Competency Based Learning Material (CBLM)

AI in Immersive Technology

Level-6

Module: Develop programs using Python

Code: OU-ICT-AIIT-02-L6-V1

National Skills Development Authority
Chief Advisor's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority
Chief Advisor's Office
Level: 10-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Email: ec@nsda.gov.bd
Website: www.nsda.gov.bd.
National Skills Portal: http:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on “Apply Math Skills for Machine Learning” under the AI in Immersive Technology, Level-6” qualification is developed based on the national competency standard approved by National Skills Development Authority (NSDA)

This document is to be used as a key reference point by the competency-based learning materials developers, teachers/trainers/assessors as a base on which to build instructional activities.

National Skills Development Authority (NSDA) is the owner of this document. Other interested parties must obtain written permission from NSDA for reproduction of information in any manner, in whole or in part, of this Competency Standard, in English or other language.

It serves as the document for providing training consistent with the requirements of industry in order to meet the qualification of individuals who graduated through the established standard via competency-based assessment for a relevant job.

This document has been developed by NSDA in association with industry representatives, academia, related specialist, trainer and related employee.

Public and private institutions may use the information contained in this CBLM for activities benefitting Bangladesh.

Approved by the Authority …………..meeting held on ....

How to use this Competency Based Learning Material (CBLM)

The module, Maintaining and enhancing professional & technical competency contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.

2. Read the **Information Sheets.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information Sheets** complete the questions in the **Self-Check.**

3. **Self-**Checks are found after each **Information Sheet**. **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information Sheet**. Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.

4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information Sheets. This is your opportunity to practise the job. You may need to practise the job or activity several times before you become competent.

5. Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

6. A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Module Content

Unit of Competency	**Develop programs using Python**
Unit Code	OU-ICT-AIIT-02-L6-V1
Module Title	**Develop programs using Python**
Module Descriptor	This module covers the knowledge, skills and attitude required to develop programs using python. It specifically includes the requirements of working with Python basics, applying input and output methods in python, solving problems associated with "loops", implementing string functions, basic I/O functions, and performing data analysis using Python.
Nominal Hours	20 Hours
Learning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs: 1. Work with Python basics 2. Apply input and output methods in python 3. Solve problems associated with "loops" 4. Implement string function 5. Implement basic I/O functions 6. Perform data analysis using Python

Assessment Criteria

1. Fundamentals of python are interpreted
2. Python environment is set up
3. Data types and variables are used
4. Numeric values are used
5. String variables are used
6. Printing with parameters is exercised
7. Getting inputs from users is exercised
8. String formatting is applied
9. Simple and complex decision making is applied using logical statements
10. Loops are interpreted
11. Problems associated with loops are exercised
12. Advanced data storage techniques are applied in python
13. String input methods are interpreted
14. String input methods are applied
15. Strings are manipulated

16. Built-in string functions are used
17. Opening and closing files are exercised
18. Modes of accessing files are exercised
19. Create, update and delete of a file is exercised
20. Types of data analysis is interpreted
21. Data analysis is exercised
22. Data visualization and explainability of data for decision making is exercised

Learning Outcome 1: Work with Python basic

Assessment Criteria	1. Fundamentals of python are interpreted 2. Python environment is set up 3. Data types and variables are used 4. Numeric values are used 5. String variables are used
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1 Interpret Fundamentals of python 2 Set up Python environment 3 Use of data types and variables 4 Use of numeric values 5 Use of string variables
Activities/job/Task	1. Create a python project using Gradient Descent Algorithm
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 1: Work with Python basic

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Work with Python basic'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Work with Python basic"	2. Read Information sheet 1: Work with Python basic 3. Answer Self-check 1: Work with Python basic 4. Check your answer with Answer key 1: Work with Python basic
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet-1.1: How to Define a class in Python? Task Sheet-1.2: How to take input from a user in python Task Sheet-1.3: How to perform arithmetic operations in python? Job Sheet-1: Develop a bank account management system using python

Information Sheet 1: Work with Python basic

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

1.1 Interpret fundamentals of python
1.2 Set up Python environment
1.3 Use Data types and variables
1.4 Use Numeric values
1.5 Use String variables

1.1 Fundamentals of python

a. Python programming

Python is a widely used general-purpose, high-level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Who invented Python?

In the late 1980s, history was about to be written. It was that time when working on Python started. Soon after that, Guido Van Rossum began doing its application-based work in December of 1989 at Centrum Wiskunde & Informatica (CWI) which is situated in the Netherlands. It was started as a hobby project because he was looking for an interesting project to keep him occupied during Christmas.

The programming language in which Python is said to have succeeded is ABC Programming Language, which had interfacing with the Amoeba Operating System and had the feature of exception handling. He had already helped create ABC earlier in his career and had seen some issues with ABC but liked most of the features. After that what he did was very clever. He had taken the syntax of ABC, and some of its good features. It came with a lot of complaints too, so he fixed those issues completely and created a good scripting language that had removed all the flaws. The inspiration for the name came from the BBC's TV Show – '**Monty Python's Flying Circus'**, as he was a big fan of the TV show and also he wanted a short, unique and slightly mysterious name for his invention and hence he named it Python! He was the "Benevolent dictator for life" (BDFL) until he stepped down from the position as the leader on 12th July 2018. For quite some time he used to work for Google, but currently, he is working at Dropbox.

b. Coding System

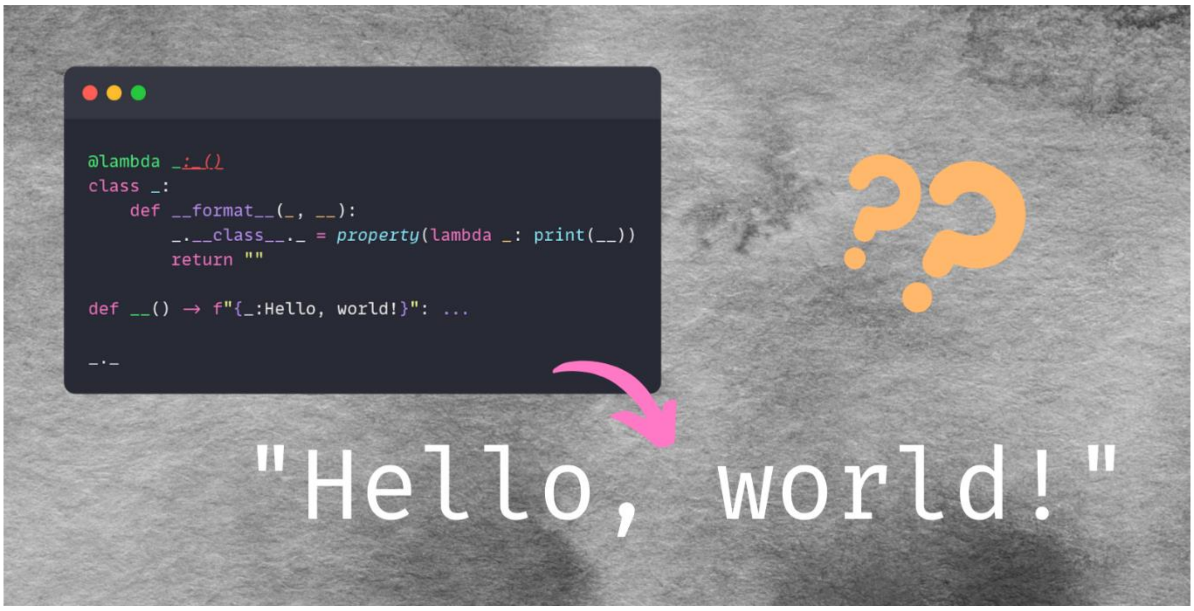
Many operating systems, including macOS and Linux, come with Python preinstalled. The version of Python that comes with your operating system is called the system Python. The system Python is used by your operating system and is usually out of date.

c. Basic Python Code

Write a Python Program

If you don't already have IDLE open, then go ahead and open it. There are two main windows that you'll work with in IDLE: **the interactive window**, which is the one that opens when you start IDLE, and the **editor window**.

You can type code into both the interactive window and the editor window. The difference between the two windows is in how they execute code. In this section, you'll learn how to execute Python code in both windows.



```
@lambda _:_()
class _:
    def __format__(_, __):
        _.__class__._ = property(lambda _: print(__))
        return ""

def __() → f"{_:Hello, world!}": ...

_._
```

The Interactive Window

IDLE’s interactive window contains a **Python shell**, which is a textual user interface used to interact with the Python language. You can type a bit of Python code into the interactive window and press Enter to immediately see the results. Hence the name interactive window.

The interactive window opens automatically when you start IDLE. You’ll see the following text, with some minor differences depending on your setup, displayed at the top of the window:

```
Python 3.9.0 (tags/v3.9.0:1b293b6)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information. >>>
```

This text shows the version of Python that IDLE is running. You can also see information about your operating system and some commands you can use to get help and view information about Python.

The >>> symbol in the last line is called the prompt. This is where you’ll type in your code.

Go ahead and type 1 + 1 at the prompt and press Enter :

```
>>> 1 + 1
>>> 2
```

Python evaluates the expression, displays the result (2), then displays another prompt. Every time you run some code in the interactive window, a new prompt appears directly below the result. Executing Python in the interactive window can be described as a loop with three steps:

- Python reads the code entered at the prompt.
- Python evaluates the code.
- Python prints the result and waits for more input.

This loop is commonly referred to as a read-evaluate-print loop and is abbreviated as REPL. Python programmers sometimes refer to the Python shell as the Python **REPL**, or just “the REPL” for short.

Let’s try something a little more interesting than adding numbers. A rite of passage for every programmer is writing a program that prints the phrase “Hello, World” on the screen.

At the prompt in the interactive window, type the word print followed by a set of parentheses with the text "Hello, World" inside:

```
>>> print("Hello, World")
Hello, World
```

A function is code that performs some task and can be invoked by a name. The above code invokes, or calls, the print() function with the text "Hello, World" as input.

The parentheses tell Python to call the print() function. They also enclose everything that gets sent to the function as input. The quotation marks indicate that "Hello, World" really is text and not something else.

IDLE highlights parts of your code in different colors as you type to make it easier for you to identify the different parts. By default, functions are highlighted in purple and text is highlighted in green.

The interactive window executes a single line of code at a time. This is useful for trying out small code examples and exploring the Python language, but it has a major limitation: you have to enter your code one line at a time!

Alternatively, you can save Python code in a text file and execute all of the code in the file to run an entire program.

d. Writing Comment

How to Write a Comment The most common way to write a comment is to begin a new line in your code with the # character. When you run your code, Python ignore lines starting with #.

Comments that start on a new line are called **block comments**. You can also write **inline comments**, which are comments that appear on the same line as the code they reference. Just put a # at the end of the line of code, followed by the text in your comment. Here’s an example of a program with both kinds of comments:

```
# This is a block comment.
greeting = "Hello, World"
print(greeting) # This is an inline comment.
```

Of course, you can still use the # symbol inside a string. For instance, Python won’t mistake the following for the start of a comment:

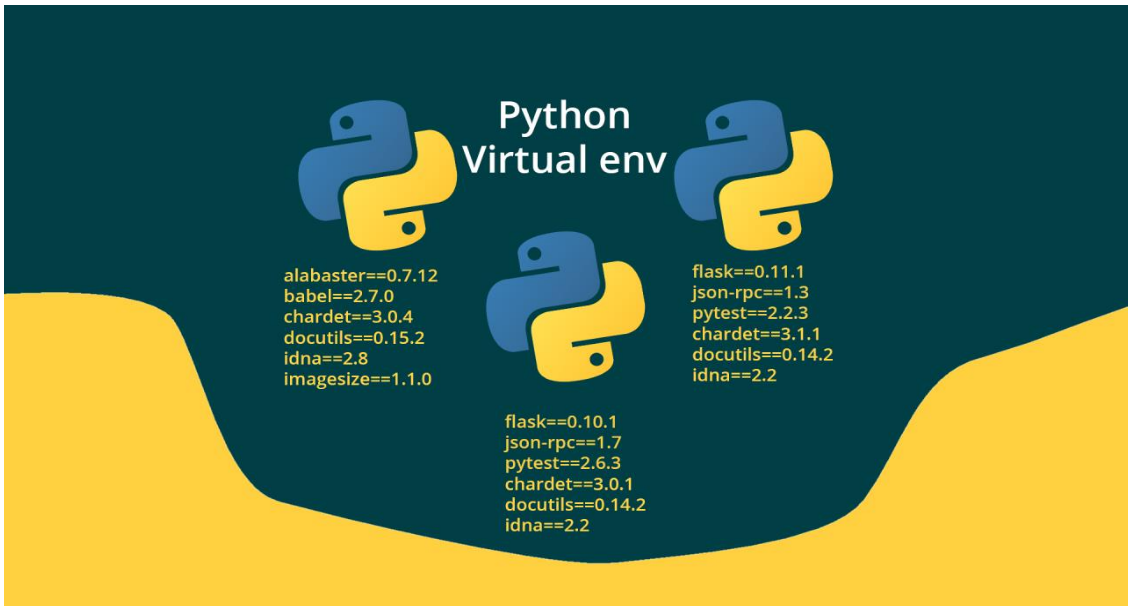
```
>>> print("#1")
#1
```

In general, it’s a good idea to keep comments as short as possible, but sometimes you need to write more than reasonably fits on a single line. In that case, you can continue your comment on a new line that also begins with the # symbol:

```
# This is my first program.
# It prints the phrase "Hello, World"
# The comments are longer than the code!
greeting = "Hello, World"
print(greeting)
```

You can also use comments to **comment out** code while you’re testing a program. Putting a # at the beginning of a line of code lets you run your program as if that line of code didn’t exist, but it doesn’t actually delete the code.

1.2 Python Environment Setup



a. Windows

Follow these steps to install Python 3 and open IDLE on Windows.

The code in this book is tested only against Python installed as described in this section. Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python
Windows doesn’t typically come with a system Python. Fortunately, installation involves little more than downloading and running the Python installer from the Python.org website.

- **Download the Python 3 Installer**
  Open a web browser and navigate to the following URL:
  https://www.python.org/downloads/windows/
- Click Latest Python 3 Release - Python 3.x.x located beneath the “Python Releases for Windows” heading near the top of the page. As of this writing, the latest version was Python 3.9. Then scroll to the bottom and click Windows x86-64 executable installer to start the download.

If your system has a 32-bit processor, then you should choose the 32-bit installer. If you aren’t sure if your computer is 32-bit or 64-bit, stick with the 64-bit installer mentioned above.

- **Run the Installer**
  Open your Downloads folder in Windows Explorer and double-click the file to run the installer. A dialog that looks like the following one will appear: It’s okay if the Python version you see is greater than 3.9.0 as long as the version is not less than 3.

Make sure you select the box that says Add Python 3.x to PATH. If you install Python without selecting this box, then you can run the installer again and select it.

Click Install Now to install Python 3. Wait for the installation to finish, then continue to open IDLE.

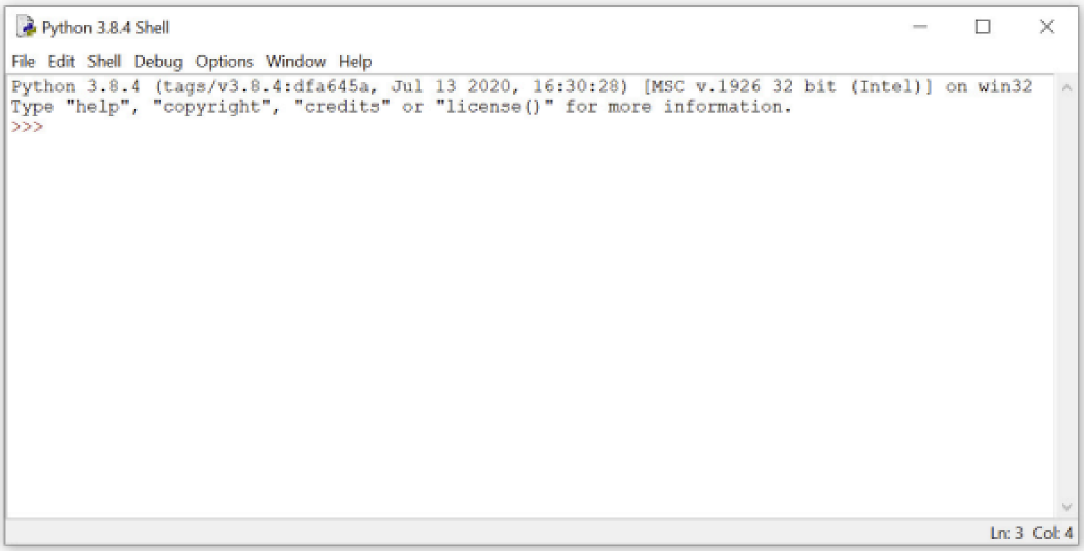
Open IDLE

You can open IDLE in two steps:

- Click the Start menu and locate the Python 3.9 folder.
- Open the folder and select IDLE (Python 3.9).

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It’s a great way to get started with Python!

