
widget.mainloop()
```

Example: creating a motion event - moving a mouse inside the widget

```
from tkinter import *
def motion(event):
    print("Mouse position: (%s %s)" % (event.x, event.y))
    return

master = Tk()
msg = Message(master, text = 'Just Move it')
msg.config(bg='lightblue', font=('times', 24, 'italic'))
msg.bind('<Motion>', motion)
msg.pack()
mainloop()
```

Whenever you move the mouse inside the widget message, the mouse position will be printed, if you leave the widget, *motion()* is not called.

Tkinter provides you with diverse widgets like buttons, labels, textboxes, and other tools that you can use in a gui application. The buttons display developed applications while canvas widgets are used to draw shapes like lines, polygons, circles and other shapes in your application.

The *mainloop()* function ensure the window continuously stays on the screen until you close it. If you don't include the function in your program, then the window appears and closes itself since you haven't indicated it should continuously run. Therefore, the *mainloop()* puts your code in an endless loop.

Python predefined events

<button-1>

This event is used when you press the mouse button over the widget. Button 1 is for the left mouse button, button 2 is for middle mouse button while button 3 is for the right-most mouse button. You can also use *buttonpress* instead of just *button*.

<b1-motion>

This occurs when you move your mouse while holding the mouse button1 down. This provides the current position of the mouse pointer in the x, y members of the event object.

<double-button-1>

This is used when you double-click button1. You can also use triple-clicks, based on the prefixes, but if bound to both single click (button1) & double-click, then a binding call should be made to both.

<buttonrelease-1>

If you release button1, the current pointer location is shown in the x & y members of the object event.

<enter>

This is used when the mouse pointer is put inside the widget. (it shouldn't be confused with pressing the enter key)

<return>

This is used when you press the enter key

<shift-up>

This is used when you press the up arrow while holding down the shift key. You can also use a combination of alt, shift, and ctrl.

<configure>

This is used when the widget changes size or its locations. You should provide the width and height of the new widget event object passed at callback.

Frames

Frames are used to organize your widgets in a window. They help you determine where to place them on a window though you can also use widgets without the frames. You can also insert multiple frames in a single window and insert widgets in them.

```
from tkinter import *

rootWindow = Tk()

rootWindow.geometry("300x300")
rootWindow.title("Hello World")

topFrame = Frame(rootWindow)
topFrame.pack(side=TOP)
bottomFrame = Frame(rootWindow)
bottomFrame.pack(side=BOTTOM)

rootWindow.mainloop()
```

Example:

When you run the code, it will create two frames, one at the top of the window and the other one at the bottom of the window. Frames create invisible rectangles on the window. Try out the code and see how python creates invisible frames for you to insert widgets.

The invisible rectangle is placed inside the rootwindow. If you pass an instance of rootwindow inside the frame class, a frame is created to fit the size of the rootwindow. once the frame is created, *rootwindow.mainloop()* places the frame inside the window, where *pack()* can specify where it should be placed, whether at the top or bottom. You can also use the 'side' argument to place the fame anywhere you want inside the window.

Grid layout

A grid layout is another way you can represent frames. It uses rows and columns to create frames. When using grid layout, you can think of the window as a spreadsheet with rows and columns to place your widget.

```
from tkinter import *

rootWindow = Tk()
rootWindow.geometry("450x100")
rootWindow.title("Hello World")

# topFrame = Frame(rootWindow)
# topFrame.pack(side=TOP, fill=BOTH)
# bottomFrame = Frame(rootWindow)
# bottomFrame.pack(side=BOTTOM)

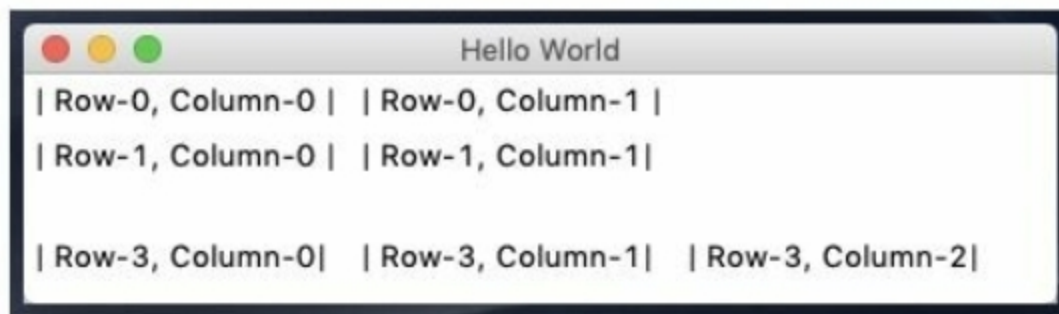
label1 = Label(rootWindow, text="| Row-0, Column-0 |")
label2 = Label(rootWindow, text="| Row-0, Column-1 |")
label3 = Label(rootWindow, text="| Row-1, Column-0 |")
label4 = Label(rootWindow, text="| Row-1, Column-1 |")
label5 = Label(rootWindow, text="| Row-3, Column-0 |")
label6 = Label(rootWindow, text="| Row-3, Column-1 |")
label7 = Label(rootWindow, text="| Row-3, Column-2 |")