The Python shell window looks like this:



```
Python 3.8.4 (tags/v3.8.4:dfa645a, Jul 13 2020, 16:30:28) [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.
The >>> symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

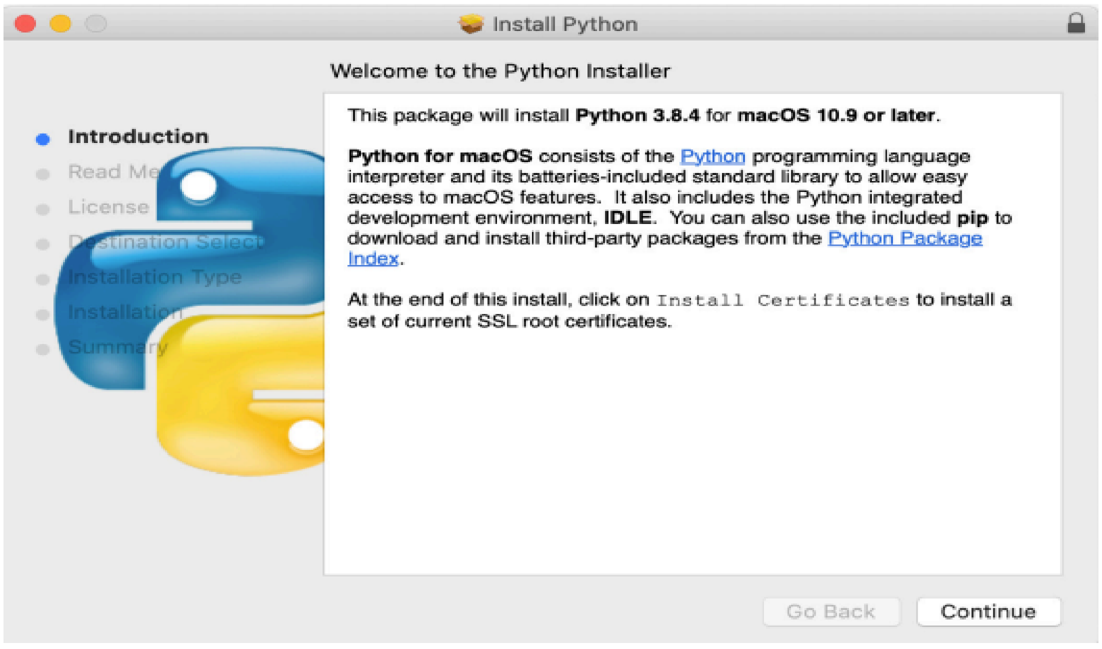
b. macOS

Follow these steps to install Python 3 and open IDLE on macOS.

The code in this book is tested only against Python installed as described in this section. Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.
Install Python To install the latest version of Python 3 on macOS, download and run the official installer from the **https://www.python.org/** website.

Step 1: Download the Python 3 Installer
Open a web browser and navigate to the following URL:
https://www.python.org/downloads/mac-osx/
Click Latest Python 3 Release - Python 3.x.x located beneath the “Python Releases for Mac OS X” heading near the top of the page. As of this writing, the latest version was Python 3.9.
Then scroll to the bottom of the page and click macOS 64-bit installer to start the download.

Step 2: Run the Installer
Open Finder and double-click the downloaded file to run the installer. A dialog box that looks like the following will appear:



Welcome to the Python Installer

This package will install **Python 3.8.4** for **macOS 10.9 or later**.

Python for macOS consists of the Python programming language interpreter and its batteries-included standard library to allow easy access to macOS features. It also includes the Python integrated development environment, **IDLE**. You can also use the included **pip** to download and install third-party packages from the Python Package Index.

At the end of this install, click on `Install Certificates` to install a set of current SSL root certificates.

Press Continue a few times until you are asked to agree to the software license agreement. Then click Agree . You'll be shown a window that tells you where Python will be installed and how much space it will take. You most likely don't want to change the default location, so go ahead and click Install to start the installation.

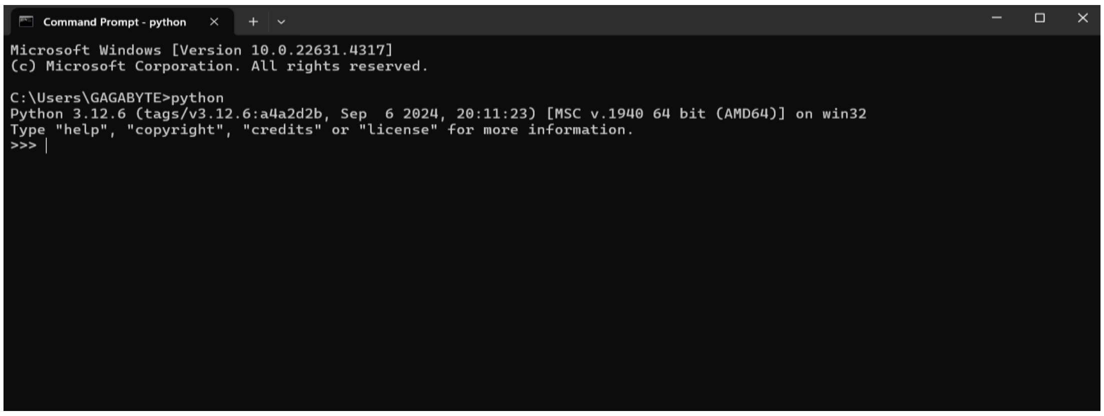
When the installer is finished copying files, click Close to close the installer window.

Open IDLE

You can open IDLE in three steps:

- Open Finder and click Applications.
- Double-click the Python 3.9 folder.
- Double-click the IDLE icon.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It's a great way to get started with Python!
The Python shell window looks like this:



```
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\GAGABYTE>python
Python 3.12.6 (tags/v3.12.6:a4a2d2b, Sep  6 2024, 20:11:23) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.
The >>> symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

.

c. Ubuntu Linux

Follow these steps to install Python 3 and open IDLE on Ubuntu Linux.

The code in this book is tested only against Python installed as described in this section. Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python

There's a good chance that your Ubuntu distribution already has Python installed, but it probably won't be the latest version, and it may be Python 2 instead of Python 3. To find out what version(s) you have, open a terminal window and try the following commands:

```
$ python --version
$ python3 --version
```

One or more of these commands should respond with a version, as below:

```
$ python3 --version
Python 3.9.0
```

Your version number may vary. If the version shown is Python 2.x or a version of Python 3 that is less than 3.9, then you want to install the latest version. How you install Python on Ubuntu depends on which version of Ubuntu you're running. You can determine your local Ubuntu version by running the following command:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 18.04.1 LTS
Release: 18.04
Codename: bionic
```

Look at the version number next to Release in the console output, and follow the corresponding instructions below.

Ubuntu 18.04 or Greater

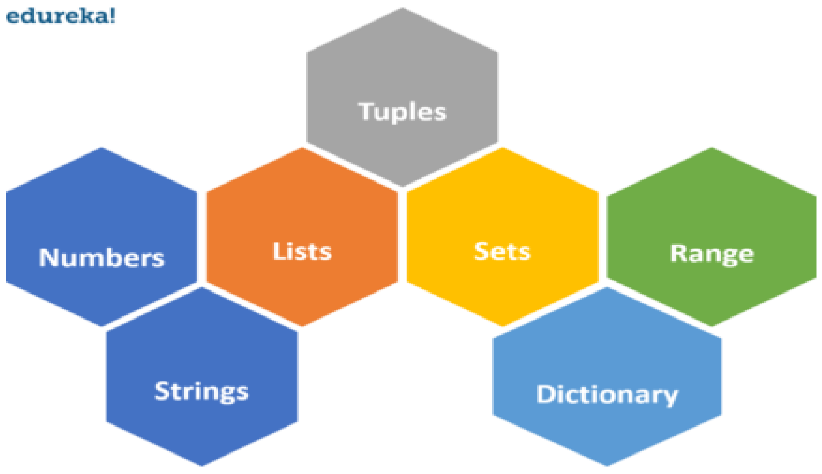
Ubuntu version 18.04 does not come with Python 3.9 by default, but it is in the Universe repository. You can install it with the following commands in the Terminal application:

```
$ sudo apt-get update
$ sudo apt-get install python3.9 idle-python3.9 python3-pip
```

1.3 Data types and Variables

What are Data Types in Python?

Data types in Python refer to classifying or categorizing data objects based on their characteristics and behavior. They define the type of values variables can hold and determine the performed operations on those values. Python has several built-in data types, including numeric types (int, float, complex), string (str), boolean (bool), and collection types (list, tuple, dict, set). Each data type has its own set of properties, methods, and behaviors that allow programmers to manipulate and process data effectively in their programs.



I. Built-in Data Types in Python

Built-in data types in Python are fundamental data structures provided by the Python programming language. They are pre-defined and available for use without requiring any additional libraries or modules. Python offers several built-in data types, including:

- **Numeric Data Types:** Numeric data types in Python are used to represent numerical values. Python provides three primary numeric data types:
  - **Integer (int):** Integers are whole numbers without any decimal points. They can be positive or negative.
  - **Floating-Point (float):** Floating-point numbers represent decimal values. They can be positive or negative and may contain a decimal point.
  - **Complex (complex):** Complex numbers are used to represent numbers with a real and imaginary part. They are written in the form of a + bj, where a is the real part and b is the imaginary part.
- **String Data Type(str):** Represents a sequence of characters enclosed in single quotes (‘ ‘) or double quotes (” “), such as “Hello, World!”, ‘Python’.
- **Boolean Data Type(bool):** Represents either True or False, used for logical operations and conditions.
- **Collection Data Types:**
  - **list**: Represents an ordered and mutable collection of items, enclosed in square brackets ([]).
  - **tuple**: Represents an ordered and immutable collection of items, enclosed in parentheses ().
  - **dict**: Represents a collection of key-value pairs enclosed in curly braces ({}) with unique keys.
  - **set**: Represents an unordered and mutable collection of unique elements, enclosed in curly braces ({}) or using the set() function.

I. Numeric Data Types

Numeric data types in Python are used to represent numerical values. Python provides three primary numeric data types – integer (int), floating-Point (float) and complex (complex). These numeric data types allow for performing various arithmetic operations, such as addition, subtraction, multiplication, and division. They provide the necessary flexibility to work with numerical data in Python programs.

- **Int** – It stores the integers values that can be positive or negative and do not contain any decimal point. Example: num1=10, num2 = 15
- **Float** – These are floating-point real numbers that stores the decimal values. It consists of integer and fraction parts. Example: fnum = 25.4, fnum1=67.8
- **Complex** – These are complex numbers specified as a real part and an imaginary part. They are stored in the form of a + bj where a is the real part and j represents the imaginary part. Example: num3= 2 + 3j, numcom = 5 – 7j

Integers (int)

Python's integer data type (int) represents whole numbers without any decimal points. It stores positive and negative whole numbers. Integers are immutable, meaning their value cannot be changed once assigned.

Example 1

```
# Assigning integer values to variables
x = 5
y = -10
# Performing arithmetic operations
sum_result = x + y
difference_result = x - y
multiplication_result = x * y
division_result = x / y
# Printing the results
print("Sum:", sum_result)
print("Difference:", difference_result)
print("Multiplication:", multiplication_result)
print("Division:", division_result)
```

Output:

```
Sum: -5
Difference: 15
Multiplication: -50
Division: -0.5
```

Example 2

```
# Using integer values in comparisons
a = 10
b = 20
# Comparing the values
greater_than = a > b
less_than_or_equal = a <= b
equal_to = a == b
not_equal_to = a != b
# Printing the results
print("Greater than:", greater_than)
print("Less than or equal to:", less_than_or_equal)
print("Equal to:", equal_to)
print("Not equal to:", not_equal_to)
```

Output:

```
Greater than: False
Less than or equal to: True
Equal to: False
Not equal to: True
```

Floating-Point Numbers (float)

The float data type in Python is used to represent floating-point numbers, which are numbers with decimal points. Floats are used when more precision is needed than what integers can provide. Floats are immutable like integers and follow the IEEE 754 standard for representing real numbers.

Example 1

```
# Assigning float values to variables
x = 3.14
y = 2.5
# Performing arithmetic operations
sum_result = x + y
difference_result = x - y
multiplication_result = x * y
division_result = x / y
# Printing the results
print("Sum:", sum_result)
print("Difference:", difference_result)
print("Multiplication:", multiplication_result)
print("Division:", division_result)
```

Output:

```
Sum: 5.64
Difference: 0.64
Multiplication: 7.85
Division: 1.256
```

Example 2

```
# Using float values in comparisons
a = 1.2
b = 2.7
# Comparing the values
greater_than = a > b
less_than_or_equal = a <= b
equal_to = a == b
not_equal_to = a != b
# Printing the results
print("Greater than:", greater_than)
print("Less than or equal to:", less_than_or_equal)
print("Equal to:", equal_to)
print("Not equal to:", not_equal_to)
```

Output:

```
Greater than: False
Less than or equal to: True
Equal to: False
Not equal to: True
```

Complex Numbers (complex)

The complex data type in Python is used to represent numbers with both real and imaginary parts. It is written in the form of a + bj, where a represents the real part and b represents the imaginary part. Complex numbers are useful in mathematical calculations and scientific computations that involve imaginary quantities.

Example 1

```
# Assigning complex values to variables
x = 2 + 3j
y = -1 + 2j
# Performing arithmetic operations
sum_result = x + y
difference_result = x - y
multiplication_result = x * y
```

```
division_result = x / y
# Printing the results
print("Sum:", sum_result)
print("Difference:", difference_result)
print("Multiplication:", multiplication_result)
print("Division:", division_result)
```

Output:

```
Sum: (1+5j)
Difference: (3+1j)
Multiplication: (-8+1j)
Division: (0.8-1.4j)
```

Example 2

```
# Using complex values in comparisons
a = 1 + 2j
b = 3 + 4j
# Comparing the values
equal_to = a == b
not_equal_to = a != b
# Printing the results
print("Equal to:", equal_to)
print("Not equal to:", not_equal_to)
```

Output:

```
Equal to: False
Not equal to: True
```

II. Textual Data Types – Strings

Textual data types in Python are used to represent and manipulate sequences of characters, such as words, sentences, or even larger blocks of text. The primary textual data type in Python is the string (str). Strings are enclosed in quotes (‘ ‘, ” “, or “”” “””) and can be manipulated using various string methods. They are immutable, meaning their values cannot be changed once assigned. String data types are commonly used for tasks like text processing, input/output operations, and data manipulation.

Accessing String Values

Each character of a string can be expressed with the help of a technique called indexing. In indexing, each character has an index value represented by either a positive or negative integer starting from 0.

Syntax:- stringname[index]

Example

```
Str1="Python
#accessing second character
Str1[1]
#accessing first four characters
Str1[0:4]  # it will print the value from index 0 to 3
str1="Python"
>>> str1[1]
'y'
>>> str1[0:4]
'Pyth'
```

Positive indexing		1	2	3	4	5
String		y	t	h	o	n
Negative indexing		-5	-4	-3	-2	-1

III. Boolean Data Type

The boolean data type in Python represents logical values of True and False. It is used to evaluate conditions and make logical decisions in a program. Booleans are the result of comparisons and logical operations.

Example 1

```
# Logical operations with booleans
a = True
b = False
result_and = a and b
result_or = a or b
result_not = not a
print(result_and)
print(result_or)
print(result_not)
```

Output:

```
False
True
False
```

Example 2

```
#Comparisons using booleans
x = 5
y = 10
greater_than = x > y
```

```
less_than_or_equal = x <= y
equal_to = x == y
not_equal_to = x != y
print(greater_than)
print(less_than_or_equal)
print(equal_to)
print(not_equal_to)
```

Output:

```
False
True
False
True
```

IV. Collection Data Types

Collection data types in Python are used to store and organize multiple values into a single entity. Python provides several built-in collection data types, including lists, tuples, dictionaries, and sets.

Lists (list)

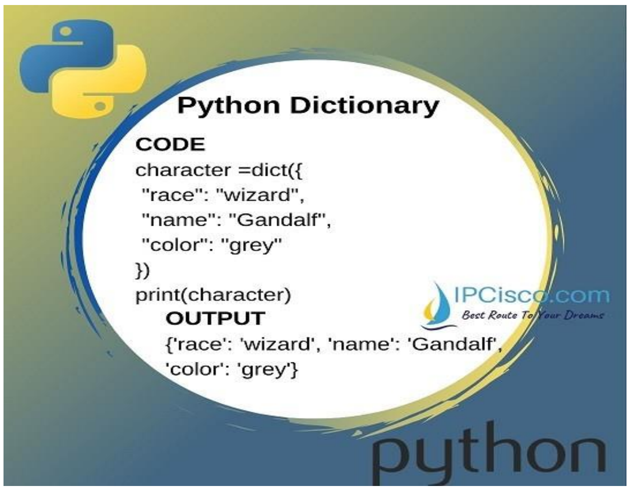
The list data type in Python is used to store multiple items in a single variable. Lists are ordered, mutable, and can contain elements of different data types. They are created by enclosing comma-separated values within square brackets [ ].

Example 1

```
List1=[123,567,89] #list of numbers
List2=['hello','how','are'] #list of strings
List3= ['They',1223,'hello'] #list of mixed data type.
```

Dictionary

Dictionary is a special data type that is a mapping between a set of keys and a set of values. It represents a key-value pair enclosed in curly brackets and each element is separated by a comma. Key-value pairs are separated by the colon. The elements of the dictionaries are unordered and the key is unique for each value in the dictionary. The elements of the dictionary are mutable that is its elements can be changed once it is created.


Python Dictionary
IPCisco.com
Best Route To Your Dreams
python

CODE

```
character =dict({
 "race": "wizard",
 "name": "Gandalf",
 "color": "grey"
})
print(character)
```

OUTPUT

```
{'race': 'wizard', 'name': 'Gandalf',
'color': 'grey'}
```

```
Syntax:- Dictionaryname={key:value}
```

Example

```
Dict1= {"comp": "computer", "sci" : "science"}

Dict2={"123":"computer",456 : "maths"}
```

Tuples (tuple)

The tuple data type in Python is similar to a list but with one crucial difference: tuples are immutable. Once created, the elements in a tuple cannot be modified. Tuples are created by enclosing comma-separated values within parentheses ( ).

Example: Tuple creation and accessing elements

```
# Creating a tuple

my_tuple = (1, 2, 3, 4, 5)

# Accessing elements by index

first_element = my_tuple[0]

last_element = my_tuple[-1]

# Slicing a tuple

subset = my_tuple[1:4]
```

```
# Printing the results

print(my_tuple)

print(first_element)

print(last_element)

print(subset)
```

Output:

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

1

5

(2, 3, 4)
```

II. Custom Data Types in Classes

In Python, classes provide a way to define custom data types. Classes encapsulate data and methods, allowing for the creation of objects with specific attributes and behaviors.Classes serve as blueprints for creating objects. They define the structure and behavior of objects, including their attributes (data) and methods (functions). Objects are instances of classes, and each object has its own unique state and behavior.

Example: Creating a class and instantiating objects

```
# Defining a class

class Point:

    def __init__(self, x, y):

        self.x = x

        self.y = y

# Creating objects

point1 = Point(2, 3)

point2 = Point(5, 7)
```

```
# Accessing object attributes

print(point1.x, point1.y)

print(point2.x, point2.y)
```

Output:

```
2 3

5 7
```

III. Variables

Create a Variable

In Python, variables are names that can be assigned a value and then used to refer to that value throughout your code. Variables are fundamental to programming for two reasons:

- Variables keep values accessible: For example, you can assign the result of some time-consuming operation to a variable so that your program doesn’t have to perform the operation each time you need to use the result.
- Variables give values context: The number 28 could mean lots of different things, such as the number of students in a class, the number of times a user has accessed a website, and so on. Giving the value 28 a name like num_students makes the meaning of the value clear.

In this section, you’ll learn how to use variables in your code, as well as some of the conventions Python programmers follow when choosing names for variables.

The Assignment Operator

An operator is a symbol, such as +, that performs an operation on one or more values. For example, the + operator takes two numbers, one to the left of the operator and one to the right, and adds them together.

Values are assigned to variable names using a special symbol called the **assignment operator** (=) . The = operator takes the value to the right of the operator and assigns it to the name on the left.

Let’s modify the hello_world.py file from the previous section to assign some text in a variable before printing it to the screen:

```
>>> greeting = "Hello, World"
>>> print(greeting)
Hello, world
```

On the first line, you create a variable named greeting and assign it the value "Hello, World" using the = operator.

print(greeting) displays the output Hello, World because Python looks for the name greeting, finds that it’s been assigned the value "Hello, World", and replaces the variable name with its value before calling the function.

If you hadn’t executed greeting = "Hello, World" before executing print(greeting), then you would have seen a NameError like you did when you tried to execute print(Hello, World) in the previous section.

Although = looks like the equals sign from mathematics, it has a different meaning in Python. This distinction is important and can be a source of frustration for beginner programmers. Just remember, whenever you see the = operator, whatever is to the right of it is being assigned to a variable on the left.

Variable names are **case sensitive**, so a variable named greeting is not the same as a variable named Greeting. For instance, the following code produces a NameError:

```
>>> greeting = "Hello, World"
>>> print(Greeting)
Traceback (most recent call last):
File "", line 1, in <module>
NameError: name 'Greeting' is not defined
```

If you have trouble with an example in this book, double-check that every character in your code—including spaces—matches the example exactly. Computers have no common sense, so being almost correct isn’t good enough!

Rules for Valid Variable Names

Variable names can be as long or as short as you like, but there are a few rules that you must follow. Variable names may contain uppercase and lowercase letters (A–Z, a–z), digits (0–9), and underscores (_), but they cannot begin with a digit.

For example, each of the following is a valid Python variable name:

- string1
- _a1p4a

list_of_names The following aren’t valid variable names because they start with a digit:

- 9lives
- 99_balloons
- 2beOrNot2Be

In addition to English letters and digits, Python variable names may contain many different valid Unicode characters.

Unicode is a standard for digitally representing characters used in most of the world's writing systems. That means variable names can contain letters from non-English alphabets, such as decorated letters.

like é and ü, and even Chinese, Japanese, and Arabic symbols.

However, not every system can display decorated characters, so it's a good idea to avoid them if you're going to share your code with people in different regions.

Just because a variable name is valid doesn't necessarily mean that it's a good name.

Choosing a good name for a variable can be surprisingly difficult. Fortunately, there are some guidelines that you can follow to help you choose better names.

Descriptive Names Are Better Than Short Names

Descriptive variable names are essential, especially for complex programs. Writing descriptive names often requires using multiple words. Don't be afraid to use long variable names.

In the following example, the value 3600 is assigned to the variable s:

```
s = 3600
```

The name s is totally ambiguous. Using a full word makes it a lot easier to understand what the code means:

```
seconds = 3600
```

seconds is a better name than s because it provides more context. But it still doesn't convey the full meaning of the code. Is 3600 the number of seconds it takes for a process to finish, or is it the length of a movie? There's no way to tell.

The following name leaves no doubt about what the code means:

```
seconds_per_hour = 3600
```

When you read the above code, there's no question that 3600 is the number of seconds in an hour. seconds_per_hour takes longer to type than both the single letter s and the word seconds, but the payoff in clarity is massive.

Although naming variables descriptively means using longer variable names, you should avoid using excessively long names. A good rule of thumb is to limit variable names to three or four words maximum.

Python Variable Naming Conventions

In many programming languages, it’s common to write variable names in **mixedCase**. In this system, you capitalize the first letter of every word except the first and leave all other letters in lowercase. For example, numStudents and listOfNames are written in mixedCase.

In Python, however, it’s more common to write variable names in **lower_case_with_underscores**. In this system, you leave every letter in lowercase and separate each word with an underscore. For instance, both num_students and *list_of_names* are written using the lower_case_with_underscores system.

There’s no rule mandating that you write your variable names in lower_case_with_underscores. The practice is codified, though, in a document called PEP8 [*https://pep8.org/*]which is widely regarded as the official style guide for writing Python.

PEP stands for Python Enhancement Proposal. A PEP is a design document used by the Python community to propose new features to the language.

Following the standards outlined in PEP 8 ensures that your Python code is readable by most Python programmers. This makes sharing code and collaborating with other people easier for everyone involved.

1.4 Numeric Values

Although math and computer programming aren’t as correlated as some people might believe, numbers are an integral part of any programming language and Python is no exception.

Integers and Floating-Point Numbers

Python has three built-in number data types: integers, floating-point numbers, and complex numbers. In this section, you’ll learn about integers and floating-point numbers, which are the two most commonly used number types.

Integers

An integer is a whole number with no decimal places. For example, 1 is an integer, but 1.0 isn't. The name for the integer data type is int, which you can see with the type() function:

```
>>> type(1)
<class 'int'>
```

You can create an integer by simply typing the number explicitly or using the *int()* function. For example, the following converts the string *"25*" to the integer *25*:

```
>>> int("25")
25
```

An **integer literal** is an integer value that is written explicitly in your code, just like a string literal is a string that is written explicitly in your code. For example, 1 is an integer literal, but *int("1")* isn't.

Integer literals can be written in two different ways:

```
>>> 1000000
1000000
>>> 1000000
1000000
```

The first example is straightforward. Just type a 1 followed by six zeros. The downside to this notation is that large numbers can be difficult to read.

When you write large numbers by hand, you probably group digits into groups of three, separated by a comma. 1,000,000 is a lot easier to read than 1000000.

In Python, you can't use commas to group digits in integer literals, but you can use an underscore (_). The value 1_000_000 expresses one million in a more readable manner.

There is no limit to how large an integer can be, which might be surprising considering computers have finite memory. Try typing the largest number you can think of into IDLE's interactive window. Python can handle it with no problem!

Floating-Point Numbers

A coating-point number, or coat for short, is a number with a decimal place. 1.0 is a floating-point number, as is -2.75. The name of a floating-point data type is float:

```
>>> type(1.0)
<class 'float'>
```

Floats can be created by typing a number directly into your code, or by using the float() function. Like int(), float() can be used to convert a string containing a number to a floating-point number:

```
>>> float("1.25")
1.25
```

A **floating-point** literal is a floating-point value that is written explicitly in your code. 1.25 is a floating-point literal, while float("1.25") is not.

Floating-point literals can be created in three different ways. Each of the following creates a floating-point literal with a value of one million:

```
>>> 1000000.0
1000000.0
>>> 1000000.0
1000000.0
>>> 1e6
1000000.0
```

The first two ways are similar to the two methods for creating integer literals that you saw earlier. The second method, which uses underscores to separate digits into groups of three, is useful for creating float literals with lots of digits.

For really large numbers, you can use **E-notation**. The third method in the previous example uses E-notation to create a float literal.

To write a float literal in E-notation, type a number followed by the letter e and then another number. Python takes the number to the left of the e and multiplies by *10* raised to the power of the number after the e. So *1e6* is equivalent to 1×10^6.

E-notation is short for **exponential notation**, and is the more common name for how many calculators and programming languages display large numbers.

Python also uses E-notation to display large floating point numbers:

```
>>> 200000000000000000.0
2e+17
```

The float *200000000000000000.0* gets displayed as *2e+17*. The + sign indicates that the exponent *17* is a positive number. You can also use negative numbers as the exponent:

```
>>> 1e-4
0.0001
```

The literal *1e-4* is interpreted as *10* raised to the power *-4*, which is *1/10000* or, equivalently, *0.0001*.

Unlike integers, floats do have a maximum size. The maximum floating-point number depends on your system, but something like *2e400* ought to be well beyond most machines' capabilities. *2e400 is* 2×10^{400}, which is far more than the total number of atoms in the universe![*https://en.wikipedia.org/wiki/Observable_universe#Matter_content*]

When you reach the maximum floating-point number, Python returns a special float value *inf*:

```
>>> 2e400
inf
```

inf stands for infinity, and it just means that the number you’ve tried to create is beyond the maximum floating-point value allowed on your computer. The type of inf is still float:

```
>>> n = 2e400
>>> n
inf
>>> type(n)
<class 'float'>
```

There is also *-inf* which stands for negative infinity, and represents a negative floating-point number that is beyond the minimum floating point number allowed on your computer:

```
>>> -2e400
-inf
```

You probably won’t come across *inf* and *-inf* often as a programmer, unless you regularly work with extremely large numbers.

Arithmetic Operators and Expressions

In this section, you’ll learn how to do basic arithmetic with numbers in Python, such as addition, subtraction, multiplication, and division. Along the way, you’ll learn some conventions for writing mathematical expressions in code.

i.**Addition**

Addition is performed with the + operator:

```
>>> 1 + 2
3
```

The two numbers on either side of the + operator are called **operands**. In the previous example, both operands are integers, but operands do not need to be the same type. You can add an int to a float with no problem:

```
>>> 1.0 + 2
3.0
```

Notice that the result of 1.0 + 2 is 3.0, which is a float. Any time a float is added to a number, the result is another float. Adding two integers together always results in an int.

PEP 8 recommends[https://pep8.org/#other-recommendations] separating both operands from an operator with a space. Python can evaluate 1+1 just fine, but 1 + 1 is the preferred format because it’s generally considered easier to read. This rule of thumb applies to all of the operators in this section.

ii.Subtraction

To subtract two numbers, just put a - in between them:

```
>>> 1 - 1
0
>>> 5.0 - 3
2.0
```

Just like adding two integers, subtracting two integers always results in an int. Whenever one of the operands is a float, the result is also a float. The - operator is also used to denote negative numbers:

```
>>> -3
-3
```

You can subtract a negative number from another number, but as you can see below, this can sometimes look confusing:

```
>>> 1 -3
4
>>> 1 -3
4
>>> 1-3
4
 >>>1-3
4
```

Of the four examples above, the first is the most PEP 8 compliant. That said, you can surround -3 with parentheses to make it even clearer that the second - is modifying 3:

```
>>> 1 - (-3)
4
```

Using parentheses is a good idea because it makes the code more explicit. Computers execute code, but humans read code. Anything you can do to make your code easier to read and understand is a good thing.

iii.Multiplication

To multiply two numbers, use the * operator:

```
>>> 3 * 3
9
>>> 2 * 8.0
16.0
```

The type of number you get from multiplication follows the same rules as addition and subtraction. Multiplying two integers results in an int, and multiplying a number with a float results in a float.

Division

The / operator is used to divide two numbers:

```
>>> 9 / 3
3.0
>>> 5.0 / 2
2.5
```

Unlike addition, subtraction, and multiplication, division with the / operator always returns a float. If you want to make sure that you get an integer after dividing two numbers, you can use int() to convert the result:

```
>>> int(9 / 3)
3
```

Keep in mind that int() discards any fractional part of the number:

```
>>> int(5.0 / 2)
2
```

5.0 / 2 returns the float 2.5, and int(2.5) returns the integer 2 with the .5 part removed.

iv.Integer Division

If writing *int(5.0 / 2)* seems a little long-winded to you, Python provides a second division operator, //, called the integer division operator:

```
>>> 9 // 3
3
>>> 5.0 // 2
2.0
>>> -3 // 2
-2
```

The // operator first divides the number on the left by the number on the right and then rounds down to an integer. This might not give the value you expect when one of the numbers is negative.

For example, -3 // 2 returns -2. First, -3 is divided by 2 to get -1.5. Then -1.5 is rounded down to -2. On the other hand, 3 // 2 returns 1 .

Another thing the above example illustrates is that // returns a floating-point number if one of the operands is a float. This is why 9 // 3 returns the integer 3 and 5.0 // 2 returns the float 2.0. Let's see what happens when you try to divide a number by 0:

```
>>> 1 / 0
Traceback (most recent call last):
```

```
File "<stdin>", line 1 , in<module>
ZeroDivisionError: division by zero
```

Python gives you a ZeroDivisionError, letting you know that you just tried to break a fundamental rule of the universe.

Exponents You can raise a number to a power using the ** operator:

```
>>> 2 ** 2
4
>>> 2 ** 3
8
>>> 2 ** 4
16
```

Exponents don’t have to be integers. They can also be floats:

```
>>> 3 ** 1.5
5.196152422706632
>>> 9 ** 0.5
3.0
```

Raising a number to the power of 0.5 is the same as taking the square root, but notice that even though the square root of 9 is an integer, Python returns the float 3.0.

For positive operands, the ** operator returns an integer if both operands are integers, and a float if any one of the operands is a floating-point number.

You can also raise numbers to negative powers:

```
>>> 2 ** -1
0.5
>>> 2 ** -2
0.25
```

Raising a number to a negative power is the same as dividing 1 by the number raised to the positive power. So, 2 ** -1 is the same as 1 / (2 ** 1), which is the same as 1 / 2, or 0.5. Similarly 2 ** -2 is the same as 1 / (2 ** 2), which is the same as 1 / 4, or 0.25.

v.The Modulus Operator

The % operator, or the modulus, returns the remainder of dividing the left operand by the right operand:

```
>>> 5 % 3
2
>>> 20 % 7
6
>>> 16 % 8
```

```
0
```

3 divides 5 once with a remainder of 2, so 5 % 3 is 2. Similarly, 7 divides 20 twice with a remainder of 6.

In the last example, 16 is divisible by 8, so 16 % 8 is 0. Any time the number to the left of % is divisible by the number to the right, the result is 0.

One of the most common uses of % is to determine whether or not one number is divisible by another. For example, a number n is even if and only if n % 2 is 0.

What do you think 1 % 0 returns? Let's try it out:

```
>>> 1 % 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This makes sense because 1 % 0 is the remainder of dividing 1 by 0. But you can't divide 1 by 0, so Python raises a ZeroDivisionError.

When you work in IDLE's interactive window, errors like **ZeroDivisionError** don't cause much of a problem. The error is displayed and a new prompt pops up allowing you to continue writing code. However, whenever Python encounters an error while running a script, execution stops. The program is said to have **crashed**.

Things get a little tricky when you use the % operator with negative numbers:

```
>>> 5 % -3
-1
>>> -5 % 3
1
>>> -5 % -3
-2
```

These potentially shocking results are really quite well defined. To calculate the remainder r of dividing a number x by a number y, Python uses the equation r = x - (y * (x // y)).

For example, to find 5 % -3, first find (5 // -3). Since 5 / -3 is about -1.67, 5 // -3 is -2. Now multiply that by -3 to get 6. Finally, subtract 6 from 5 to get -1.

Arithmetic Expressions

You can combine operators to form complex expressions. An expression is a combination of numbers, operators, and parentheses that Python can compute, or evaluate, to return a value. Here are some examples of arithmetic expressions:

```
>>> 2*3 - 1
```

```
5
>>> 4/2 + 2**3
10.0
>>> -1 + (-3*2 + 4)
-3
```

The rules for evaluating expressions work are the same as in everyday arithmetic. In school, you probably learned these rules under the name “order of operations.”

The *, /, //, and % operators all have equal precedence, or priority, in an expression, and each of these has a higher precedence than the + and - operators. This is why 2*3 - 1 returns 5 and not 4. 2*3 is evaluated first, because * has higher precedence than the - operator. You may notice that the expressions in the previous example do not follow the rule for putting a space on either side of all of the operators. PEP 8 says the following about whitespace in complex expressions:

Challenge: Perform Calculations on User Input

Write a script called exponent.py that receives two numbers from the user and displays the first number raised to the power of the second number.

A sample run of the program should look like this (with example input that has been provided by the user included below):

```
Enter a base: 1.2
Enter an exponent: 3
1.2 to the power of 3 = 1.7279999999999998
```

Make Python Lie to You

What do you think 0.1 + 0.2 is? The answer is 0.3, right? Let’s see what Python has to say about it. Try this out in the interactive window:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Well, that’s… almost right! What in the heck is going on here? Is this a bug in Python?

No, it isn’t a bug! It’s a **floating-point representation error**, and it has nothing to do with Python. It’s related to the way floating-point numbers are stored in a computer’s memory.

The number 0.1 can be represented as the fraction 1/10. Both the number 0.1 and it’s fraction 1/10 are decimal representations, or base 10 representations. Computers, however, store floating-point numbers in base 2 representation, more commonly called **binary representation.**

When represented in binary, something familiar yet possibly unexpected happens to the decimal number 0.1. The fraction 1/3 has no finite decimal representation. That is, 1/3 = 0.3333... with infinitely many 3's after the decimal point. The same thing happens to the fraction 1/10 in binary.

The binary representation of 1/10 is the following infinitely repeating fraction:

```
0.00011001100110011001100110011...
```

Computers have finite memory, so the number 0.1 must be stored as an approximation and not as its true value. The approximation that gets stored is slightly higher than the actual value, and looks like this:

```
0.1000000000000000055511151231257827021181583404541015625
```

You may have noticed, however, that when asked to print 0.1, Python prints 0.1 and not the approximated value above:

```
>>> 0.1
0.1
```

Python doesn't just chop off the digits in the binary representation for 0.1. What actually happens is a little more subtle.

Because the approximation of 0.1 in binary is just that—an approximation—it is entirely possible that more than one decimal number have the same binary approximation.

For example, the numbers 0.1 and 0.10000000000000001 both have the same binary approximation. Python prints out the shortest decimal number that shares the approximation.

This explains why, in the first example of this section, 0.1 + 0.2 does not equal 0.3. Python adds together the binary approximations for 0.1 and 0.2, which gives a number which is not the binary approximation for 0.3.

If all this is starting to make your head spin, don't worry! Unless you are writing programs for finance or scientific computing, you don't need to worry about the imprecision of floating-point arithmetic.

1.5 String Variables

Strings are sequences of characters. Your name can be considered a string. Or, say you live in Zambia, then your country name is "Zambia", which is a string.

In this tutorial you will see how strings are treated in Python, the different ways in which strings are represented in Python, and the ways in which you can use strings in your code.

How to create a string and assign it to a variable

To create a string, put the sequence of characters inside either single quotes, double quotes, or triple quotes and then assign it to a variable. You can look into how variables work in Python in the Python variables tutorial.

For example, you can assign a character ‘a’ to a variable single_quote_character. Note that the string is a single character and it is “enclosed” by single quotes.

```
>>> single_quote_character = 'a'
>>> print(single_quote_character)
a
>>> print(type(single_quote_character)) # check the type of the variable.
<class 'str'>
```

Similarly, you can assign a single character to a variable double_quote_character. Note that the string is a single character but it is “enclosed” by double quotes.

```
>>> double_quote_character = "b"
>>> print(double_quote_character)
b
>>> print(type(double_quote_character))
<class 'str'>
```

Also check out if you can assign a sequence of characters or multiple characters to a variable. You can assign both single quote sequences and double quote sequences.

```
>>> double_quote_multiple_characters = "aeiou"
>>> single_quote_multiple_characters = 'aeiou'
>>> print(type(double_quote_multiple_characters),
type(single_quote_multiple_characters))
<class 'str'> <class 'str'>
```

Interestingly if you check the equivalence of one to the other using the keyword is, it returns True.

```
>>> print(double_quote_multiple_characters is double_quote_multiple_characters)
True
```

Take a look at assignment of strings using triple quotes and check if they belong to the class str as well.

```
>>> triple_quote_example = """this is a sentence written in triple quotes"""
>>> print(type(triple_quote_example))
<class 'str'>
```

In the examples above, the function type is used to show the underlying class that the object will belong to. Please note that all the variables that have been initiated with single, double, or triple quotes are taken as string. You can use single and double quotes for a single line of characters. Multiple lines are generally put in triple quotes.

Self-Check Sheet -1: Work with Python basic

Q1. Write python code to print “Hello World”

Q2. Take user input, performing a simple calculation, and printing the result

Q3. Write Python code to check if a number is even or odd.

Q4. Write Python code to print numbers from 1 to 5 using a for loop.

Q5. Write a python code that will define a function to calculate the factorial of a number.

Answer Sheet - 1: Work with Python basic

Q1. Write python code to print "Hello World"

Answer:

```
print("Hello, World!")
```

Q2. Take user input, performing a simple calculation, and printing the result

Answer:

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
year = 2022 - age
print(f"Hello, {name}! You were born in {year}.")
```

Q3. Write Python code to check if a number is even or odd.

Answer:

```
number = int(input("Enter a number: "))
if number % 2 == 0:
    print("The number is even.")
else:
    print("The number is odd.")
```

Q4. Write Python code to print numbers from 1 to 5 using a for loop.

Answer:

```
for i in range(1, 6):
    print(i)
```

Q5. Write a python code that will definine a function to calculate the factorial of a number.