# label1.grid(row=0) This will also work because default column value is 0
label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
label3.grid(row=1, column=0)
label4.grid(row=1, column=1)
rootWindow.grid_rowconfigure(2, minsize=20)
label5.grid(row=3, column=0)
label6.grid(row=3, column=1)
label7.grid(row=3, column=2)

rootWindow.mainloop()
```

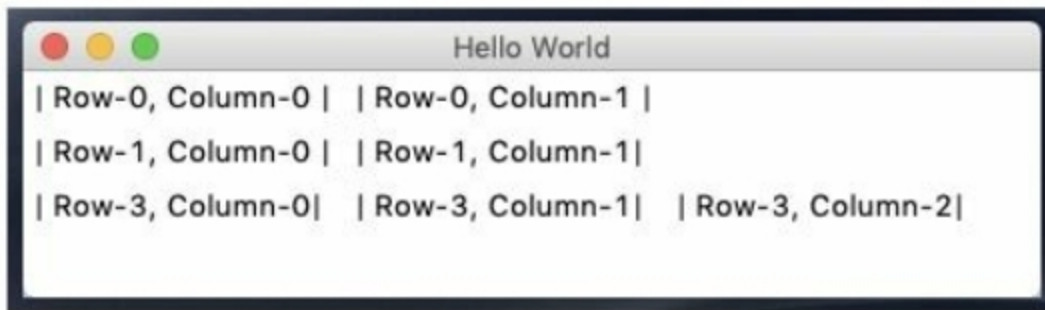
Example:

// output



Defining a label as shown above will allow you to enter text into your window as shown above. The created labels are placed in a grid layout by passing row and column number to the *grid()*.

calling *rootwindow.grid_rowconfigure(2, minsize=20)* creates an empty row in the window. In the program, we have used row [0,1, and 3] and skipped row 2. If you don't call the *grid_rowconfigure()* function, it will put label 5 in row 2 even though you have specified 0 be placed in row 3 as shown below.



Therefore, *grid_rowconfigure()* and *grid_columnconfigure()* will help space rows and columns in your grid.

Python asyncio module

The asyncio module provides an infrastructure for handling events in python by writing a single-threaded concurrent code using subroutines.

Any event-driven program should have an event loop function.

Using an event loop

This is a function used to handle all the events in a particular code. It loops along the entire execution program and keep track of the incoming and execution of events. The module allow only a single event to loop in any given process.

Methods used

Loop=get_event_loop(): this is an event loop for the current program in the current context.

Loop.call_later (time_delay,callback,argument): the method makes an arrangement for a callback that is supposed to be made after the provided time_delay seconds.

Loop.call_soon (callback,argument): the method makes an arrangement for a call back immediately. The callback should take place after *call_soon()* is returned and the control is taken to the event loop.

Loop.time(): this method show the current time based on the events internal clock. Asyncio.set_event_loop(): the current context to the loop is set based on event loop. Asyncio.new_event_loop(): this creates and returns a new loop object.

Loop.run_forever(): this will run until a call to *stop()* is made. Example:

```
import asyncio

def hello_world(loop):
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

loop.call_soon(hello_world, loop)

loop.run_forever()
loop.close()
```

// output hello world

chapter summary

In this chapter, we are able to learn how to:

- Use the graphics library to create and manipulate images
- Use the turtle python module to create graphical image
- Install the standard python image library in a program
- Import an image from a particular library
- Work with images in python and manipulate them
- Handle events and events-driven programming
- Use tkinter in event-driven programming
- Work with the asyncio module

In the next chapter you will learn network and client server networking. At the end of the chapter, you will be able to explain how communication takes place between a client and server and how cgi applications communicate with each other.

cCHAPTER NINE:

Network and client/server programming

Python programming provides network access through two levels: lower level access and higher level access. You can access the basic socket support and its underlying operating system at the lower level. This enables you to implement a client server network for both connection-oriented and connectionless protocol.

The python library provides high-level access to certain application level protocols like ftp, http, tcp, and others.

Socket programming

A socket provides a network connection and acts as the backbone for networking. Sockets transfer make information transfer between different programs and devices possible. For example, whenever you open your browser to search for information, you're creating a connection to the server for transfer of information.

Sockets can communicate within a process and exchange information between processes on the same device or between processes in different devices.

Sockets in python

Sockets acts as interior endpoints designed for sending and receiving data. When connecting to a network, each network connection will have two sockets for connecting to the communication device and program. Each socket has an ip address and a port.

The socket, or node, on one of the connected devices listens to a specific port with the ip address while the other socket forms another connection. The server machine forms the listening socket while the client reaches out to the server with various requests. Each device has 'n' number of sockets determined by the port number used. Different ports use different protocols.

Protocol	Port Number	Python Library	Function
HTTP	80	httplib, urllib,xmllrpclib	Web pages
FTP	20	ftplib, urllib	File transfers
NNTP	119	nntplib	Unsent news
SMTP	25	smtplib	Sending email
Telnet	23	telnetlib	Command lines
POP3	110	poplib	Fetching email
Gopher	70	gopherlib	Document transfer

Some of these protocols and port numbers include:

The above port numbers and protocols form a socket module essential in network programming.

Socket components

The socket library provides classes to handle a transport layer alongside an interface for connecting with other devices and programs.

Some of these components include:

Serial number	Term	Description
1	Domain	Domain protocol is used in transport mechanism and uses constant values like af_inet and pf_unix among others
2	Type	It provides communication between two nodes. Sock_stream is used for connection-oriented

communication while `sock_dgram` is used for connectionless communication.

3 Protocol

This is a protocol identifier for domain and type socket components.

This is a network interface which consists of:

An integer represented as a binary number in the host byte order.

4 Hostname

A string that specifies the `inaddr_broadcast` Address.

A string which acts as the hostname or an ipv6 address in colon or dotted notation.

A zero-length string to specify the address of `inaddr`

5 Port

The server machine listens to a client machine making requests via one or more ports. In this case, the port may be a string that contains the port number, the service being requested or just a fixnum port number.

Using a socket module

When creating sockets, you have to use the `socket.socket()` library function in the socket module. This can be done by:

`S= socket.socket (socket_family, socket_type, protocol=0)` where

The socket family: which is either in the form of `af_unix`, `af_inet`, or `sock_dgram`.

Socket type: uses `sock_stream` or `sock_dgram`.

Protocol: by default it's always 0.

Therefore, you can import the socket library and pass two parameters to it as shown below: `import socket`

`S= socket.socket (socket.af_inet, socket.sock_stream)`

The `socket.socket` is required by both the client and server machine to create sockets.

The `socket.af_inet` parameter represent ipv4 address family whereas `sock_stream` is a connection-oriented tcp protocol.

once the socket object is created, then you can use socket functions to create a client or server program.

These functions include:

Server socket functions

Method Description

`S.bind ()` This function binds the specified address based on the parameter

(the hostname and the device port number to the socket).

`S.listen()` A method used for setting up and starting a tcp listener.

`S.accept` It passively allow connection through the tcp client and waits until connection is established (blocking)

client socket functions

Method Description

`S.connect()` Used to actively connection. Initiate A Tcp Server

General socket functions

Method Description

`S.recv()` To receive a tcp message

`S.send()` For transmission of tcp message

`S.recvfrom()` For receiving udp messages

`S.sendto()` Transmission of udp messages

`S.close()` Function to close the socket

`Socket.gethostname()` Function to obtain the hostname of the device

connection to the server

A server is a powerful computing device or program used for managing network resources. A server can be connected on the same computer or locally connected to other computer devices or through a remote connection. There are different types of servers you can create, for example, a database server, printer server, and network server amongst others.