Answer:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

number = int(input("Enter a number: "))
print(f"The factorial of {number} is {factorial(number)}.")
```

Task Sheet-1.1: How to Define a class in Python?

In Python, you can define a class using the class keyword followed by the class name. Here's the syntax for defining a class in Python:

```
class ClassName:
    # Class attributes and methods go here
```

Now, let's break down the components of a class definition:

Class Keyword: It starts with the class keyword followed by the name of the class. By convention, class names are written in CamelCase.

Class Name: This is the name given to the class. It should be descriptive of the entity the class represents.

Class Body: The body of the class contains class attributes (variables) and methods (functions).

Here's a brief description of each:

- **Class Attributes:** These are variables that are shared by all instances (objects) of the class. They are defined inside the class but outside of any method.
- **Methods:** These are functions defined within a class. They define the behavior of the class and can perform operations on the class's attributes. There are different types of methods in a class:
  - **Initializer Method (**__init__): This method is called when an object of the class is instantiated. It is used to initialize the object's attributes.
  - Instance Methods: These methods take self as the first parameter and operate on individual instances of the class.
  - **Class Methods**: These methods are decorated with @classmethod and take cls as the first parameter. They operate on the class itself rather than individual instances.
  - **Static Methods:** These methods are decorated with @staticmethod and do not take self or cls parameters. They are independent of the class and can be called without creating an instance.

Here's a simple example of a Python class with an initializer method (__init__) and an instance method:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."

# Creating an instance of the Person class
person1 = Person("Alice", 30)

# Calling the greet method
print(person1.greet())
```

In this example, we define a Person class with an initializer method (__init__) that sets the name and age attributes of the object. We also define an instance method greet that returns a greeting message using the object's name and age attributes. Finally, we create an instance of the Person class (person1) and call the greet method on it.

Task Sheet-1.2: How to take input from a user in python

In Python, you can take input from a user using the input() function. Here's how you can use it:

```
user_input = input("Enter something: ")
```

The input() function takes an optional string argument, which is displayed to the user as a prompt. It then waits for the user to input something from the keyboard and press Enter. Whatever the user types is returned as a string and can be stored in a variable.

Here's a breakdown of the code:

- input(): This function reads a line from the input, converts it into a string (stripping a trailing newline), and returns that. It prompts the user with the optional prompt string.
- "Enter something: ": This is the optional prompt string that is displayed to the user. It's a good practice to provide a prompt so that the user knows what type of input is expected.
- user_input: This variable stores the input provided by the user. It will contain whatever the user types after the prompt and presses Enter.

Here's an example:

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

When you run this code, it will prompt the user to enter their name. After the user enters their name and presses Enter, the program will greet them with "Hello, " followed by their name.

Remember that the input() function always returns a string. If you want to convert the input to a different data type (e.g., integer, float), you can do so explicitly using type conversion functions like int(), float(), etc. For example:

```
age = int(input("Enter your age: "))
```

This will convert the user's input to an integer before storing it in the age variable.

Task Sheet-1.3: How to perform arithmetic operations in python?

In Python, you can perform arithmetic operations such as addition, subtraction, multiplication, division, and more using arithmetic operators. Here's how you can perform basic arithmetic operations:

- **Addition (+):** Used to add two numbers.

```
result = 5 + 3
print(result)  # Output: 8
```

- **Subtraction (-):** Used to subtract one number from another.

```
result = 10 - 4
print(result)  # Output: 6
```

- **Multiplication (*):** Used to multiply two numbers.

```
result = 6 * 4
print(result)  # Output: 24
```

- **Division (/):** Used to divide one number by another. In Python 3, division always returns a floating-point number.

```
result = 20 / 5
print(result)  # Output: 4.0
```

- **Modulus (%):** Returns the remainder of the division of the first number by the second.

```
result = 20 % 6
print(result)  # Output: 2
```

- **Exponentiation (**):** Raises the first number to the power of the second.

```
result = 2 ** 3
print(result)  # Output: 8
```

These are the basic arithmetic operations you can perform in Python. You can combine these operators and use parentheses to perform more complex arithmetic expressions. For example:

```
result = (10 + 5) * 2 - 3 / 2
print(result)  # Output: 27.5
```

In this expression, first, the addition (10 + 5) is performed, then the result is multiplied by 2, and finally, the division (3 / 2) is performed and subtracted from the previous result.

Job Sheet-1: Develop a bank account management system using python

Problem Statement: Develop a bank account management system using python. The system should contain at least the following features.

- **Deposit**: Users can deposit funds into their account.
- **Withdraw**: Users can withdraw funds from their account, provided they have sufficient balance.
- **Check Balance**: Users can check their current account balance.
- **Exit**: Users can choose to exit the system.

Solutions:

Lets develop the bank management system step by step:

Class Definition: The code defines a class BankAccount which represents a bank account. It has methods to deposit, withdraw, and check balance.

```
class BankAccount:
    def __init__(self, name, balance=0):
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount}. New balance is ${self.balance}.")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance is ${self.balance}.")
        else:
            print("Insufficient funds or invalid withdrawal amount.")

    def get_balance(self):
        return self.balance
```

Creating an Account: The main function prompts the user to input their name to create a new bank account.

```
account_name = input("Enter account holder's name: ")
account = BankAccount(account_name)
```

Menu Loop: The program then enters a loop where the user can perform different operations like deposit, withdraw, check balance, or exit.

```
while True:
    print("\nSelect an option:")
    print("1. Deposit")
    print("2. Withdraw")
    print("3. Check Balance")
    print("4. Exit")
    choice = input("Enter your choice: ")
```

Performing Operations: Depending on the user's choice, the corresponding method of the BankAccount object is called.

```
if choice == '1':
    amount = float(input("Enter amount to deposit: "))
    account.deposit(amount)
elif choice == '2':
    amount = float(input("Enter amount to withdraw: "))
    account.withdraw(amount)
elif choice == '3':
    print(f"Your current balance is ${account.get_balance()}.")
elif choice == '4':
    print("Thank you for using our bank system. Goodbye!")
    break
else:
    print("Invalid choice. Please try again.")
```

Repeating the Process: After each operation, the menu is displayed again until the user chooses to exit.

Example Interaction:

```
Enter account holder's name: John

Select an option:
1. Deposit
2. Withdraw
3. Check Balance
4. Exit
Enter your choice: 1
Enter amount to deposit: 100
Deposited $100. New balance is $100.

Select an option:
1. Deposit
2. Withdraw
3. Check Balance
4. Exit
Enter your choice: 3
Your current balance is $100.

Select an option:
1. Deposit
2. Withdraw
3. Check Balance
4. Exit
Enter your choice: 2
Enter amount to withdraw: 50
Withdrew $50. New balance is $50.

Select an option:
1. Deposit
2. Withdraw
3. Check Balance
4. Exit
Enter your choice: 4
Thank you for using our bank system. Goodbye!
```

The Whole Process in a Code:

```
class BankAccount:
    def __init__(self, name, balance=0):
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount}. New balance is ${self.balance}.")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance is ${self.balance}.")
        else:
            print("Insufficient funds or invalid withdrawal amount.")

    def get_balance(self):
        return self.balance


def main():
    # Create a new bank account
    account_name = input("Enter account holder's name: ")
    account = BankAccount(account_name)

    while True:
        print("\nSelect an option:")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Check Balance")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            amount = float(input("Enter amount to deposit: "))
            account.deposit(amount)
        elif choice == '2':
            amount = float(input("Enter amount to withdraw: "))
            account.withdraw(amount)
        elif choice == '3':
            print(f"Your current balance is ${account.get_balance()}.")
        elif choice == '4':
            print("Thank you for using our bank system. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Learning Outcome 2: Apply Input and Output Methods in Python

Assessment Criteria	1. Printing with parameters is exercised 2. Getting inputs from users is exercised 3. String formatting is applied 4. Simple and complex decision making is applied using logical statements
Condition and Resource	1.Actual workplace or training environment 2.CBLM 3.Handouts 4.Laptop 5.Multimedia Projector 6.Paper, Pen, Pencil and Eraser 7.Internet Facilities 8.Whiteboard and Marker 9.Imaging Device (Digital camera, scanner etc.)
Content	▪ Printing with parameters ▪ Getting inputs from users ▪ String formatting ▪ Simple and complex decision making using logical statements o The “if” statement o Logical operator o Complex expressions
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning

Learning Experience 2: Apply Input and Output Methods in Python

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Apply Input and Output Methods in Python'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Work with Python basic"	2. Read Information sheet 2: Apply Input and Output Methods in Python 3. Answer Self-check 2: Apply Input and Output Methods in Python 4. Check your answer with Answer key 3: Apply Input and Output Methods in Python
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet - 2.1: Printing with Parameters Task Sheet - 2.2: String Formatting Task Sheet - 2.3: Complex Expressions Job Sheet 2: Student Grading System

Information Sheet 2: Apply Input and Output Methods in Python

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

2.1 Exercise printing with parameters
2.2 Getting inputs from users
2.3 String formatting
2.4 Simple and complex decision-making approach using logical statements

2.1 Printing with parameters

Printing parameters, or string formatting parameters, are essential in Python for several reasons. Firstly, they significantly enhance the readability of code by allowing developers to embed variable values within strings, making it easier to understand the context and purpose of the output. Moreover, parameters enable the creation of dynamic output, enabling programs to adapt to changing data and requirements. They also play a crucial role in localization and internationalization efforts by facilitating the customization of output formats for different regions or languages. Additionally, parameters provide control over data representation, allowing developers to format numbers, dates, and other types of data according to specific requirements. Furthermore, they contribute to security by helping prevent vulnerabilities such as injection attacks, where untrusted data could inadvertently be executed as code. Overall, printing parameters are fundamental to string manipulation and output formatting in Python, crucial for producing readable, flexible, and secure code.

Let’s jump in by looking at a few real-life examples of printing in Python. By the end of this section, you’ll know every possible way of calling print(). Or, in programmer lingo, you’d say you’ll be familiar with the function signature.

Calling print()

The simplest example of using Python print() requires just a few keystrokes:

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

You don’t pass any arguments, but you still need to put empty parentheses at the end, which tell Python to actually execute the function rather than just refer to it by name.

This will produce an invisible newline character, which in turn will cause a blank line to appear on your screen. You can call print() multiple times like this to add vertical space. It’s just as if you were hitting Enter on your keyboard in a word processor.

As you just saw, calling print() without arguments results in a blank line, which is a line composed solely of the newline character. Don’t confuse this with an empty line, which doesn’t contain any characters at all, not even the newline!

You can use Python’s string literals to visualize these two:

```
'\n'  # Blank line
''    # Empty line
```

The first one is one character long, whereas the second one has no content.

To remove the newline character from a string in Python, use its .rstrip() method, like this:

```
>>> 'A line of text.\n'.rstrip()
'A line of text.'
```

This strips any trailing whitespace from the right edge of the string of characters.

In a more common scenario, you’d want to communicate some message to the end user. There are a few ways to achieve this.

First, you may pass a string literal directly to print():

```
>>> print('Please wait while the program is loading...')
```

This will print the message verbatim onto the screen.

Secondly, you could extract that message into its own variable with a meaningful name to enhance readability and promote code reuse:

```
>>> message = 'Please wait while the program is loading...'
>>> print(message)
```

Lastly, you could pass an expression, like string concatenation, to be evaluated before printing the result:

```
>>> import os
>>> print('Hello, ' + os.getlogin() + '! How are you?')
Hello, jdoe! How are you?
```

In fact, there are a dozen ways to format messages in Python. I highly encourage you to take a look at f-strings, introduced in Python 3.6, because they offer the most concise syntax of them all:

```
>>> import os
>>> print(f'Hello, {os.getlogin()}! How are you?')
```

Moreover, f-strings will prevent you from making a common mistake, which is forgetting to type cast concatenated operands. Python is a strongly typed language, which means it won’t allow you to do this:

```
>>> 'My age is ' + 42
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    'My age is ' + 42
TypeError: can only concatenate str (not "int") to str
```

That’s wrong because adding numbers to strings doesn’t make sense. You need to explicitly convert the number to string first, in order to join them together:

```
>>> 'My age is ' + str(42)
'My age is 42'
```

Unless you handle such errors yourself, the Python interpreter will let you know about a problem by showing a traceback.

str() is a global built-in function that converts an object into its string representation.

You can call it directly on any object, for example, a number:

```
>>> str(3.14)
'3.14'
```

Built-in data types have a predefined string representation out of the box, but later in this article, you’ll find out how to provide one for your custom classes.

As with any function, it doesn’t matter whether you pass a literal, a variable, or an expression. Unlike many other functions, however, print() will accept anything regardless of its type.

So far, you only looked at the string, but how about other data types? Let’s try literals of different built-in types and see what comes out:

```
>>> print(42)                            # <class 'int'>
42
>>> print(3.14)                          # <class 'float'>
3.14
>>> print(1 + 2j)                        # <class 'complex'>
(1+2j)
>>> print(True)                          # <class 'bool'>
True
>>> print([1, 2, 3])                     # <class 'list'>
[1, 2, 3]
>>> print((1, 2, 3))                     # <class 'tuple'>
(1, 2, 3)
>>> print({'red', 'green', 'blue'})      # <class 'set'>
{'red', 'green', 'blue'}
>>> print({'name': 'Alice', 'age': 42})  # <class 'dict'>
{'name': 'Alice', 'age': 42}
>>> print('hello')

hello                                    # <class 'str'>
```

Despite being used to indicate an absence of a value, it will show up as 'None' rather than an empty string:

```
>>> print(None)
None
```

How does print() know how to work with all these different types? Well, the short answer is that it doesn’t. It implicitly calls str() behind the scenes to type cast any object into a string. Afterward, it treats strings in a uniform way.

Later in this tutorial, you’ll learn how to use this mechanism for printing custom data types such as your classes.

Okay, you’re now able to call print() with a single argument or without any arguments. You know how to print fixed or formatted messages onto the screen. The next subsection will expand on message formatting a little bit.

Separating Multiple Arguments

You saw print() called without any arguments to produce a blank line and then called with a single argument to display either a fixed or a formatted message.

However, it turns out that this function can accept any number of positional arguments, including zero, one, or more arguments. That’s very handy in a common case of message formatting, where you’d want to join a few elements together.

Let’s have a look at this example:

```
>>> import os
>>> print('My name is', os.getlogin(), 'and I am', 42)
My name is jdoe and I am 42
```

print() concatenated all four arguments passed to it, and it inserted a single space between them so that you didn’t end up with a squashed message like 'My name is jdoe and I am42'.

Notice that it also took care of proper type casting by implicitly calling str() on each argument before joining them together. If you recall from the previous subsection, a naïve concatenation may easily result in an error due to incompatible types:

```
>>> print('My age is: ' + 42)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    print('My age is: ' + 42)
TypeError: can only concatenate str (not "int") to str
```

Apart from accepting a variable number of positional arguments, print() defines four named or keyword arguments, which are optional since they all have default values. You can view their brief documentation by calling help(print) from the interactive interpreter.

Let’s focus on sep just for now. It stands for separator and is assigned a single space (' ') by default. It determines the value to join elements with.

It has to be either a string or None, but the latter has the same effect as the default space:

```
>>> print('hello', 'world', sep=None)
hello world
>>> print('hello', 'world', sep=' ')
hello world
>>> print('hello', 'world')
hello world
```

If you wanted to suppress the separator completely, you’d have to pass an empty string ('') instead:

```
>>> print('hello', 'world', sep='')
helloworld
```

You may want print() to join its arguments as separate lines. In that case, simply pass the escaped newline character described earlier:

```
>>> print('hello', 'world', sep='\n')
hello
world
```

A more useful example of the sep parameter would be printing something like file paths:

```
>>> print('home', 'user', 'documents', sep='/')
home/user/documents
```

Remember that the separator comes between the elements, not around them, so you need to account for that in one way or another:

```
>>> print('/home', 'user', 'documents', sep='/')
/home/user/documents
>>> print('', 'home', 'user', 'documents', sep='/')
/home/user/documents
```

Specifically, you can insert a slash character (/) into the first positional argument, or use an empty string as the first argument to enforce the leading slash.

Be careful about joining elements of a list or tuple.

Doing it manually will result in a well-known TypeError if at least one of the elements isn’t a string:

```
>>> print(' '.join(['jdoe is', 42, 'years old']))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    print(','.join(['jdoe is', 42, 'years old']))
TypeError: sequence item 1: expected str instance, int found
```

It’s safer to just unpack the sequence with the star operator (*) and let print() handle type casting:

```
>>> print(*['jdoe is', 42, 'years old'])
jdoe is 42 years old
```

Unpacking is effectively the same as calling print() with individual elements of the list.

One more interesting example could be exporting data to a comma-separated values (CSV) format:

```
>>> print(1, 'Python Tricks', 'Dan Bader', sep=',')
1,Python Tricks,Dan Bader
```

This wouldn’t handle edge cases such as escaping commas correctly, but for simple use cases, it should do. The line above would show up in your terminal window. In order to save it to a file, you’d have to redirect the output. Later in this section, you’ll see how to use print() to write text to files straight from Python.

Finally, the sep parameter isn’t constrained to a single character only. You can join elements with strings of any length:

```
>>> print('node', 'child', 'child', sep=' -> ')
node -> child -> child
```

In the upcoming subsections, you’ll explore the remaining keyword arguments of the print() function.

2.2 Getting Inputs from User

You may often want to make your Python programs more interactive by responding dynamically to input from the user. Learning how to read user input from the keyboard unlocks exciting possibilities and can make your code far more useful.

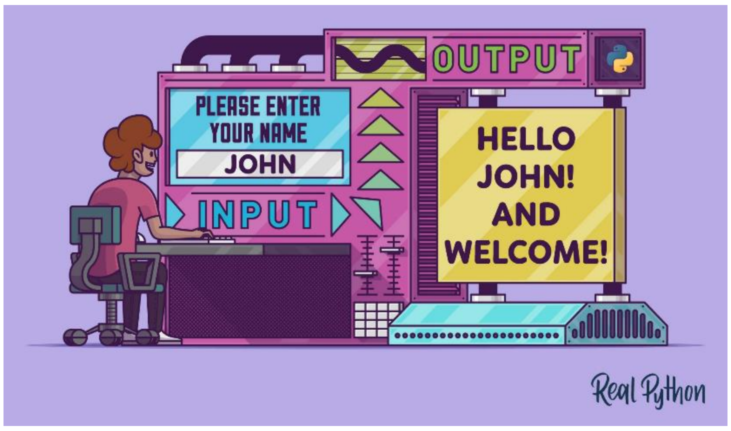
The ability to gather input from the keyboard with Python allows you to build programs that can respond uniquely based on the preferences, decisions, or data provided by different users. By fetching input and assigning it to variables, your code can react to adjustable conditions rather than just executing static logic flows. This personalized programs to individual users.

The input() function is the simplest way to get keyboard data from the user in Python. When called, it asks the user for input with a prompt that you specify, and it waits for the user to type a response and press the Enter key before continuing. This response string is returned by input() so you can save it to a variable or use it directly.

Using only Python, you can start building interactive programs that accept customizable data from the user right within the terminal. Taking user input is an essential skill that unlocks more dynamic Python coding and allows you to elevate simple scripts into personalized applications.

How to Read Keyboard Input in Python With input()

You can create robust and interactive programs by taking advantage of input(). It opens up possibilities for creating scripts that respond dynamically based on adjustable conditions, personalized data, and real-time decision-making from the user. It is an ideal option when you want to read keyboard input with Python.



Getting started is straightforward. Just launch the Python interpreter shell, and you can immediately utilize input() for basic interactivity.

To try it, first open your terminal and launch the Python REPL by typing python and hitting Enter. This will display the familiar >>> Python prompt indicating the REPL is ready. Use input(), passing any string in the parentheses:

```
>>> input("What is your name? ")
What is your name? Vincent
'Vincent'
```

Once you execute the code, the string that you specified in input() will be rendered on screen, and you'll be able to type in any response. Your response will also print to the screen once you press Enter. This is because the REPL automatically prints return values—in this case, the value returned by input().

The best practice when using input() is to assign it to a variable that you can use later in your code. For example, you can ask the user to type in their name. Assign input() to the name variable:

```
>>> name = input("What is your name? ")
What is your name? Vincent
```

This prints the prompt What is your name? and pauses, waiting for keyboard input. After the user types a name and presses Enter, the text string is stored in the name variable. This time, your input won't automatically print in the REPL, because a variable stores the value instead.

You can now use the variable in any part of your code in the same session, like printing a personalized greeting:

```
>>> print(f"Hello there, {name}!")
Hello there, Vincent!
```

Your program reacted based on the custom name that you provided. It used the name that you gave in the input() request.

This is the essential pattern when working with Python keyboard input:

- Call input() with a prompt explaining what to enter.
- Assign the result to a descriptively named variable.
- Use that variable later in your code.

By following this template, you can start building all types of interactive scripts tailored to custom data from different users. Your programs can gather input of all types, such as names, numbers, and lists, making it quite handy in processing data from your users.

Using input() processes all inputs as string literals, the users' input isn't executed as code. However, you should be wary of users' inputs and assess them before executing them in your program, using them in database queries, or otherwise trusting them.

This is just the initial step that you can take in using input() in your interactive program. There are other modifications that you can make to ensure that the function takes in all manner of data types.

Reading Specific Data Types with the Input Function

The general rule for input () is that it collects textual input and delivers strings. Your code often needs numbers, Booleans or other data types instead. For example, maybe you need to get integers for math operations, floats for calculations with decimals, or Booleans for logical conditions.

As input () returns strings, you need to convert all the input into the targeted desired data type before using it in your code. If you ask for numerical input, the outcome will still be returned as a string:

```
>>> age = input("How old are you? ")
How old are you? 35

>>> type(age)
<class 'str'>
```

The input function accepts the numerical value. When you check the age variable, its type is a string. This is the function's default behavior. But what if you need to do calculations on those numbers? You can convert the string input to integers and floats.

Luckily, Python makes it straightforward to convert input strings into any data type needed. You can pass the input () result into a type conversion function like int (), float(), or bool() to transform it on demand. For example, using the int () type conversion will convert the input to the desired integer type:

```
>>> age = int(input("How old are you? "))
How old are you? 35
```

```
>>> type(age)
<class 'int'>
```

You wrapped input () in a type conversion function, int(). This converts any numeric string into an integer. Now when you check the type of the age variable, it returns int.

You can use the same approach to convert numerical inputs to floats. For example, you might want to log the user's weight:

```
>>> weight = float(input("What is your weight in pounds? "))
What is your weight in pounds? 165

>>> weight
165.0

>>> type(weight)
<class 'float'>
```

In the example, you convert "165" to the floating-point value 165.0. You check the type of weight to confirm that the conversion worked as expected.

You'll use int() or float() whenever you need a numeric value. The input will always start as a string, but you can convert it to numbers as needed. This is an important point to note to ensure you get the desired results from your interactive program.

Handling Errors With input () in Python

Errors can occur if you wrap input() in a conversion function, and it receives a different data type than what it expected. The program will then raise a ValueError and terminate. This is not the way that you'd want your program to behave. Try out the next example to see how the program will reject a different data type input:

```
>>> age = int(input("How old are you? "))
How old are you? Thirty-five
Traceback (most recent call last):
 ...

ValueError: invalid literal for int() with base 10: 'Thirty-five'
```

This code raises a ValueError since Python can't convert "Thirty-five" into an integer. Your users might not comprehend the error.

To handle such a case, you can catch the exception before your user sees it. In the next example, you'll write a script within a Python file. Create a folder within your preferred directory.

Open your favorite editor and create a new Python file, which you should save to the folder that you just created. You can name it whatever you want. In this case, it's named integer_input.py. Enter the following code:

```
integer_input.py
while True:
    try:
        age = int(input("How old are you? "))
    except ValueError:
        print("Please enter a number for your age.")
    else:
        break

print(f"Next year, you'll be {age + 1} years old")
```

The above code has utilized the try … except code blocks in handling exceptions. The content inside the try block is the code that might raise an error. In this example, the code is trying to convert user input to an integer. The except block is what gets executed if there's an error in the corresponding try block. For example, it might catch a ValueError.

Save your file and then, in your terminal, navigate to where you've saved this Python file. You can execute it with this command:

```
PS> python integer_input.py
How old are you? Thirty-five
Please enter a number for your age.
How old are you? 35
Next year, you'll be 36 years old
```

The content of your file will load, asking you How old are you?. Now if you enter a string like "Thirty-five", your program will gracefully handle the error and give you information on why your input wasn't successful. It'll give you another chance to enter the correct values, and it'll execute to the end.

Converting input with int() and float() gives you the flexibility to accept numeric and textual input. Handling the ValueError cases makes the program more robust and user-friendly when incorrect types are entered, and it gives meaningful error messages to the user.

Reading Multiple Entries from User Input

There are scenarios in which your program would require multiple entries from a user. This can be the case when your user might have multiple answers to a question.

In Python, the list data type allows you to store and organize a sequence of elements. Lists are good at storing multiple entries from users. You can request that users enter multiple inputs, which your program can store in a list. Open your folder and create a new Python file, list_input.py. Enter the following code:

```
list_input.py
user_colors = input("Enter the three secondary colors separated by commas: ")
colors = [s.strip() for s in user_colors.split(",")]

print(f"List of colors: {colors}")
```

This Python code takes input from the user, expecting three secondary colors separated by commas. You use a list comprehension to iterate over comma-separated substrings. For each substring, the .strip() method removes any leading or trailing whitespaces. The processed list of colors is then printed out.

In your terminal, run the Python file to see the code in action:

```
PS> python list_input.py
Enter the three secondary colors separated by commas: orange, purple, green
List of colors: ['orange', 'purple', 'green']
```

Your program will place whatever the user types as a string within a list. It'll then print the list out on the terminal. In this case, the program doesn't check if the entries are correct. For example, you could type red, yellow, blue, and it'd render them on the terminal.

What if you wanted to check the entries and compare them with the correct answers before printing them on the terminal? That's possible.

You can make even more dynamic programs, such as quizzes, using input() together with lists, sets, or dictionaries. Create a new Python file called multiple_inputs.py and save it in your folder. In this file, add this code to ask for multiple inputs from the user:

```
multiple_inputs.py
print("Name at least 3 colors in the Kenyan flag.")

kenya_flag_colors = {"green", "black", "red", "white"}
user_colors = set()

while len(user_colors) < 3:
    color = input("Enter a color on the Kenyan flag: ")
    user_colors.add(color.lower())

if user_colors.issubset(kenya_flag_colors):
    print(
        "Correct! These colors are all in the Kenyan flag: "
        + ", ".join(user_colors)
    )
else:
    print(
        "Incorrect. The colors of the Kenyan flag are: "
        + ", ".join(kenya_flag_colors)
    )
```

This code asks the user to input three of the colors of the Kenyan flag, and it checks if they guessed correctly by comparing the input to the kenya_flag_colors set. It uses a while loop to repeatedly prompt the user for a color input and adds it to user_colors. The loop runs as long as user_colors has fewer than three items.

You use the .lower() string method to convert all inputs into lowercase strings. By converting each color to lowercase, you ensure a case-insensitive match during the comparison since all the colors in kenya_flag_colors are also in lowercase.

For example, if the user enters Green, the comparison will still recognize it as a correct answer due to the uniform lowercase conversion.

Once the loop ends, the code then checks if each of the user's guesses is one of the actual colors. If so, a success message is printed along with the user's colors. If not, a failure message is printed with the actual Kenyan flag colors.

Now you can run the multiple_inputs.py file in your terminal to see the code in action:

```
PS> python multiple_inputs.py
Name at least 3 colors in the Kenyan flag.
Enter a color on the Kenyan flag: Red
Enter a color on the Kenyan flag: Yellow
Enter a color on the Kenyan flag: Green
Incorrect. The colors of the Kenyan flag are: green, red, black, white
```

In these examples, you use input() with a list or a set to repeatedly prompt for user input and collect the results. You can then validate the input against expected values to provide feedback to the user.

Here's another example of how you can collect user input without prompting the user repeatedly, like the last example. It's scalable since you can add more multiple-answer questions to the program. Create another Python file and name it quiz.py. Inside the file, enter the following code:

```
quiz.py
questions = {
    "What are the four colors of the Kenyan flag? ": {
        "green", "black", "red", "white"
    },
    "What are the three colors of the French flag? ": {
        "blue", "red", "white"
    },
    "How do you spell the first three numbers in Norwegian? ": {
        "en", "to", "tre"
    },
}
```

for question, correct in questions.items():

```
while True:
        answers = {
            answer.strip().lower() for answer in input(question).split(",")
        }
        if len(answers) == len(correct):
            break
        print(f"Please enter {len(correct)} answers separated by comma")
    if answers == correct:
        print("Correct")
    else:
        print(f"No, the correct answer is {', '.join(correct)}")
```

In the above code, you ask your user questions, check their answers, and provide feedback. You define a dictionary named questions containing the questions as keys and the correct answers as values in sets. You can add more questions and answers to the dictionary.

The code loops through the dictionary items using .items(). Inside the loop, it enters a while True loop to repeatedly prompt the user to enter the right number of answers.

You convert the user's input to a set after first converting it to a list with .split(). Then you compare the length of the set to see if the user gave the expected number of answers. If not, you print a message telling the user how many answers to provide and repeat the loop. Once the number of answers is correct, you move on to the next question.

You then compare the user's answer set to the correct set using ==. If they match, you print a Correct message. Otherwise, a message shows the correct answers. You can now run the file in your terminal to see the quiz code in action:

```
PS> python quiz.py
What are the four colors of the Kenyan flag? red, white, black
Please enter 4 answers separated by comma
What are the four colors of the Kenyan flag? red, white, black, green
Correct
```

```
What are the three colors of the French flag? blue, red, green
No, the correct answer is red, blue, white
How do you spell the first three numbers in Norwegian? en, to, tre
Correct
```

Check out the quiz tutorial to learn how to create a more complete quiz application. This will also show you how to leverage input() in your bigger and more involved projects.

2.3 String Formatting

String formatting in Python allows you to create strings with placeholders that you can later replace with variables or values.

Let's jump right in, as we've got a lot to cover. In order to have a simple toy example for experimentation, let's assume you've got the following variables (or constants, really) to work with:

```
>>> errno = 50159747054
>>> name = 'Bob'
```

Based on these variables, you'd like to generate an output string containing a simple error message:

```
'Hey Bob, there is a 0xbadc0ffee error!'
```

That error could really spoil a dev's Monday morning… But we're here to discuss string formatting. So let's get to work.

String Formatting (str.format)

Python 3 introduced a new way to do string formatting that was also later back-ported to Python 2.7. This "new style" string formatting gets rid of the %-operator special syntax and makes the syntax for string formatting more regular. Formatting is now handled by calling .format() on a string object.

You can use format() to do simple positional formatting, just like you could with "old style" formatting:

```
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

Or, you can refer to your variable substitutions by name and use them in any order you want. This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to format():

```
>>> 'Hey {name}, there is a 0x{errno:x} error!'.format(
...     name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

This also shows that the syntax to format an int variable as a hexadecimal string has changed. Now you need to pass a format spec by adding a :x suffix. The format string syntax has become more powerful without complicating the simpler use cases. It pays off to read up on this string formatting mini-language in the Python documentation.

In Python 3, this "new style" string formatting is to be preferred over %-style formatting. While "old style" formatting has been de-emphasized, it has not been deprecated. It is still supported in the latest versions of Python. According to this discussion on the Python dev email list and this issue on the Python dev bug tracker, %-formatting is going to stick around for a long time to come.

Still, the official Python 3 documentation doesn't exactly recommend "old style" formatting or speak too fondly of it:

"The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the str.format() interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text." (Source)

This is why I'd personally try to stick with str.format for new code moving forward. Starting with Python 3.6, there's yet another way to format your strings. I'll tell you all about it in the next section.

String Interpolation / f-Strings (Python 3.6+)

Python 3.6 added a new string formatting approach called formatted string literals or "f-strings". This new way of formatting strings lets you use embedded Python expressions inside string constants. Here's a simple example to give you a feel for the feature:

```
>>> f'Hello, {name}!'
'Hello, Bob!'
```

As you can see, this prefixes the string constant with the letter “f“—hence the name “f-strings.” This new formatting syntax is powerful. Because you can embed arbitrary Python expressions, you can even do inline arithmetic with it. Check out this example:

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
'Five plus ten is 15 and not 30.'
```

Formatted string literals are a Python parser feature that converts f-strings into a series of string constants and expressions. They then get joined up to build the final string.

Imagine you had the following greet() function that contains an f-string:

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...
>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

When you disassemble the function and inspect what’s going on behind the scenes, you’ll see that the f-string in the function gets transformed into something similar to the following:

```
>>> def greet(name, question):
...    return "Hello, " + name + "! How's it " + question + "?"
```

The real implementation is slightly faster than that because it uses the BUILD_STRING opcode as an optimization. But functionally they’re the same:

```
>>> import dis
>>> dis.dis(greet)
  2           0 LOAD_CONST               1 ('Hello, ')
              2 LOAD_FAST                0 (name)
              4 FORMAT_VALUE             0
              6 LOAD_CONST               2 ("! How's it ")
              8 LOAD_FAST                1 (question)
```

```
        10 FORMAT_VALUE          0
        12 LOAD_CONST            3 ('?')
        14 BUILD_STRING          5
        16 RETURN_VALUE
```

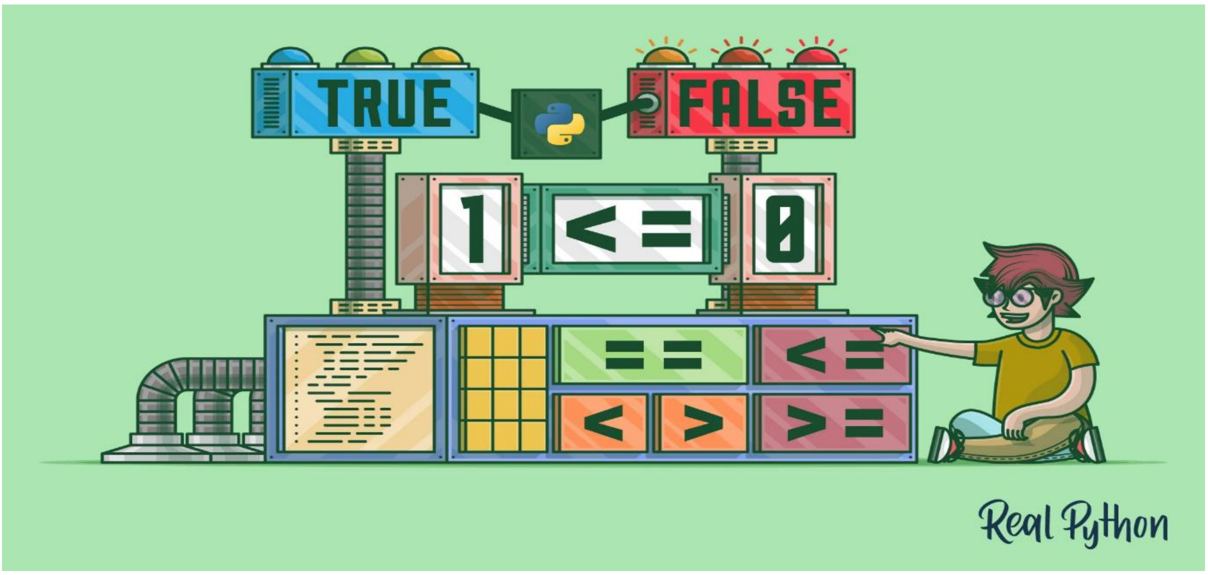
String literals also support the existing format string syntax of the str.format() method. That allows you to solve the same formatting problems we've discussed in the previous two sections:

```
>>> f"Hey {name}, there's a {errno:#x} error!"
"Hey Bob, there's a 0xbadc0ffee error!"
```

Python's new formatted string literals are similar to JavaScript's Template Literals added in ES2015. I think they're quite a nice addition to Python, and I've already started using them in my day to day (Python 3) work. You can learn more about formatted string literals in our in-depth Python f-strings tutorial.

2.4 Simple and complex decision-making approach using logical statements

In this section, you will learn how to write programs that perform different actions based on different conditions using conditional logic. Paired with functions and loops, conditional logic allows you to write complex programs that handle many different situations.



Compare Values Conditional logic is based on performing different actions depending on whether or not some expression, called a conditional, is true or false. This idea is not specific to computers. Humans use conditional logic all the time to make decisions. For example, the legal age for purchasing alcoholic beverages in the United States is 21. The statement "If you are at least 21 years old, then you may purchase a beer" is an example of conditional logic. The phrase “you are at least 21 years old” is a conditional because it may be either true or false. In computer programming, conditionals often take the form of comparing two values, such as determining if one value is greater than another, or whether or not two values are equal to each other. A standard set of symbols called boolean comparators are used to make comparisons, and most of them may already be familiar to you. The following table describes these boolean comparators:

Boolean Comparator	Example	Meaning
>	a > b	a greater than b
<	a < b	a less than b
>=	a >= b	a greater than or equal to b
<=	a <= b	a less than or equal to b
!=	a != b	a not equal to b
==	a ==b	a equal to b

The term **boolean** is derived from the last name of the English mathematician George Boole, whose works helped lay the foundations of modern computing. In Boole’s honor, conditional logic is sometimes called **boolean logic,** and conditionals are sometimes called **boolean expressions.** There is also a fundamental data type called the boolean, or bool for short, which can have only one of two values. In Python, these values are conveniently named True and False:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Note that True and False both start with capital letters. The result of evaluating a conditional is always a boolean value:

```
>>> 1 == 1
True
>>> 3 > 5
```

```
False
```

In the first example, since 1 is equal to 1, the result of 1 == 1 is True. In the second example, 3 is not greater than 5, so the result is False.

In the first example, since 1 is equal to 1, the result of 1 == 1 is True. In the second example, 3 is not greater than 5, so the result is False.

A common mistake when writing conditionals is to use the assignment operator =, instead of ==, to test whether or not two values are equal. Fortunately, Python will raise a SyntaxError if this mistake is encountered, so you'll know about it before you run your program.

You may find it helpful to think of boolean comparators as asking a question about two values. a == b asks whether or not a and b have the same value. Likewise, a != b asks whether or not a and b have different values.

Conditional expressions are not limited to comparing numbers. You may also compare values such as strings:

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

The last two examples above may look funny to you. How could one string be greater than or less than another?

The comparators < and > represent the notions of greater than and less than when used with numbers, but more generally they represent the notion of order. In this regard, "a" < "b" checks if the string "a" comes before the string "b". But how are strings ordered?

In Python, strings are ordered lexicographically, which is a fancy way to say they are ordered as they would appear in a dictionary. So you can think of "a" < "b" as asking whether or not the letter a comes before the letter b in the dictionary.

Lexicographic ordering extends to strings with two or more characters by looking at each component letter of the string:

```
>>> "apple" < "astronaut"
True
>>> "beauty" > "truth"
False
```

Since strings can contain characters other than letters of the alphabet, the ordering must extend to those other characters as well. We won't go in to the details of how characters other than letters are ordered. In practice, the < and > comparators are most often used with numbers, not strings.