```
import socket
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(),1234))#port number can be anything between 0-65535

s.listen(5)

while True:
    clt,adr=s.accept()
    print(f"Connection to {adr}established")
    #f string is literal string prefixed with f which
    #contains python expressions inside braces
    clt.send(bytes("Socket Programming in Python","utf-8 "))
    #to send info to clientsocket
```

Example 1: program to connect to server

From the above example, to create a socket, you have to first import the module. Then, use the `socket.socket` function to create a server socket.

The `listen ()` function allows connection to the server, with parameter 5 being the queue to multiple simultaneous connections. You can pass any parameter to the function with the

Minimum number being 0. If no parameter is specified, a default parameter zero is used.

The while loop accept all connections to the server. `clt` and `adr` represent client object and address respectively. The print statement only prints the address and the port number of the connected device on the client socket. The `clt.send ()` function is used to send clients data in bytes.

```

#connecting to google website
import socket
import sys
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print "successful creation of the socket"
except socket.error as err:
    print "socket creation failed: error message %s" %(err)

# default port for socket
port = 80

try:
    host_ip = socket.gethostbyname('www.google.com')
except socket.gaierror: # throw exception to show it could not resolve the host

    print "there was an error incurred when resolving the host"
    sys.exit()

# connecting to the server
s.connect((host_ip, port))

print "Socket connection to Google is successful\
on port == %s" %(host_ip)

```

Example 2: connecting to google website

// output

Successful creation of socket

Socket connection to google is successful on port ==173.194.40.19

Sometimes a socket error occurs when connecting to the server. In this case, a socket error is thrown, and for you to connect to the server, you have to use the ip address of the server.

To get the server ip address, you can type the following code in the command prompt:

\$ping www.google.com

or use the python program to obtain the ip address.

Import socket ip=socket.gethostbyname('www.google.com') print ip
connection to client device

A client device is the computer that receives information or services from the server machine. In a client-server architecture, the client requests for services or resources from the server and the

Server provides the services to the client. For example, use your web browser like google chrome or mozilla firefox to request a particular webpage from the web server.

```
import socket
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 2346))
msg=s.recv(1024)
print(msg.decode("utf-8"))
```

Example: client-side requests

This code imports a socket module and then creates the socket. *connect()* creates a connection between the client and server machine by specifying the hostname and the port number.

The *recv()* method is used by the client to receive information from the server. The information is stored in a different variable message. The message is transferred in bytes.

Note: *gethostname()* method is used where the client and server are on the same machine. Example: python socket server connection using a local host ip address

```

import socket
serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serv.bind(('0.0.0.0', 8080))
serv.listen(8)
while True:
    conn, addr = serv.accept()
    from_client = ''
    while True:
        data = conn.recv(4096)
        if not data: break
        from_client += data
        print from_client
        conn.send("SERVER CONNECTION\n")
    conn.close()
    print 'Disconnection of client device'

```

This program binds the socket object to the local host using the port number 8080 which acts as the socket server. When the client device connects to the server using the specified address, the server will listen for the data requested and store it in the variable named 'data'.

The program then logs the client data using the 'print' output operator, which sends a message string to the client 'server connection'.

on the client side:

```

import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('0.0.0.0', 8080))
client.send("CLIENT device\n")
from_server = client.recv(4096)
client.close()
print from_server

```

The client socket connection code establishes a socket connection with the server. This only happens when the server program is currently running.

Execution of client server programs

To execute client server program, open a command prompt and navigate to where you have created the client server program then type the following

command:

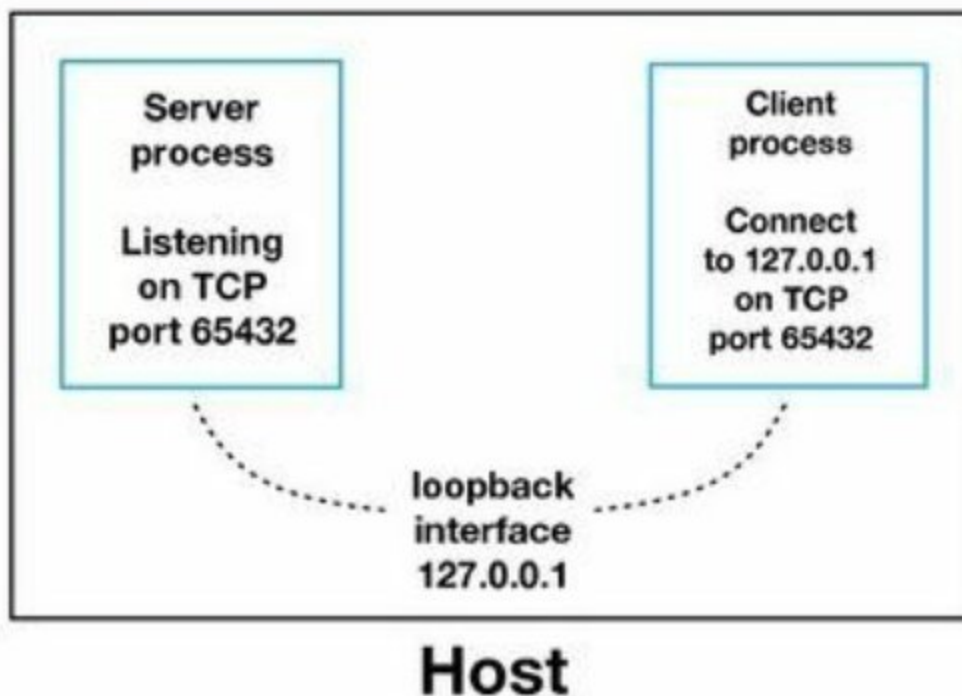
Py server.py

Server.py is the file name of the server. once the server is running, you can execute a client program. This is done by typing the following command in a new command prompt window:

Py client.py where the client.py is the filename of the client program.

When the client device calls for *connect()* to establish a connection with the server, it initiates a three-way handshake. The three-way handshake is very important in socket communication as it ensures the network connection is reachable from both sides. That is, the client can reach the server and vice versa. The client and server exchange data using *send()* and *recv()* methods.

communication breakdown



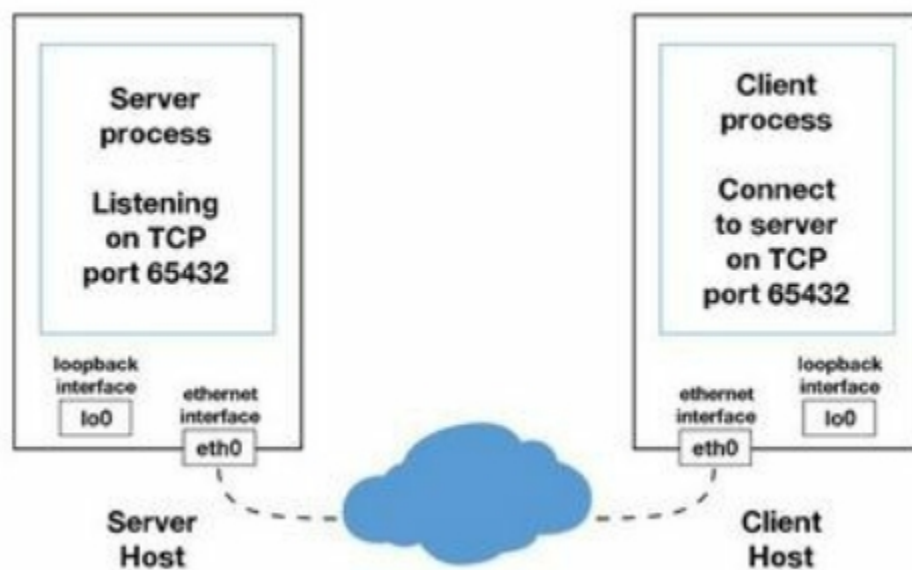
The client and the server communicate with each other in the following ways:

If you're using the loopback interface 127.0.0.1 (ipv4 address or the ipv6 address::1), then the data will not go beyond the host or touch the external network. The interface is always within the host.

Application programs rely on the loopback interface to communicate with other programs that run on the host. This improves application security and isolation via the external network. Since the communication happens within the host, there is no threat from external sources.

This type of communication is applied in an application server with its own private database. The database is configured to be used by other servers or configured to listen for all connections on a loopback interface only. Any other hosts on the network can't connect to it.

If another ip address is used other than 127.0.0.1 on the network, it will be bound on an ethernet interface connected to an external network. This allows you to connect to other hosts outside the localhost network.



Multiple communications

```
Import socket
```

```
S= socket.socket( socket. AF_INET, Socket.SOCK_STREAM)
```

```
S.connect((socket gethostname (), 2346))
```

```
While true;
```

```
Msg= s.recv(6)
```

```
Print (msg.decode("utf-8"))
```

For the communication to go until the client receives all information, a while loop is used.

In the above program, the message will receive six bytes at each transfer. The program is not terminated at the end since the actual file size transferred between the client and server is unknown. To ensure the program terminates at the end, you can add the following code to the client side.

```

Complete_info=""

while True:

    msg=s.recv(6)

    if len(msg)<=0:

        break

    complete_info +=msg.decode("utf-8")

    print (complete_info)

```

on the server side, the *close()* method is used. *ctl.close()*

The above block code check the size of the information and print it in a buffer of two bytes at a time. When the information transfer is completed, the connection is terminated.

Transferring python objects

Just like other programming languages, python socket programming enables you to transfer objects such as tuples, data sets, and dictionaries. Transfer of objects is achieved through the python pickle method.

The pickle method is used during serialization or deserialization of objects in python. Example:

```

Import pickle

```

```

Example=[1, 2, 'abc']

```

```

Msg=pickle.dumps (example)

```

Print (msg)

```
b'x80x03] qx00 (kx01x02x00x00x00abcqx01e'
```

// output

The example variable is serialized using the *dump()* function. The output of this program starts with 'b' indicating the program is converted to bytes. You can also implement the pickle module when you want to transfer objects between clients and the server.

How to transfer objects

The pickle module can allow you to transfer anything via the network. The program below will enable you to transfer information from the server side to the client side device.