Logical Operators

In addition to boolean comparators, Python has special keywords called logical operators that can be used to combine boolean expressions. There are three logical operators: and, or, and not.

Logical operators are used to construct compound logical expressions. For the most part, these have meanings similar to their meaning in the English language, although the rules regarding their use in Python are much more precise.

The and Keyword

Consider the following statements:

- Cats have four legs.
- Cats have tails.

In general, both of these statements are true.

When we combine these two statements using and, the resulting sentence "cats have four legs and cats have tails" is also a true statement. If both statements are negated, the compound statement "cats do not have four legs and cats do not have tails" is false.

Even when we mix and match false and true statements, the compound statement is false. "Cats have four legs and cats do not have tails" and "cats do not have four legs and cats have tails" are both false statements.

When two statements P and Q are combined with and, the truth value of the compound statement "P and Q" is true if and only if both P and Q are true.

Python and operator work exactly the same way. Here are four example of compound statements with and:

```
>>> 1 < 2 and 3 < 4 # Both are True
True
Both statements are True, so the combination is also True.
>>> 2 < 1 and 4 < 3 # Both are False
False
Both statements are False, so their combination is also False.
>>> 1 < 2 and 4 < 3 # Second statement is False
False 1 < 2 is True, but 4 < 3 is False, so their combination is False.
>>> 2 < 1 and 3 < 4 # First statement is False
```

```
False
```

2 < 1 is False, and 3 < 4 is True, so their combination is False.

The following table summarizes the rules for the and operator:

Combination using and	Result
True and True	True
True and False	False
False and True	False
False and False	False

You can test each of these rules in the interactive window:

```
>>> True and True
True >>> True and False
False
>>> False and True
False
>>> False and False
False
```

The or Keyword

When we use the word “or” in everyday conversation, sometimes we mean an **exclusive or**. That is, only the first option or the second option can be true.

For example, the phrase “I can stay or I can go” uses the exclusive or. I can’t both stay and go. Only one of these options can be true. In Python the or keyword is inclusive. That is, if P and Q are two expressions, the statement “P or Q” is true if any of the following are true:

- P is true
- Q is true
- Both P and Q are true

Let’s look at some examples using numerical comparisons:

```
>>> 1 < 2 or 3 < 4 # Both are True
True
>>> 2 < 1 or 4 < 3 # Both are False
False
>>> 1 < 2 or 4 < 3 # Second statement is False
True
>>> 2 < 1 or 3 < 4 # First statement is False
True
```

Note that if any part of a compound statement is True, even if the other part is False, the result is always true True. The following table summarizes these results:

Combination using or	Result
True or True	True
True or False	True
False or True	True
False or False	False

Again, you can verify all of this in the interactive window:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

The not Keyword

The not keyword reverses the truth value of a single expression:

Use of not	Result
not True You can	False
not False	True

verify this in the interactive window:

```
>>> not True
False
>>> not False
True
```

One thing to keep in mind with not, though, is that it doesn't always behave the way you might expect when combined with comparators like ==. For example, not True == False returns True, but False == not True will raise an error:

```
>>> not True == False
True
>>> False == not
True File "<stdin>", line 1
False == not True
                 ^
SyntaxError: invalid syntax
```

This happens because Python parses logical operators according to an operator precedence, just like arithmetic operators have an order of precedence in everyday math.

The order of precedence for logical and boolean operators, from highest to lowest, is described in the following table. Operators on the same row have equal precedence.

Operator Order of Precedence (Highest to Lowest)
<, <=, ==, >=, >
not
and
or

Looking again at the expression False == not True, not has a lower precedence than == in the order of operations. This means that when Python evaluates False == not True, it first tries to evaluate False == not which is syntactically incorrect.

You can avoid the SyntaxError by surrounding not True with parentheses:

```
>>> False == (not True)
True
```

Grouping expressions with parentheses is a great way to clarify which operators belong to which part of a compound expression.

Building Complex Expressions

You can combine the and, or and not keywords with True and False to create more complex expressions. Here's an example of a more complex expression:

True and not (1 != 1)

What do you think the value of this expression is? To find out, break the expression down by starting on the far right side. 1 != 1 is False, since 1 has the same value as itself. So you can simplify the above expression as follows:

True and not (False)

Now, not (False) is the same as not False, which is True. So you can simplify the above expression once more:

True and True

Finally, True and True is just True. So, after a few steps, you can see that True and not (1 != 1) evaluates to True.

When working through complicated expressions, the best strategy is to start with the most complicated part of the expression and build outward from there.

For instance, try evaluating the following expression:

("A" != "A") or not (2 >= 3)

Start by evaluating the two expressions in parentheses. "A" != "A" is False because "A" is equal to itself. 2 >= 3 is also False because 2 is smaller than 3. This gives you the following equivalent, but simpler, expression:

(False) or not (False)

Since not has a higher precedence than or, the above expression is equivalent to the following: False or (not False) not False is True, so you can simplify the expression once more:

False or True

Finally, since any compound expression with or is True if any one of the expressions on the left or right of the or is True, you can conclude that ("A" != "A") or not (2 >= 3) is True.

Grouping expressions in a compound conditional statement with parentheses improves readability. Sometimes, though, parenthesis are required to produce the expected value.

For example, upon first inspection, you may expect the following to output True, but it actually returns False:

```
>>> True and False == True and False
False
```

The reason this is False is that the == operator has a higher precedence than and, so Python interprets the expression as True and (False == True) and False. Since False == True is False, this is equivalent to True and False and False, which evaluates to False.

The following shows how to add parentheses so that the expression evaluates to True:

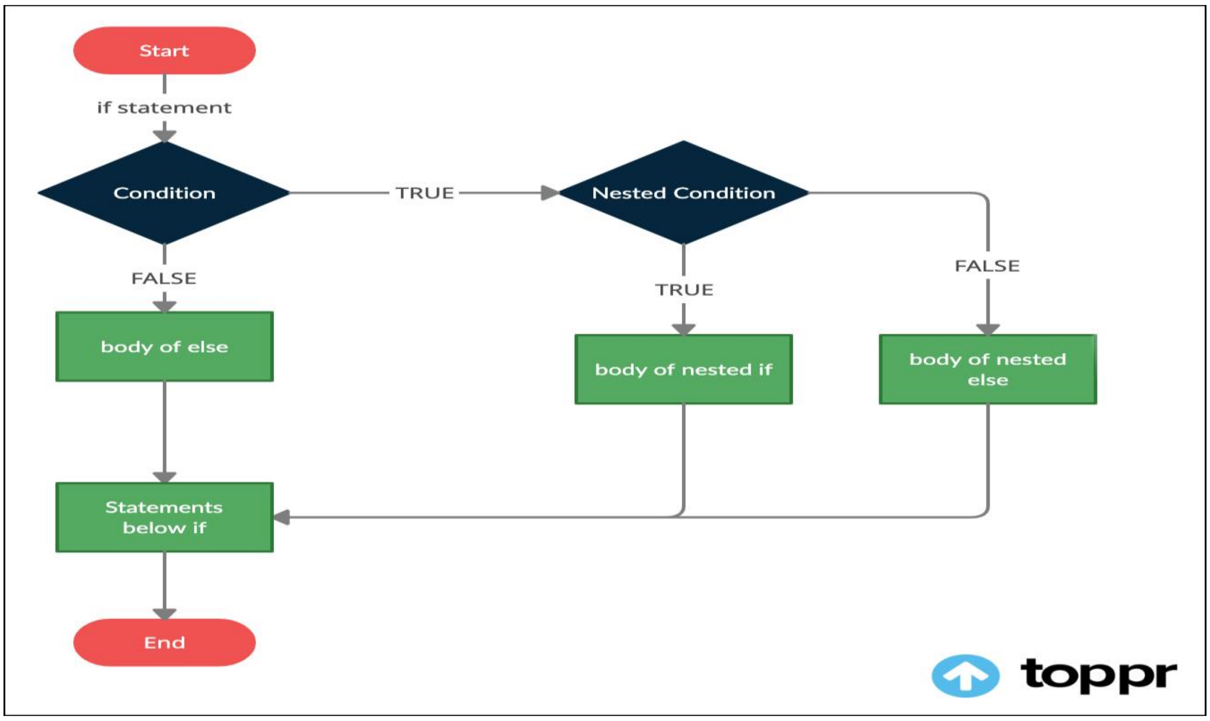
```
>>> (True and False) == (True and False)
True
```

Logical operators and boolean comparators can be confusing the first time you encounter them, so if you don't feel like the material in this section comes naturally, don't worry!

With a little bit of practice, you'll be able to make sense of what's going on and build your own compound conditional statements when you need them.

If Statement: Control the Flow of Your Program

Now that we can compare values to one other with boolean comparators and build complex conditional statements with logical operators, we can add some logic to our code so that it performs different actions for different conditions. The if Statement tells Python to only execute a portion of code if a condition is met.



For example, the following if statement will print 2 and 2 is 4 if the conditional 2 + 2 == 4 is True:

```
if 2 + 2 == 4:
    print("2 and 2 is 4")
```

In English, you can read this as “if 2 + 2 is 4, then print the string '2 and 2 is 4'.” Just like while loops, an if statement has three parts:

- The if keyword
- A test condition, followed by a colon
- An indented block of code that is executed if the test condition is True

In the above example, the test condition is 2 + 2 == 4. Since this expression is True, executing the if statement in IDLE displays the text 2 and 2 is 4.

If the test condition is False (for instance, 2 + 2 == 5), Python skips over the indented block of code and continues execution on the next non-indented line.

For example, the following if statement does not print anything:

```
if 2 + 2 == 5:
print("Is this the mirror universe?")
```

A universe where 2 + 2 == 5 is True would be pretty strange indeed!

Note Leaving off the colon (:) after the test condition in an if statement raises a SyntaxError:

```
>>> if 2 + 2 == 4
SyntaxError: invalid syntax
```

Once the indented code block in an if statement is executed, Python will continue to execute the rest of the program. Consider the following script:

```
grade = 95
if grade >= 70:
print("You passed the class!")
print("Thank you for attending.")
```

The output looks like this:

```
You passed the class!
Thank you for attending.
```

Since grade is 95, the test condition grade >= 70 is True and the string "You passed the class!" is printed. Then the rest of the code is executed and "Thank you for attending." is printed.

If you change the value of grade to 40, the output looks like this:

```
Thank you for attending.
```

The line print("Thank you for attending.") is executed whether or not grade is greater than or equal to 70 because it is after the indented code block in the if statement.

A failing student will not know that they failed if all they see from your code is the text "Thank you for attending.".

Let’s add another if statement to tell the student they did not pass if their grade is less than 70:

```
grade = 40
if grade >= 70:
print("You passed the class!")
if grade < 70:
print("You did not pass the class :(")
print("Thank you for attending.")
```

The output now looks like this:

```
You did not pass the class :(
Thank you for attending.
```

In English, we can describe an alternate case with the word “otherwise.” For instance, “If your grade is 70 or above, you pass the class. Otherwise, you do not pass the class.”

Fortunately, there is a keyword that does for Python what the word “otherwise” does in English.

The else Keyword

The else keyword is used after an if statement in order to execute some code only if the if statement's test condition is False.

The following script uses else to shorten the code in the previous script for displaying whether or not a student passed a class:

```
grade = 40
if grade >= 70:
print("You passed the class!")
else:
print("You did not pass the class :(")
print("Thank you for attending.")
```

In English, the if and else statements together read as "If the grade is at least 70, then print the string "You passed the class!"; otherwise, print the string "You did not pass the class :(".

Notice that the else keyword has no test condition, and is followed by a colon. No condition is needed, because it executes for any condition that fails the if statement's test condition.

```
SyntaxError:
>>> if 2 + 2 == 5:
... print("Who broke my math?")
... else
SyntaxError: invalid syntax
```

```
You did not pass the class :(
Thank you for attending.
```

The output from the above script is:The line that prints "Thank you for attending." still runs, even if the indented block of code after else is executed. The if and else keywords work together nicely if you only need to test a condition with exactly two states. Sometimes, you need to check three or more conditions. For that, you use elif.

The elif Keyword

The elif keyword is short for "else if" and can be used to add additional conditions after an if statement.

Just like if statements, elif statements have three parts:

- The elif keyword
- A test condition, followed by a colon
- An indented code block that is executed if the test condition evaluates to True

```
>>> if 2 + 2 == 5:
... print("Who broke my math?")
... elif 2 + 2 == 4
SyntaxError: invalid syntax
```

The following script combines if, elif, and else to print the letter grade a student earned in a class:

```
grade = 85 # 1
if grade >= 90: # 2
print("You passed the class with a A.")
elif grade >= 80: # 3
print("You passed the class with a B.")
elif grade >= 70: # 4
print("You passed the class with a C.")
else: # 5
print("You did not pass the class :(")
print("Thanks for attending.") # 6
```

Both grade >= 80 and grade >= 70 are True when grade is 85, so you might expect both elif blocks on lines 3 and 4 to be executed.

However, only the first block for which the test condition is True is executed. All remaining elif and else blocks are skipped, so executing the script has the following output:

```
You passed the class with a B.
Thanks for attending.
```

Let’s break down the execution of the script step-by-step:

- grade is assigned the value 85 in the line marked 1.
- grade >= 90 is False, so the if statement marked 2 is skipped.
- grade >= 80 is True, so the block under the elif statement in line 3 is executed, and "You passed the class with a B." is printed.
- The elif and else statements in lines 4 and 5 are skipped, since the condition for the elif statement on line 3 was met.

Finally, line 6 is executed and "Thanks for attending." is printed

The if, elif, and else keywords are some of the most commonly used keywords in the Python language. They allow you to write code that responds to different conditions with different behavior.

The if statement allows you to solve more complex problems than code without any conditional logic. You can even nest an if statement inside another one to write code that handles tremendously complex logic!

Nested if Statements

Just like for and while loops can be nested within one another, you nest an if statement inside another to create complicated decision making structures.

Consider the following scenario. Two people play a one-on-one sport against one another. You must decide which of two players wins depending on the players' scores and the sport they are playing:

- If the two players are playing basketball, the player with the greatest score wins.
- If the two players are playing golf, then the player with the lowest score wins.
- In either sport, if the two scores are equal, the game is a draw.

The following program solves this using nested if statements:

```
sport = input("Enter a sport: ")
p1_score = int(input("Enter player 1 score: "))
p2_score = int(input("Enter player 2 score: "))
# 1
if sport.lower() == "basketball":
if p1_score == p2_score:
print("The game is a draw.")
elif p1_score > p2_score:
print("Player 1 wins.")
else:
print("Player 2 wins.")
# 2
elif sport.lower() == "golf":
if p1_score == p2_score:
print("The game is a draw.")
elif p1_score < p2_score:
print("Player 1 wins.")
else:
print("Player 2 wins.")
# 3
else:
print("Unknown sport")
```

This program first asks the user to input a sport and the scores for two players.

In (#1), the string assigned to sport is converted to lowercase using .lower() and is compared to the string "basketball". This ensures that user input such as "Basketball" or

"BasketBall" all get interpreted as the same sport.

Then the players' scores are compared. If they are equal, the game is a draw. If player 1's score is larger than player 2's score, then player 1 wins the basketball game. Otherwise, player 2 wins the basketball game.

In (#2), the string assigned to sport is converted to lowercase gain and compared to the string "golf". Then the players' scores are checked again. If the two scores are equal, the game is a draw. If player 1's score is less than player 2's score, then player 1 wins. Otherwise, player 2 wins.

Finally, in (#3), if the sport variable is assigned to a string other than "basketball" or "golf", the message "Unknown sport" is displayed.

The output of the script depends on the input value. Here's a sample execution using "basketball" as the sport:

```
Enter a sport: basketball
Player 1 score: 75
Player 2 score: 64
Player 1 wins.
```

Here's the output with the same player scores and the sport changed to "golf":

```
Enter a sport: golf
Player 1 score: 75
Player 2 score: 64
Player 2 wins.
```

If you enter anything besides basketball or golf for the sport, the program displays Unknown sport.

Altogether, there are seven possible ways that the program can run, which are described in the following table:

Sport	**Score value**
"basketball"	p1_score == p2_score
"basketball"	p1_score > p2_score
"basketball"	p1_score < p2_score
"golf"	p1_score == p2_score
"golf"	p1_score > p2_score

"golf"	p1_score < p2_score
everything else	any combination

Nested if statements can create many possible ways that your code can run. If you have many deeply nested if statements (more than two levels), then the number of possible ways the code can execute grows quickly.

The complexity that results from using deeply nested if statements may make it difficult to predict how your program will behave under given conditions. For this reason, nested if statements are generally discouraged.

Let's see how we simplify the previous program by removing nested if statements.

First, regardless of the sport, the game is a draw if p1_score is equal to p2_score. So, we can move the check for equality out from the nested if statements under each sport to make a single if statement:

```
if p1_score == p2_score:
print("The game is a draw.")
elif sport.lower() == "basketball":
if p1_score > p2_score:
print("Player 1 wins.")
else:
print("Player 2 wins.")
elif sport.lower() == "golf":
if p1_score < p2_score:
print("Player 1 wins.")
else:
print("Player 2 wins.")
else:
print("Unknown sport.")
```

Now there are only six ways that the program can execute.

That's still quite a few ways. Can you think of any way to make the program simpler?

Here's one way to simplify it. Player 1 wins if the sport is basketball and their score is greater than player 2's score, or if the sport is golf and their score is less than player 2's score.

We can describe this with compound conditional expressions:

```
sport = sport.lower()
p1_wins_basketball = (sport == "basketball") and (p1_score > p2_score)
p1_wins_golf = (sport == "golf") and (p1_score < p2_score)
p1_wins = player1_wins_basketball or player1_wins_golf
```

This code is pretty dense, so let’s walk through it one step at a time.

First the string assigned to sport is converted to all lowercase so that we can compare the value to other strings without worrying about errors due to case.

On the next line, we have a structure that might look a little strange. There is an assignment operator (=) followed by an expression with the equality comparator (==). This line evaluates the following compound logical expression and assigns its value to the p1_wins_basketball variable:

```
(sport == "basketball") and (p1_score > p2_score)
```

If sport is "basketball" and player 1’s score is larger than player 2’s score, then p1_wins_basketball is True

Next, a similar operation is done for the p1_wins_golf variable. If score 208 8.3. Control the Flow of Your Program is "golf" and player 1’s score is less than player 2’s score, then p1_- wins_golf is True

Finally, p1_wins will be True if player 1 wins the basketball game or the golf game, and will be False otherwise.

Using this code, you can simplify the program quite a bit:

```
if p1_score == p2_score:
print("The game is a draw.")
elif (sport.lower() == "basketball") or (sport.lower() == "golf"):
sport = sport.lower()
p1_wins_basketball = (sport == "basketball") and (p1_score > p2_score)
p1_wins_golf = (sport == "golf") and (p1_score < p2_score)
p1_wins = p1_wins_basketball or p1_wins_golf
if p1_wins:
print("Player 1 wins.")
else:
print("Player 2 wins.")
else:
print("Unknown sport")
```

In this revised version of the program, there are only four ways the program can execute, and the code is easier to understand.

Nested if statements are sometimes necessary. However, if you find yourself writing lots of nested if statements, it might be a good idea to stop and think about how you might simplify your code.

Self-Check Sheet -2: Apply Input and Output Methods in python

Q1: How do you print a message with parameters in Python?