```
import socket
import pickle

x=8
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), 2133)) #binding tuple
s.listen(7)
while True:
    clt , adr = s.accept()
    print(f"Connection to {adr}established")

    y={1:"Client", 2:"Server"}
    mymsg = pickle.dumps(y) #the msg to be printed
    mymsg = {len(mymsg):{x}}"utf-8") + mymsg
    clt.send(mymsg)
```

on the server side:

In this server-side program, variable *y* acts as the python object that is sent from the server to the client machine. To send the message down to the client, the object is serialized using *dumps ()* and then converted to bytes.

client-side program to receive data:

```
import socket
import pickle
x=8
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 2133))

while True:
    complete_info = b''
    rec_msg = True
    while True:
        mymsg = s.recv(8)
        if rec_msg:
            print(f"The length of message = {mymsg[:x]}")
            z = int (mymsg[:x] )
            rec_msg = False
            complete_info += mymsg
            if len(complete_info)-x == z:
                print("Recieved complete information")
                print(complete_info[x:])
                y = pickle.loads(complete_info[x:])
                print(y)
                rec_msg = True
    complete_info = b''
    print(complete_info)
```

// output

Sending objects using socket programming in python

The first while loop will help you keep track of the information sent—whether it is complete or incomplete using `complete_info`—as well as know the length of the message received using `rec_msg`. As the message is being received, it is printed in a buffer of 8 bytes. You can choose any size you want to the buffer to be.

When the message is fully received, print received complete information then deserialize the message using the `loads()` method.

Why you should use sockets to send data

All internet based applications operate in real-time, thus a great need to implement sockets programming in the networking code. Some of the apps

that rely on socket programming include:

1. chatting apps like whatsapp, slack, and wechat.
2. Web pages with live notifications like facebook, twitter, ebay, etc.
3. Iot devices like nest and august locks.
4. Real-time data dashboards like use of coinbase web page.
5. A multiplayer online game.

Python programming uses synchronous execution of programs; hence it was built to be more robust and support socket programming.

Sockets allow data to be streamed (sent and received) at any time.

Python provides a socket class that makes it easy for the programmers to implement a socket object.

cgi programming

Python programming is used in the implementation of common gateway interface (cgi) through the hyper-text transfer protocol (http). It has a set of standards that defines how information is exchanged between a web user and an application program, that is, passing of information between a webserver and a custom script.

If a user requests for information from a certain web page, the web server sends back the requested information into the web page. The server device or program passes information into all application programs that process data before sending back an acknowledgement message. Passing information back and forth between the server and the application forms a common gateway interface.

The http server invokes a cgi script to allow processing of user inputs submitted via the html <form> element. When a client makes a request, the server creates a script shell environment where it stores information about the request, the client's hostname, requested url, the message query string, and any other information requested. The server then executes the script and sends the output results back to the client.

The input script is connected to the client and the server reads data from the input form this way or pass the data submitted via the form through a query string which acts as part of the url.

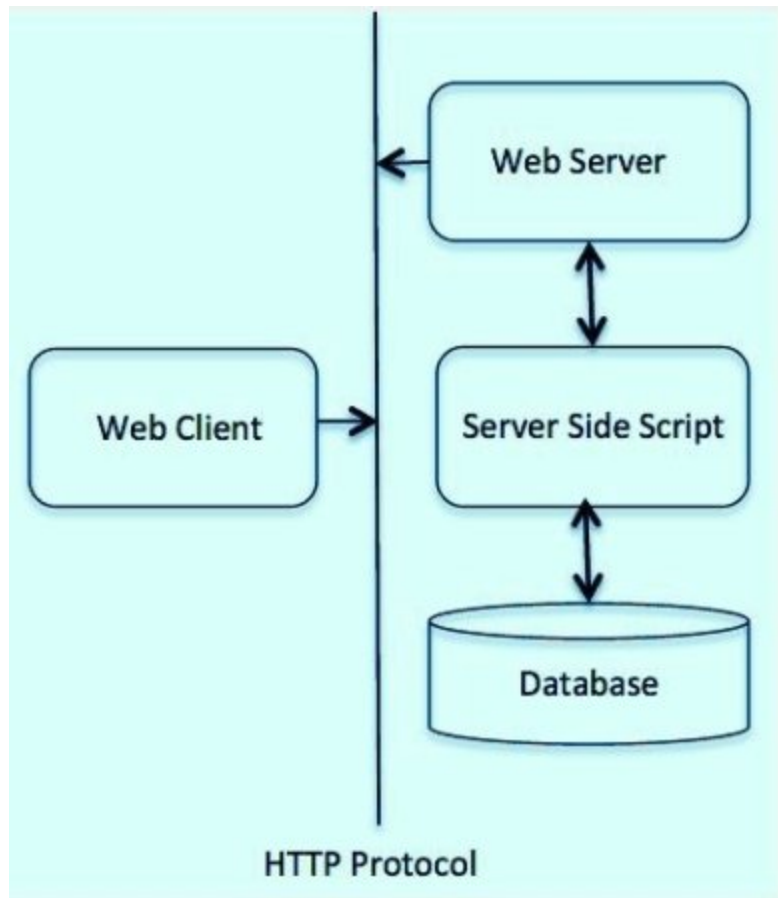
The cgi programs are dynamically written and generate user inputs that respond to input data or a web page that interacts with the server software.

How information passing works

1. When a user opens a web browser and click on a hyperlink to a particular webpage or its uniform resource locator (url),
2. The browser contacts the http server and demand for the url.
3. The web server parses the url of the requested page and look for the specific filename of the requested information.
4. If the file is found, the request is sent back.
5. The web browser accepts the response back from the web server.
6. The response is either the received file or an error message to alert the user the file was not found.

When you request for a specific directory, the http server executes the program and displays the output in your browser instead of sending back the file.

cgi configuration



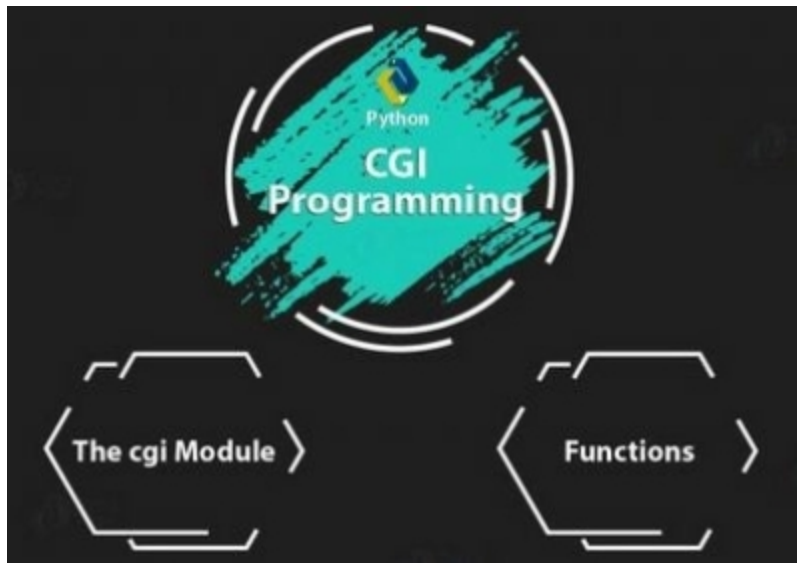
The python cgi architecture works in the following ways:

1. Requires you to specify the user running the web server
2. Analyzes the server configuration and find out if it is possible to run the scripts in a particular directory.
3. Analyzes file permission rights.
4. Makes sure the written scripts are clear, easily readable, and executable by the user of the web browser.
5. Ensures the python script's first line represent the web server run by the interpreter.

Before you configure the cgi program, ensure the web server supports the cgi program and it is well configured. cgi programs executed in the http server are all in a pre-configured device. The files are stored with an extension *.cgi*, but you can also store the files with the extension *.py*.

Python cgi program structure

The cgi scripts output has two parts: modules and functions. A black line separates each python script.



The cgi module

The python cgi module helps debug scripts and find solutions to various situations. The module can also help you to upload a file from a form. The script output results into two parts separated by a black line. The module consists of a number of headers to define what kind of data is following to the client. The header part instructs the clients to expect a certain type of data.

To use the module, you have to import it to your program.

```
>>>import cgi
```

once you import the module, you can activate an exception handler to catch errors on the web browser. The exception will give a detailed report in the web browser in an event of error.

```
>>> import cgi
```

```
>>> cgi.enable()
```

If you want to store the report in the file, you have to add the filename to store the report and specify the file path.

```
>>> cgi.enable (display=0, logdir='path to the directory')
```

```
print ("Content-Type : text/html")

print ("<html>")
print ("<head>")
print ("<title>My First CGI-Program </title>")
print ("<head>")
print ("<body>")
print ("<h3>This is HTML's Body section </h3>")
print ("</body>")
print ("</html>")
```

cgi header section

You can save the header section as header.py and run the code. The output of this program will be:

This is html's body section

The content-type: text/html printed on the first line specifies the type of content to be displayed by the browser. The above program is a basic cgi program using the html tags. The written scripts can interact with external systems like rdbms to exchange information.

All http header section should be in the form of:

Http field name: field content

```

<!doctype html>
<html>
<head>
<title>Example of python web file</title>
</head>
<body>
<form name="pytform" method="POST" action="/cgi-bin/form.py">
  <input type="text" name="firstname" />
  <input type="submit" name="submit" value="Submit" />
</form>
</body>
</html>

```

Example: html form with python cgi-bin folder

This script points to the following cgi program.

```

#!/usr/local/bin/python3.6
print("Content-Type: text/html")
print()

import cgi,cgitb
cgitb.enable() #for debugging
form = cgi.FieldStorage()
name = form.getvalue('firstname')
print("Name of the user is:",name)

```

The `#!/usr/local/bin/python3.6` indicates the location of python installation. Then, import the `cgi` and `cgitb` modules to handle any exception. `cgitb.enable()` helps report errors.

The `form` variable is used to store the `cgi` class `fieldstorage`, thus, making the `form` function as a data dictionary. The `getvalue()` function extracts the name supplied by the user in the input html form. Use the `print()` to display the name of the user.

components of http header

This table consists of common http headers frequently used in cgi programming.

Value	Http header	Description
Text/tml	content type	This is a mime string used to define the file return type format.
		This shows the expiry date/time. It indicates specific date when information will be invalid.
Date	Expires	A browser uses this header information to determine when to refresh a browser. A valid date string is in the form of 01 jan 2020 08:00:00 gmt
Url	Location	This returns the url of the requested file. The url redirects to any file requests.
String	Set-cookie	A unique field that sets passed cookies through the string.
		This header returns the file or data length inform of bytes.
N	content-length	This format uses the length value to determine the estimated download time of the file.
Date	Last-modified	It defines the last date the resources were modified.

cgi environment variables

Environment Variables	Description
CONTENT_TYPE	describes the data-type of the content
HTTP_COOKIE	returns the visitor's cookie if one is set
CONTENT_LENGTH	It is available for POST request to define the length of query information
HTTP_USER_AGENT	defines the browser type of the visitor
PATH_INFO	defines the path of a CGI script
REMOTE_HOST	defines the host-name of the visitor
REMOTE_ADDR	defines the IP Address of the visitor
REQUEST_METHOD	used to make request & the most common methods are - GET and POST

cgi programs have access to the environment variable listed below. These variables are very important when writing a cgi program.

Example:

```
#!/usr/bin/python

import os

print "Content-type: text/html;
print "<font size=+5> CGI Environment Variable</font><\br>";
for param in os.environ.keys():
    print "<b>%20s</b>: %s<\br>" % (param, os.environ[param])
```

chapter summary

In this chapter, you learned about networks and client/server programming using sockets. A python program is able to provide a network access through both low- and high-level network access. You also learned how:

- The lower level access provides you with basic support to socket programming as well as allow you to implement both client and server network connections.

- The high level access is implemented through the http,ftp, and tcp protocols.
- A socket in one of the connected devices listens to a specific port with the ip address while the other socket forms another connection. Sockets in python programming are designed for sending and receiving data between various devices with each socket having an ip address and a port number.
- The various port numbers and protocols form a socket module.
- The different classes for handling transport layer and provide an interface to connect to other devices.
- To import the socket module and how to connect to the server
- You can connect to google website
- connect to a client device
- communication takes place between the client server architecture.
- The importance of using sockets in sending data between devices in real-time.
- Basics of cgi programming

FINAL WoRDS

Python programming is an object oriented programming language that allows you to design, develop, and implement robust programs. The programs use algorithms and information processing tools to develop applications. Developing any program to solve a certain problem requires that you first define the problem. once you're able to define the program, you can come up with the program needs in terms of inputs and processes, which need to be followed in order to come up with the solution to the problem.

Major programming languages use algorithms when coming up with solutions to solve problems. An algorithm is a set of instructions that must be followed in order to perform a particular task or solve certain problem. In computer programming, algorithms act as recipes that describe the procedures necessary to achieve a set goal. You need to provide the input data taken by your computer system for processing. Following these predetermined procedures will result in a particular output. Algorithms in python programming are intended to instruct python interpreters what needs to be done, and they do exactly that.

Programs designed in python can be represented in the form of flow charts or use of english-like statements known as pseudocode. Flow charts represent data in the form of inputs, processes, and outputs. Inputs involve the use of computer hardware and software to enter data into the computer.

computer software is the term for programs designed to perform various tasks. These programs are classified into applications software, system software, and programming languages. In this chapter, we have focused on the use of programming languages, in particular, the python programming language, to develop and implement programs.

We learned how to use the python interactive shell and how to execute a simple python script. In our second chapter, we learned how to use python strings. Strings are a built-in data type. We have learned how to create a string of variables and how to access a single character from that string. Another important feature you learned is string slicing, which allows you to

return sub- parts of a string in a certain sequence using built-in methods to manipulate strings.

Data sequences is another built-in data type that consists of a series of values held together by a container. A sequence ensures various objects and values are organized in an efficient manner. It consists of strings, lists, tuples, xrange objects, use of buffers, and byte arrays. As discussed in the previous chapter, strings consists of an array of elements whereas a list is a group of elements organized in a specific order.

Lists are essential in the implementation of sequence protocol. It enables you to easily add and remove elements from a sequence. Tuples work similar to lists except that they're immutable. You can't add, delete, or change elements created using tuples. Tuples prevents you from making any accidental changes to your program as well as allow for fast execution of lists.

You also learned how to use string concatenation, check for string membership sequence, and

Slicing. You learned various sequence methods used in manipulating data like *len()*, *count()*, and

Index(). You are also now able to create sets and manipulate them using set functions.

The python dictionary allows you to add, change, or remove elements from a dictionary. Dictionary use a key and a value to make calls to a function. Elements in a dictionary are organized to retrieve data elements using a key. Learn how to add elements to the dictionary and also remove elements from a dictionary.

In chapter four, we learned how to use the math module to perform mathematical calculations. The math module defines two common constants used in calculations: *pi* and *euler's number*. Python properties are accessed via *math.pi* while the functions. other constants used in python include *inf* and *nan*.

Python math functions support different classes in its standard library. These classes include: numbers and numeric representations, power and logic functions, and trigonometric and angular conversion functions. Each of these classes have functions which can perform a specific tasks.

Python random module uses pseudo-random number generator tool to generate random numbers for use. The generated numbers are in the range of 0.0 to 1.0. Some of the functions used in generating random numbers include *randint()*, *randrange()*, *sample()*, *shuffle ()*, and *choice()*. In this chapter, we discussed how to apply each of the random functions to generate random integer numbers. We also discussed how you can apply *randint ()* to determine a lucky draw in a game.

A python *seed ()* function is used to get random numbers. Then, the seed number is passed to a pseudorandom algorithm that generates random sequence bytes. The pseudorandom generator rely on previously generated values. The *seed()* initiates the pseudorandom number generator. It uses the seeding value as the base when generating random numbers. If the seeding value is unavailable, then it automatically uses the current date.

The seed value always indicates the previous value generated by the pseudorandom number generator. It also saves the state of the random function in order to enable multiple execution of random numbers.

The generated random is not secure; you can make the random numbers generated secure by implementing a cryptography to securely generate pseudorandom numbers. This helps in making your data more secure and prevent any unauthorized access to data.

We also discussed how to retrieve the current system state and how to restore the previous system status using *getstate ()* and *setstate ()*, respectively. These functions determine the current internal state of a random number generator, and this information is very important when generating a sequence of data. With a program example, we saw how to record the current internal state of a random number as well as restore the internal state to its previous state using *random.setstate ()*.

We are also able to generate numpy.random numbers using arrays and random numbers using unique ids.

The date and time module helps in determining the date and time of the day. The date time module has various functions with a wide range of information on the date object. From the date and time module, we are able to define a datetime class inside the datetime module, and then use

The *now()* function to create an object with the current date and time.

The python module keeps track of date and time in different ways. The datetime module keeps track of the dates and time as well as converting the dates formats.

The datetime module allow you to manipulate the date and time using the date class, a timestamp and using the time class. other methods used in date manipulation include the datetime class, *strftime()* to format date objects into string, and *strptime()* to change the datetime format. The method creates a datetime object from the date and time string.

Timedelta objects are used to predict past and future dates. A timedelta object calculates the past and future dates but doesn't print the date or time. You can also work with different time zones in your program by calling the pytz module. Time module is also very important when working with time in python programs. The time module enables us to determine the number of days elapsed since the epoch era.