Q2: How do you take input from a user in Python?

Q 3: How can you format a string to display two decimal places in Python?

Q 4: Write a Python code to check if a given number is positive, negative, or zero.

Q5: How do you combine logical operators to make a complex decision in Python?

Q6: Write a code snippet to check if a number is within a specific range (10-20 inclusive).

Answer Sheet 2: Apply Input and Output Methods in Python

Q1: How do you print a message with parameters in Python?

Answer:
To print a message with parameters, use formatted strings, such as f-strings or the format() method:

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

Q2: How do you take input from a user in Python?

Answer:
Use the input() function to get user input. The input is stored as a string

```
user_name = input("Enter your name: ")
print(f"Hello, {user_name}!")
```

Q3: How can you format a string to display two decimal places in Python?
Answer:
You can use an f-string or the format() method to format numbers to two decimal places

```
pi = 3.14159
print(f"Pi rounded to two decimal places is {pi:.2f}")
```

Q4: Write a Python code to check if a given number is positive, negative, or zero.
Answer:
Use an if, elif, and else statement to check the conditions.

```
num = float(input("Enter a number: "))
if num > 0:
   print("The number is positive.")
elif num < 0:
   print("The number is negative.")
```

```
else:
    print("The number is zero.")
```

Q5: How do you combine logical operators to make a complex decision in Python?

Answer:
Use and, or, and not to create complex conditions.

```
age = int(input("Enter your age: "))
has_license = input("Do you have a driving license? (yes/no): ").lower() == 'yes'
if age >= 18 and has_license:
    print("You are eligible to drive.")
else:
    print("You are not eligible to drive.")
```

Q6: Write a code snippet to check if a number is within a specific range (10-20 inclusive).

Answer:
Use a logical expression with and to define the range.

```
num = int(input("Enter a number: "))
if 10 <= num <= 20:
    print("The number is within the range.")
else:
    print("The number is out of the range.")
```

Task Sheet - 2.1: Printing with Parameters

Task: Write a function greet user that takes a name as a parameter and prints a greeting message, "Hello, [name]! Welcome to the system."

Answer:

```python
def greet_user(name):
    print(f"Hello, {name}! Welcome to the system.")

# Example usage:
greet_user("Alice")
```

Task Sheet - 2.2: String Formatting

Task: Write a program that takes the user's name and age as input, then prints: "Hello, [name]! You are [age] years old."

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello, {}! You are {} years old.".format(name, age))
```

Example Input and Output :

```
Enter your name: John
Enter your age: 25
Hello, John! You are 25 years old.
```

Task Sheet - 2.3: Complex Expressions

Task: Write a program that takes two numbers as input. If both numbers are positive, print "Both are positive." If both are negative, print "Both are negative." If one is positive and the other is negative, print "Mixed signs."

```python
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))

if num1 > 0 and num2 > 0:
    print("Both are positive.")
elif num1 < 0 and num2 < 0:
    print("Both are negative.")
else:
    print("Mixed signs.")
```

Example Input and Output:

```
Enter the first number: -5
Enter the second number: 3
Mixed signs
```

Job Sheet 2: Student Grading System

Create a program that takes the details of students (name, marks in three subjects) and calculates their total marks, average, and grade based on specific criteria. The program should perform the following:

1. **Get the student’s name** and **marks for three subjects** as inputs.
2. **Calculate the total marks** and **average marks**.
3. Determine the **grade** based on the average:
   - If the average is **90 or above**, grade is **A**.
   - If the average is **80–89**, grade is **B**.
   - If the average is **70–79**, grade is **C**.
   - If the average is **60–69**, grade is **D**.
   - If the average is below **60**, grade is **F**.
4. Print a **formatted summary** of the student's information, including name, total marks, average, and grade.
5. **Bonus**: Check if the student qualifies for an **Honor Roll**. To qualify:
   - The average must be **90 or above**.
   - No individual subject mark can be below **85**.

Solution:

```
def calculate_grade(average):
    """Determine grade based on average score."""
    if average >= 90:
        return 'A'
    elif average >= 80:
        return 'B'
    elif average >= 70:
        return 'C'
    elif average >= 60:
        return 'D'
    else:
        return 'F'

def honor_roll_eligibility(marks, average):
    """Check if student is eligible for the Honor Roll."""
    if average >= 90 and all(mark >= 85 for mark in marks):
```

```python
        return True
    return False

def display_student_summary(name, total, average, grade, honor_roll):
    """Print a formatted summary of the student's performance."""
    print("\n--- Student Performance Summary ---")
    print(f"Name: {name}")
    print(f"Total Marks: {total}")
    print(f"Average Marks: {average:.2f}")
    print(f"Grade: {grade}")
    if honor_roll:
        print("Congratulations! You have qualified for the Honor Roll.")
    else:
        print("You did not qualify for the Honor Roll this time.")
    print("-----------------------------------\n")

# Main program
print("Welcome to the Student Grading System!")

# Getting inputs from the user
name = input("Enter the student's name: ")
marks = []
for i in range(1, 4):
    mark = float(input(f"Enter marks for subject {i} (0–100): "))
    marks.append(mark)

# Calculating total and average marks
total_marks = sum(marks)
average_marks = total_marks / len(marks)

# Determining the grade and honor roll eligibility
grade = calculate_grade(average_marks)
honor_roll = honor_roll_eligibility(marks, average_marks)

# Displaying the formatted summary with parameters
display_student_summary(name, total_marks, average_marks, grade, honor_roll)
```

Explanation of Code Components:

1. **Function Definitions:**
   - calculate_grade(average): Determines the grade based on the average marks using logical conditions.
   - honor_roll_eligibility(marks, average): Checks if the student qualifies for the Honor Roll by ensuring all marks are **85 or above** and the average is **90 or higher**.
   - display_student_summary(...): Prints the summary of the student's information with formatted details, displaying either a congratulatory message or indicating that they didn't qualify for the Honor Roll.
2. **Main Program Logic:**
   - **Input**: Prompts the user for the student's name and three subject marks.
   - **Calculations**: Computes the total and average marks.
   - **Decision-Making**:
     - Uses an if-elif structure in calculate_grade to determine the grade.
     - Checks for Honor Roll eligibility using logical operators in honor_roll_eligibility.
   - **Output**: Calls display_student_summary to print a well-formatted summary.

Example Run

```
Welcome to the Student Grading System!
Enter the student's name: John Doe
Enter marks for subject 1 (0–100): 95
Enter marks for subject 2 (0–100): 88
Enter marks for subject 3 (0–100): 91

--- Student Performance Summary ---
Name: John Doe
Total Marks: 274
Average Marks: 91.33
Grade: A
Congratulations! You have qualified for the Honor Roll.
----------------------------------
```

Learning Outcome 3: Solve Problem Associated with Loop

Condition and Resource	1. Actual workplace or training environment 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil and Eraser 7. Internet Facilities 8. Whiteboard and Marker 9. Imaging Device (Digital camera, scanner etc.)
Content	▪ Interpret loops are interpreted o “for” loop o “while” loop ▪ Exercise problems associated with loops ▪ Apply advanced data storage techniques are applied in python o Indexing in list and dictionary o Create, update and delete list and dictionary elements o Basic operations on list and dictionary elements
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning

Learning Experience 3: Solve Problem Associated with Loop

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Solve Problem Associated with Loop'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Work with Python basic"	2. Read Information sheet 3: Solve Problem Associated with Loop 3. Answer Self-check 3: Solve Problem Associated with Loop 4. Check your answer with Answer key 3: Solve Problem Associated with Loop
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet 3.1: Printing Even Numbers Using Loops Task Sheet 3.2: Calculating the Sum of List Elements Task Sheet 3.3: Dictionary Operations Job Sheet 3: Inventory Management System

Information Sheet 3: Solve Problem Associated with Loop

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

3.1 Interpret loops
3.2 Exercise problems associated with loops
3.3 Apply advanced data storage techniques

3.1 Loops

One of the great things about computers is that you can make them do the same thing over and over again, and they rarely complain or get tired.

A loop is a block of code that gets repeated over and over again either a specified number of times or until some condition is met. There are two kinds of loops in Python: while loops and for loops. In this section, you'll learn how to use both.

Let's start by looking at how while loops work.

The while Loop while loops repeat a section of code while some condition is true. There are two parts to every while loop:

- The while statement starts with the while keyword, followed by a test condition, and ends with a colon (:).
- The loop body contains the code that gets repeated at each step of the loop. Each line is indented four spaces.

When a while loop is executed, Python evaluates the test condition and determines if it is true or false. If the test condition is true, then the code in the loop body is executed. Otherwise, the code in the body is skipped and the rest of the program is executed.

If the test condition is true and the body of the loop is executed, then once Python reaches the end of the body, it returns to the while statement and re-evaluates the test condition. If the test condition is still true, the body is executed again. If it is false, the body is skipped.

This process repeats over and over until the test condition fails, causing Python to loop over the code in the body of the while loop.

Let's look at an example. Type the following code into the interactive window:

```
>>> n = 1
>>> while n < 5:
... print(n)
... n = n + 1
...
1
2
3
4
```

First, the integer 1 is assigned to the variable n. Then a while loop is created with the test condition n < 5, which checks whether or not the value of n is less than 5.

If n is less than 5, the body of the loop is executed. There are two lines of code in the loop body. In the first line, the value of n is printed on the screen, and then n is incremented by 1 in the second line.

The loop execution takes place in five steps, described in the following table:

Step #	Value of n	Test Condition	What Happens
1	1	1 < 5 (true)	1 printed; n incremented to 2
2	2	2 < 5 (true)	2 printed; n incremented to 3
3	3	3 < 5 (true)	3 printed; n incremented to 4
4	4	4 < 5 (true)	4 printed; n incremented to 5
5	5	5 < 5 (false)	Nothing printed; loop ends.

If you aren’t careful, you can create an inpnite loop. This happens when the test condition is always true. An infinite loop never terminates. The loop body keeps repeating forever.

Here’s an example of an infinite loop:

```
>>> n = 1
>>> while n < 5:
... print(n)
...
```

The only difference between this while loop and the previous one is that n is never incremented in the loop body. At each step of the loop, n is equal to 1. That means the test condition n < 5 is always true, and the number 1 is printed over and over again forever.

Infinite loops aren’t inherently bad. Sometimes they are exactly the kind of loop you need. For example, code that interacts with hardware may use an infinite loop to constantly check whether or not a button or switch has been activated.

If you run a program that enters an infinite loop, you can force Python to quit by pressing Ctrl+C. Python stops running the program and raises a KeyboardInterrupt error:

```
Traceback (most recent call last):
File "<pyshell#8>", line 2, in <module>
print(n)
KeyboardInterrupt
```

Let’s look at an example of a while loop in practice. One use of a while loop is to check whether or not user input meets some condition and, if not, repeatedly ask the user for new input until valid input is received.

For instance, the following program continuously asks a user for a positive number until a positive number is entered:

```
num = float(input("Enter a positive number: "))
while num <= 0:
print("That's not a positive number!")
num = float(input("Enter a positive number: "))
```

First, the user is prompted to enter a positive number. The test condition num <= 0 determines whether or not num is less than or equal to 0.

If num is positive, then the test condition fails. The body of the loop is skipped and the program ends.

Otherwise, if num is 0 or negative, the body of the loop executes. The user is notified that their input was incorrect, and they are prompted again to enter a positive number.

while loops are perfect for repeating a section of code while some condition is met. They aren’t well-suited, however, for repeating a section of code a specific number of times.

The for Loop

A for loop executes a section of code once for each item in a collection of items. The number of times that the code is executed is determined by the number of items in the collection.

Like its while counterpart, the for loop has two main parts:

- The for statement begins with the for keyword, followed by a membership expression, and ends in a colon (:).
- The loop body contains the code to be executed at each step of the loop, and is indented four spaces.

Let’s look at an example. The following for loop prints each letter of the string "Python" one at a time:

```
for letter in "Python":
print(letter)
```

In this example, the for statement is for letter in "Python". The membership expression is letter in "Python".

At each step of the loop, the variable letter is assigned the next letter in the string "Python", and then the value of letter is printed.

The loops runs once for each character in the string "Python", so the loop body executes six times. The following table summarizes the execution of this for loop:

Step #	Value of letter	What Happens
1	"P"	P is printed
2	"y"	y is printed
3	"t"	t is printed
4	"h"	h is printed
5	"o"	o is printed
6	"n"	n is printed

To see why for loops are better for looping over collections of items, let’s re-write the for loop in previous example as a while loop.

To do so, we can use a variable to store the index of the next character in the string. At each step of the loop, we’ll print out the character at the current index and then increment the index.

The loop will stop once the value of the index variable is equal to the length of the string. Remember, indices start at 0, so the last index of the string "Python" is 5.

Here’s how you might write that code:

```
word = "Python"
index = 0
while index < len(word):
print(word[index])
index = index + 1
```

That’s significantly more complex than the for loop version!

Not only is the for loop less complex, the code itself looks more natural. It more closely resembles how you might describe the loop in English.

You may sometimes hear people describe some code as being particularly “Pythonic.” The term Pythonic is generally used to describe code that is clear, concise, and uses Python’s built-in features to its advantage. In these terms, using a for loop to loop over a collection of items is more Pythonic than using a while loop.

Sometimes it’s useful to loop over a range of numbers. Python has a handy built-in function range() that produces just that — a range of numbers!

For example, range(3) returns the range of integers starting with 0 and up to, but not including, 3. That is, range(3) is the range of numbers 0, 1, and 2.

You can use range(n), where n is any positive number, to execute a loop exactly n times. For instance, the following for loop prints the string "Python" three times:

```
for n in range(3):
print("Python")
```

You can also give a range a starting point. For example, range(1, 5) is the range of numbers 1, 2, 3, and 4. The first argument is the starting number, and the second argument is the endpoint, which is not included in the range.

Using the two-argument version of range(), the following for loop prints the square of every number starting with 10 and up to, but not including, 20:

```
for n in range(10, 20):
print(n * n)
```

Let’s look at a practical example. The following program asks the user to input an amount and then displays how to split that amount between 2, 3, 4, and 5 people:

```
amount = float(input("Enter an amount: "))
for num_people in range(2, 6):
print(f"{num_people} people: ${amount / num_people:,.2f} each")
```

The for loop loops over the number 2, 3, 4, and 5, and prints the number of people and the amount each person should pay. The formatting specifier ,.2f is used to format the amount as a fixed-point number rounded to two decimal places and commas every three digits.

Running the program with the input 10 produces the following output:

```
Enter an amount: 10
2 people: $5.00 each
3 people: $3.33 each
4 people: $2.50 each
5 people: $2.00 each
```

for loops are generally used more often than while loops in Python. Most of the time, a for loop is more concise and easier to read than an equivalent while loop.

Nested Loops

As long as you indent the code correctly, you can even put loops inside of other loops.

Type the following into IDLE’s interactive window:

```
for n in range(1, 4):
for j in range(4, 7):
print(f"n = {n} and j = {j}")
```

When Python enters the body of the first for loop, the variable n is assigned the value 1. Then the body of the second for loop is executed and j is assigned the value 4. The first thing printed is n = 1 and j = 4.

After executing the print() function, Python returns to the inner for loop, assigns to j the value of 5, and then prints n = 1 and j = 5. Python doesn’t return the outer for loop because the inner for loop, which is inside the body of the outer for loop, isn’t done executing.

Next, j is assigned the value 6 and Python prints n = 1 and j = 6. At this point, the inner for loop is done executing, so control returns to the outer for loop.

The variable n gets assigned the value 2, and the inner for loop executes a second time. That is, j is assigned the value 4 and n = 2 and j = 4 is printed to the console.

The two loops continue to execute in this fashion, and the final output looks like this:

```
n = 1 and j = 4
n = 1 and j = 5
n = 1 and j = 6
```

```
n = 2 and j = 4
n = 2 and j = 5
n = 2 and j = 6
n = 3 and j = 4
n = 3 and j = 5
n = 3 and j = 6
```

A loop inside of another loop is called a nested loop, and they come up more often than you might expect. You can nest while loops inside of for loops, and vice versa, and even nest loops more than two levels deep!

Nesting loops inherently increases the complexity of your code, as you can see by the dramatic increase in the number of steps run in the previous example compared to examples with a single for loop. Using nested loops is sometimes the only way to get something done, but too many nested loops can have a negative effect on a program's performance

Loops are a powerful tool. They tap into one of the greatest advantages computers provide as tools for computation: the ability to repeat the same task a vast number of times without tiring and without complaining.

The While Loop

A while loop in Python repeatedly executes a block of code as long as a given condition is True. Here's the general syntax:

```
while condition:
   # Code to execute as long as the condition is True
```

Example: Simple Counter

This loop will print numbers from 1 to 5.

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

3.2 Problems Associated with Loops

Problem 1: Write a for loop that prints out the integers 2 through 10, each on a new line, by using the range() function.

Code:

```
for i in range(2, 11):
    print(i)
```

Problem 2: Use a while loop that prints out the integers 2 through 10 (Hint: You'll need to create a new integer first.)

Code:

```
i = 2
while i < 11:
    print(i)
    i = i + 1
```

Problem 3: Write a function called doubles() that takes one number as its input and doubles that number. Then use the doubles() function in a loop to double the number 2 three times, displaying each result on a separate line. Here is some sample output: 4 8 16Code:

```
def doubles(num):
    """Return the result of multiplying an input number by 2."""
    return num * 2


# Call doubles() to double the number 2 three times
my_num = 2
for i in range(0, 3):
    my_num = doubles(my_num)
    print(my_num)
```

3.3 Advanced Data Storage technique

In Python, both lists and dictionaries are categorized as mutable and composite data types.

a. Lists

Lists are one of the most versatile data structures in Python. They allow you to store multiple items in a single variable, making it easy to manage collections of related data. Lists can contain elements of different data types, including numbers, strings, and even other lists.

Key Features of Lists:

- **Ordered**: The items in a list maintain their order, meaning that the position of each item is fixed and can be accessed using an index.
- **Mutable**: Lists can be modified after creation. You can add, remove, or change items at any time.
- **Dynamic Size**: Lists can grow or shrink in size, allowing for flexible data management.

Common Operations:

- Accessing elements using indices (e.g., my_list[0]).
- Adding elements with append() or insert().
- Removing elements with remove() or pop().
- Slicing lists to get sublists (e.g., my_list[1:3]).

Concise Overview of Lists and Their Operations

The list data structure is another sequence type in Python. Just like strings and tuples, lists contain items that are indexed by integers, starting with 0.

On the surface, lists look and behave a lot like tuples. You can use index and slicing notation with lists, check for the existence of an element using in, and iterate over lists with a for loop

Unlike tuples, however, lists are mutable, meaning you can change the value at an index even after the list has been created.

In this section, you will learn how to create lists and compare them with tuples.

Creating Lists

A list literal looks almost exactly like a tuple literal, except that it is surrounded with square brackets ([and]) instead of parentheses:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> type(colors)
<class 'list'>
```

When you inspect a list, Python displays it as a list literal:

```
>>> colors
['red', 'yellow', 'green', 'blue']
```

Like tuples, list values are not required to be of the same type. The list literal ["one", 2, 3.0] is perfectly valid.

Aside from list literals, you can also use the list() built-in to create a new list object from any other sequence. For instance, the tuple (1, 2, 3) can be passed to list() to create the list [1, 2, 3]:

```
>>> list((1, 2, 3))
[1, 2, 3]
```

You can even create a list from a string:

```
>>> list("Python")
['P', 'y', 't', 'h', 'o', 'n']
```

Each letter in the string becomes an element of the list.