Python manages time in floating point numbers (seconds). From our discussion, you can use the time module to determine the number of seconds elapsed since epoch era. You can also determine the current date and time based on specific format. Time can be represented based on your physical location and it's called local time. The concept of time representation forms time zone.

A time zone allows us to conform to specific standardized time within our religion and it is defined using the coordinated universal time, utc. Utc is a

standard for synchronizing and coordinating the world's time.

Python time can also be represented using data structures like tuples. We are able to discuss with examples how you can use tuples to represent time. You are also able to learn how to use python time as an object with an example, how to convert time from seconds to objects, and how to convert local time to string.

File processing is another important feature in python programming. In this chapter, you learned about various components of a file, how to use the file path to determine the file storage location, and about various file operation processes.

You learned how to open a python program and use various file operation modes to manipulate a file. You can open buffered binary files the same way you open the normal files for manipulation, except that the file object either returns a `bufferedReader` or `bufferedWriter`. You also learned how to close files and free up memory resources.

When writing to a file, you can use 'w' when writing a character at a time or `write()` when writing a string or block of statements. Similarly, the 'r' mode is used in reading a single character in a file while the `read()` reads a string of statements. You can specify the number of characters you need to access by modifying the read function with `read(size)`.

File i/o operations provide you with built-in functions to manipulate files in python. The file i/o operations are classified into text i/o, binary i/o, and raw i/o. Each of these categories has a concrete object or stream. And each of the stream objects can either be a read-only, write-only, or read-write option.

A text i/o stream sends and receives string objects while a binary i/o stream accepts bytes-like

objects and generates byte objects applied on non-text data. Raw i/o is a low-level building block for both binary and text streams.

The python print function is used to display output data on a standard output device like a display unit or printer.

Python also allows you to import the os and os.path modules, which have a wide range of functions to handle file operations and perform other functions of an operating system. The os module can allow you to create a directory, rename or remove directory among other functions.

Sometimes, when making a call to an unavailable or invalid file, an error message is generated. This error message can be handled by exceptions. An exception is any interruption of the normal program flow caused by a certain condition or error. Whenever an error occurs, an exception is thrown to catch the error. If the error is not handled immediately, it will result in the termination or exit of the program.

Exceptions can occur due to various errors as discussed, and by knowing what kind of error you may expect in your program, you can easily raise an exception. Exceptions can also be handled using the try...except statements, else clauses, or try..finally clauses. When exceptions occur, they affect the flow control of the program and the control is taken to the exception block before resuming the normal execution of the program. We are also able to discuss different exception classes and how each class hierarchy is applied.

Images and graphics are essential in today's data driven world. A lot of modern applications rely on the use of images and graphics to pass information. Python provides a variety of libraries you can use to process images as well as manipulate image information.

There are various computer image processing algorithms that accepts image inputs and process these inputs to meaningful information.

To use images in python, you have to install standard library packages designed to handle image processing. Some of the common image processing python libraries include: opencv, scikit, numpy, scipy, and the python imaging library.

Based on your image processing needs, you can install the appropriate package. In this chapter, we installed pil to handle various image manipulation techniques

Graphics is an important tool in python programming when dealing with events. Importing the graphics library allows you to draw various images and manipulate graphic images as well as change the image color and type. You also saw how to use gui features to carry out various tasks.

The graphical user interface provides you with onscreen widgets you can interact with to create events. These events may result from either using labels to enter text, using textboxes, clicking command buttons, using list boxes, and menus.

When handling event driven programs, you can import python modules like tkinter and asyncio that allow you to perform various tasks.

In our last chapter, we are able to learn about network and client/server programming using sockets. A python program is able to provide a network access through both low level and high level network access. The lower level access provides you with basic support to socket

Programming as well as allow you to implement both client and server network connections.

Socket programming acts as the network backbone for communication. It makes the transfer of information between different devices and platforms possible. Sockets in python programming are designed for sending and receiving data between various devices, with each socket having an ip address and a port number.

The socket module provides various classes to handle connections between the devices and other programs. The various classes handle the transport layer and an interface for connecting devices.

To use the socket, you have to import it to your python program using `socket.socket()`. This allows you to import the library functions and ports used in creating client/server models and ensures that connecting devices are able to establish communication.

You are able to connect to the server machine, access a website, and also connect to a client device and execute client server programs. We were also able to discuss how communication takes place between the client and server.

You learned how python socket programming allows you to transfer objects through the pickle method. You also now know the reasons why you should use sockets to send data and better understand how some application programs like whatsapp, facebook, and twitter rely on sockets to exchange data.

You also know how information is exchanged between your device (mobile phone or computer) and the application program through cgi programming and how communication takes place.

REFERENCES

Image credit: edureka.co image credit: w3schools.com

Image credit: tutorialspoint.com image credit: realpython.com

<https://python-textbok.readthedocs.io/en/1.0/introduction.html>
<https://developers.google.com/edu/python/strings>
https://www.tutorialspoint.com/python/python_strings.htm <https://data-flair.training/blogs/python-sequence/>
<https://docs.python.org/2/library/sets.html>

<https://data-flair.training/blogs/python-set-and-booleans-with-examples/>
https://en.wikibooks.org/wiki/python_programming/sequences
<https://www.geeksforgeeks.org/python-dictionary/>
<https://www.tutorialsteacher.com/python/math-module>
<https://realpython.com/python-random/>
<https://www.tutorialsteacher.com/python/random-module>
<https://www.geeksforgeeks.org/random-seed-in-python/>
<https://pynative.com/python-random-module/>
<https://www.programiz.com/python-programming/datetime>
https://www.tutorialspoint.com/python/python_date_time.htm
<https://www.programiz.com/python-programming/time>
<https://realpython.com/python-time-module/>
<https://www.programiz.com/python-programming/file-operation>
<https://docs.python.org/3/library/io.html> <https://www.edureka.co/blog/os-module-in-python> <https://www.geeksforgeeks.org/os-path-module-python/>
<https://docs.python.org/3/library/exceptions.html>

<http://www.co-pylit.org/courses/cosc1336/simple-graphics/01-python-graphics.html> https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_eventdriven_pro
<https://www.edureka.co/blog/socket-programming-python/>

<https://realpython.com/python-sockets/>