There is a more useful way to create a list from a string. You can create a list from a string of a comma-separated list of items using the string object’s .split() method:

```
>>> groceries = "eggs, milk, cheese"
>>> grocery_list = groceries.split(", ")
>>> grocery_list
['eggs', 'milk', 'cheese']
```

The string argument passed to .split() is called the separator. By changing the separator you can split strings into lists in numerous ways:

```
>>> # Split string on semicolons
>>> "a;b;c".split(";")
['a', 'b', 'c']
>>> # Split string on spaces
>>> "The quick brown fox".split(" ")
['The', 'quick', 'brown', 'fox']
>>> # Split string on multiple characters
>>> "abbaabba".split("ba")
['ab', 'ab', '']
```

In the last example above, the string is split around occurrences of the substring "ba", which occurs first at index 2 and again at index 6. The separator has two characters, only the characters at indices 1, 2, 5, and 6 become elements of the list.

.split() always returns a string whose length is one more than the number of separators contained in the string. The string "abbaabba" contains two instances of the separator "ba" so the list returned by split() has three elements. Since the third separator isn’t followed by any other characters, the third element of the list is set to the empty string.

If the separator is not contained in the string at all, .split() returns a list with the string as its only element:

```
>>> "abbaabba".split("c")
['abbaabba']
```

In all, you’ve seen three ways to create a list:

- A list literal
- The list() built-in
- The string .split() method

Lists support all of the same operations supported by tuples.

Basic List Operations

Indexing and slicing operations work on lists the same way they do on tuples.

You can access list elements using index notation:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1]
2
```

You can create a new list from an existing once using slice notation:

```
>>> numbers[1:3]
[2, 3]
```

You can check for the existence of list elements using the in operator:

```
>>> # Check existence of an element
>>> "Bob" in numbers
False
```

Because lists are iterable, you can iterate over them with a for loop.

```
>>> # Print only the even numbers in the list
>>> for number in numbers:
... if number % 2 == 0:
... print(number)
...
2
4
```

The major difference between lists and tuples is that elements of lists may be changed, but elements of tuples can not.

Changing Elements in a List

Think of a list as a sequence of numbered slots. Each slot holds a value, and every slot must be filled at all times, but you can swap out the value in a given slot with a new one whenever you want.

The ability to swap values in a list for other values is called mutability. Lists are mutable. The elements of tuples may not be swapped for new values, so tuples are said to be immutable.

To swap a value in a list with another, assign the new value to a slot using index notation:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[0] = "burgundy"
```

The value at index 0 changes from "red" to "burgundy":

```
>>> colors
['burgundy', 'yellow', 'green', 'blue']
```

You can change several values in a list at once with a slice assignment:

```
>>> colors[1:3] = ["orange", "magenta"]
>>> colors
['burgundy', 'orange', 'magenta', 'blue']
```

colors[1:3] selects the slots with indices 1 and 2. The values in these slots are assigned to "orange" and "magenta", respectively.

The list assigned to a slice does not need to have the same length as the slice. For instance, you can assign a list of three elements to a slice with two elements:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[1:3] = ["orange", "magenta", "aqua"]
>>> colors
['red', 'orange', 'magenta', 'aqua', 'blue']
```

The values "orange" and "magenta" replace the original values "yellow" and "green" in colors at the indices 1 and 2. Then a new slot is created at index 4 and "blue" is assigned to this index. Finally, "aqua" is assigned to index 3.

When the length of the list being assigned to the slice is less than the length of the slice, the overall length of the original list is reduced:

```
>>> colors
['red', 'orange', 'magenta', 'aqua', 'blue']
```

```
>>> colors[1:4] = ["yellow", "green"]
>>> colors
['red', 'yellow', 'green', 'blue']
```

The values "yellow" and "green" replace the values "orange" and "magenta" in colors at the indices 1 and 2. Then the value at index 3 is replaced with the value "blue". Finally, the slot at index 4 is removed from colors entirely.

The above examples show how to change, or mutate, lists using index and slice notation. There are also several list methods that you can use to mutate a list.

List Methods For Adding and Removing Elements

Although you can add and remove elements with slice notation, list methods provide a more natural and readable way to mutate a list.

We'll look at several list methods, starting with how to insert a single value into a list at a specified index.

`list.insert()`

The list.insert() method is used to insert a single new value into a list. It takes two parameters, an index i and a value x, and inserts the value x at index i in the list.

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> # Insert "orange" into the second position
>>> colors.insert(1, "orange")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue']
```

There are a couple of important observations to make about this example.

The first observation applies to all list methods. To use them, you first write the name of the list you want to manipulate, followed by a dot (.) and then the name of the list method.

So, to use insert() on the colors list, you must write colors.insert(). This works just like string and number methods do.

Next, notice that when the value "orange" is inserted at the index 1, the value "yellow" and all following values are shifted to the right.

If the value for the index parameter of .insert() is larger than the greatest index in the list, the value is inserted at the end of the list:

```
>>> colors.insert(10, "violet")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'violet']
```

Here the value "violet" is actually inserted at index 5, even though .insert() was called with 10 for the index.

You can also use negative indices with .insert():

```
>>> colors.insert(-1, "indigo")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

This inserts "indigo" into the slot at index -1 which is the last element of the list. The value "violet" is shifted to the right by one slot.

When you .insert() an item into a list, you do not need to assign the result to the original list. For example, the following code actually erases the colors list:

```
>>> colors = colors.insert(-1, "indigo")
>>> print(colors)
```

None .insert() is said to alter colors in place.

This is true for all list methods that do not return a value.

If you can insert a value at a specified index, it only makes sense that you can also remove an element at a specified index.

```
list.pop()
```

The list.pop() method takes one parameter, an index i, and removes the value from the list at that index. The value that is removed is returned by the method:

```
>>> color = colors.pop(3)
>>> color
'green'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Here, the value "green" at index 3 is removed and assigned to the variable color. When you inspect the colors list, you can see that the string "green" has indeed been removed. 248 9.2. Lists Are Mutable Sequences Unlike .insert(), Python raises an IndexError if you pass to .pop() an argument larger than the last index:

```
>>> colors.pop(10)
Traceback (most recent call last):
File "<pyshell#0>", line 1, in <module>
colors.pop(10)


IndexError: pop index out of range
```

Negative indices also work with .pop():

```
>>> colors.pop(-1)
'violet'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

If you do not pass a value to .pop(), it removes the last item in the list:

```
>>> colors.pop()
'indigo'
>>> colors
['red', 'orange', 'yellow', 'blue']
```

This way of removing the final element, by calling .pop() with no specified index, is generally considered the most Pythonic.

`list.append()`

The list.append() method is used to append an new element to the end of a list:

```
>>> colors.append("indigo")
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

After calling .append(), the length of the list increases by one and the value "indigo" is inserted into the final slot. Note that .append() alters the list in place, just like .insert().

.append() is equivalent to inserting an element at an index greater than or equal to the length of the list. The above example could also have been written as follows:

```
>>> colors.insert(len(colors), "indigo")
```

this way, and is generally considered the more Pythonic way of added an element to the end of a list.

```
list.extend()
The list.extend() method is used to add several new elements to the end of a list:
>>> colors.extend(["violet", "ultraviolet"])
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet', 'ultraviolet']
```

.extend() takes a single parameter that must be an iterable type. The elements of the iterable are appended to the list in the same order that they appear in the argument passed to .extend().\

Just like .insert() and .append(), .extend() alters the list in place.

Typically, the argument passed to .extend() is another list, but it could also be a tuple. For example, the above example could be written as follows:

```
>>> colors.extend(("violet", "ultraviolet"))
```

The four list methods discussed in this section make up the most common methods used with lists. The following table serves to recap everything you have seen here:

List Method	Description
.insert(i, x)	Insert the value x at index i
.append(x)	Insert the value x at the end of the list
.extend(iterable)	Insert all the values of iterable at the end of the list, in order
.pop(i)	Remove and return the element at index i

In addition to list methods, Python has a couple of useful built-in functions for working with lists of numbers.

Lists of Numbers

One very common operation with lists of numbers is to add up all the values to get the total.

You can do this with a for loop:

```
>>> nums = [1, 2, 3, 4, 5]
>>> total = 0
>>> for number in nums:
... total = total + number
...
>>> total
15
```

First you initialize the variable total to 0, and then loop over each number is nums and add it to total, finally arriving at the value 15.

Although this for loop is straightforward, there is a much more succinct way of doing this in Python:

```
>>> sum([1, 2, 3, 4, 5])
15
```

The built-in sum() function takes a list as an argument and returns the total of all the values in the list.

If the list passed to sum() contains any values that aren’t numeric, a TypeError is raised:

```
>>> sum([1, 2, 3, "four", 5])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Besides sum(), there are two other useful built-in functions for working with lists of numbers: min() and max(). These functions return the minimum and maximum values in the list, respectively:

```
>>> min([1, 2, 3, 4, 5 ])
1
>>> max([1, 2, 3, 4, 5])
5
Note that sum(), min(), and max() also work with tuples:
>>> sum((1, 2, 3, 4, 5))
15
>>> min((1, 2, 3, 4, 5))
1
>>> max((1, 2, 3, 4, 5))
5
```

The fact that sum(), min(), and max() are all built-in to Python tells you that they are used frequently. Chances are, you'll find yourself using them quite a bit in your own programs!

List Comprehensions

Yet another way to create a list from an existing iterable is with a list **comprehension**:

```
>>> numbers = (1, 2, 3, 4, 5)
>>> squares = [num**2 for num in numbers]
>>> squares
[1, 4, 9, 16, 25]
```

A list comprehension is a short-hand for a for loop. In the example above, a tuple literal containing five numbers is created and assigned to the numbers variable. On the second line, a list comprehension loops over each number in numbers, squares each number, and adds it to a new list called squares.

To create the sqaures list using a traditional for loop involves first creating an empty list, looping over the numbers in numbers, and appending the square of each number to the list:

```
>>> squares = []
>>> for num in numbers:
... sqaures.append(num**2)
...
>>> squares
[1, 4, 9, 16, 25]
```

List comprehensions are commonly used to convert values in one list to a different type. For instance, suppose you needed to convert a list of strings containing floating point values to a list of float objects. The following list comprehensions achieves this:

```
>>> str_numbers = ["1.5", "2.3", "5.25"]
>>> float_numbers = [float(value) for value in str_numbers]
>>> float_numbers
[1.5, 2.3, 5.25]
```

List comprehensions are not unique to Python, but they are one of its many beloved features. If you find yourself creating an empty list, looping over some other iterable, and appending new items to the list, then chances are you can replace your code with a list comprehension!

Nesting, Copying, and Sorting Tuples and Lists

Now that you have learned what tuples and lists are, how to create them, and some basic operations with them, let's look at three more concepts:

- Nesting
- Copying
- Sorting

Nesting Lists and Tuples

Lists and tuples can contain values of any type. That means lists and tuples can contain lists and tuples as values. A nested list, or nested tuple, is a list or tuple that is contained as a value in another list or tuple.

For example, the following list has two values, both of which are other lists:

```
>>> two_by_two = [[1, 2], [3, 4]]
>>> # two_by_two has length 2
>>> len(two_by_two)
2
>>> # Both elements of two_by_two are lists
>>> two_by_two[0]
[1, 2]
>>> two_by_two[1]
[3, 4]
```

Since two_by_two[1] returns the list [3, 4], you can use double index notation to access an element in the nested list:

```
>>> two_by_two[1][0]
3
```

First, Python evaluates two_by_two[1] and returns [3, 4]. Then Python evaluates [3, 4][0] and returns the first element 3.

In very loose terms, you can think of a list of lists or a tuple of tuples as a sort of table with rows and columns.

The two_by_two list has two rows, [1, 2] and [3, 4]. The columns are made of of the corresponding elements of each row, so the first columns contains the elements 1 and 3, and the second column contains the elements 2 and 4.

This table analogy is only an informal way of thinking about a list of lists, though. For example, there is no requirement that all the lists in a list of lists have the same length, in which case this table analogy starts to break down.

Copying a List

Sometimes you need to copy one list into another list. However, you can't just reassign one list object to another list object, because you'll get this (possibly surprising) result:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals
>>> large_cats.append("Tigger")
>>> animals
['lion', 'tiger', 'frumious Bandersnatch', 'Tigger']
```

In this example, you first assign the list stored in the animals variable to the variable large_cats, and then we add a new string to the large_- cats list. But, when the contents of animals are displayed you can see that the original list has also been changed.

This is a quirk of object-oriented programming, but it’s by design. When you say large_cats = animals, the large_cats and animals variables both refer to the same object.

A variable name is really just a reference to a specific location in computer memory. Instead of copying all the contents of the list object and creating a new list, large_cats = animals assigns the memory location referenced by animals to large_cats. That is, both variables now refer to the same object in memory, and any changes made to one will affect the other.

To get an independent copy of the animals list, you can use slicing notation to return a new list with the same values:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals[:]
>>> large_cats.append("leopard")
>>> large_cats
['lion', 'tiger', 'frumious Bandersnatch', 'leopard']
>>> animals
["lion", "tiger", "frumious Bandersnatch"]
```

Since no index numbers are specified in the [:] slice, every element of the list is returned from beginning to end. The large_cats list now has the same elements as animals, and in the same order, but you can .append() items to it without changing the list assigned to animals.

If you want to make a copy of a list of lists, you can do so using the [:] notation you saw earlier:

```
>>> matrix1 = [[1, 2], [3, 4]]
>>> matrix2 = matrix1[:]
>>> matrix2[0] = [5, 6]
>>> matrix2
[[5, 6], [3, 4]]
>>> matrix1
[[1, 2], [3, 4]]
```

Let’s see what happens when you change the first element of the second list in matrix2:

```
>>> matrix2[1][0] = 1
>>> matrix2
[[5, 6], [1, 4]]
>>> matrix1
[[1, 2], [1, 4]]
```

Notice that the second list in matrix1 was also altered!

This happens because a list does not really contain objects themselves, but references to those objects in memory. When you make a copy of the list using the [:] notation, a new list is returned containing the same references as the original list. In programming jargon, this method of copying a list is called a shallow copy.

Sorting Lists

Lists have a .sort() method that sorts all of the items in ascending order. By default, the list is sorted in alphabetical or numerical order, depending on the type of elements in the list:

```
>>> # Lists of strings are sorted alphabetically
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors.sort()
>>> colors
['blue', 'green', 'red', 'yellow']
>>> # Lists of numbers are sorted numerically
>>> numbers = [1, 10, 5, 3]
>>> numbers.sort()
>>> numbers
[1, 3, 5, 10]
```

Notice that .sort() sorts the list in place, so you don’t need to assign it’s result to anything.

.sort() has an option parameter called key that can be used to adjust how the list gets sorted. The key parameter accepts a function, and the list is sorted based on the return value of that function.

For example, to sort a list of strings by the length of each string, you can pass the len function to key:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors.sort(key=len)
>>> colors
['red', 'blue', 'green', 'yellow']
```

You don’t need to call the function when you pass it to key. Pass the name of the function without any parentheses. For instance, in the previous example the name len is passed to key, and not len().

The function that gets passed to key must only accept a single argument.

You can also pass user defined functions to key. In the following example, a function called get_second_element() is used to sort a list of tuples by their second elements:

```
>>> def get_second_element(item):
... return item[1]
...
>>> items = [(4, 1), (1, 2), (-9, 0)]
>>> items.sort(key=get_second_element)
>>> items
[(-9, 0), (4, 1), (1, 2)]
```

Keep in mind that any function that you pass to key must accept only a single argument.

Challenge: List of lists

Write a program that contains the following lists of lists:

```
universities = [
['California Institute of Technology', 2175, 37704],
['Harvard', 19627, 39849],
['Massachusetts Institute of Technology', 10566, 40732],
['Princeton', 7802, 37000],
['Rice', 5879, 35551],
['Stanford', 19535, 40569],
['Yale', 11701, 40500]
]
```

Define a function, enrollment_stats(), that takes, as an input, a list of lists where each individual list contains three elements: (a) the name of a university, (b) the total number of enrolled students, and (c) the annual tuition fees.

enrollment_stats() should return two lists: the first containing all of the student enrollment values and the second containing all of the tuition fees.

Next, define a mean() and a median() function. Both functions should take a single list as an argument and return the mean and median of the values in each list.

Using universities, enrollment_stats(), mean(), and median(), calculate the total number of students, the total tuition, the mean and median of the number of students, and the mean and median tuition values.

Finally, output all values, and format the output so that it looks like this:

```
******************************
Total students: 77,285
Total tuition: $ 271,905
Student mean: 11,040.71
Student median: 10,566
Tuition mean: $ 38,843.57
Tuition median: $ 39,849
******************************
```

b. Dictionaries

Dictionaries provide a way to associate values with unique keys, allowing for efficient data retrieval based on those keys. This structure is particularly useful when you need to store data that has a logical relationship between a key and a value, like a person's name and their age.

Key Features of Dictionaries:

- **Unordered**: The items in a dictionary do not maintain any specific order. As of Python 3.7, dictionaries maintain insertion order as an implementation detail, but you should not rely on this if order matters.
- **Mutable**: Dictionaries can also be modified after creation. You can add new key-value pairs or update existing ones.
- **Key-Value Pairs**: Each entry in a dictionary consists of a unique key and its associated value, making it easy to retrieve, add, or modify data.

Common Operations:

- Accessing values using keys (e.g., my_dict["name"]).
- Adding new key-value pairs (e.g., my_dict["age"] = 25).
- Removing entries with del my_dict["key"].
- Iterating over keys, values, or both using methods like keys(), values(), and items().

Concise overview on dictionary:

- **Creating a Dictionary** : Initialize a dictionary using curly braces or the dict() constructor.

```
      my_dict = {"name": "Alice", "age": 25, "city": "New York"}
     # or
      my_dict = dict(name="Alice", age=25, city="New York")
```

- **Accessing Values:** Retrieve the value associated with a specific key

```
age = my_dict["age"]  # Output: 25
```

- **Adding or Updating Key-Value Pairs**: Add a new key-value pair or update the value of an existing key.

```
my_dict["email"] = "alice@example.com"  # Adding a new key
my_dict["age"] = 26
                    # Updating an existing key
```

- **Removing Key-Value Pairs** : Remove a specific key-value pair using the del statement or pop() method.

```
del my_dict["city"]                         # Remove using del
age = my_dict.pop("age")                    # Remove using pop and return
the value
```

- **Checking for Key Existence**: Determine if a specific key exists in the dictionary.

```
if "name" in my_dict:
    print("Name exists in the dictionary.")
```

- **Iterating Over a Dictionary :** Loop through keys, values, or key-value pairs

```
for key in my_dict:  # Iterating over keys
  print(key)

for value in my_dict.values():  # Iterating over values
    print(value)

for key, value in my_dict.items():  # Iterating over key-value pairs
    print(f"{key}: {value}")
```

These operations allow you to effectively manage and manipulate dictionaries in Python, providing a robust way to store and retrieve data through key-value associations.

Self-Check Sheet -3: Apply Input and Output Methods in python

Q1: How does a for loop work in Python?

Q2. Write a Python code that uses a while loop to print numbers from 1 to 5.

Q3. What problem can occur with a while loop if it doesn't have a proper exit condition?

Q4. Write a code that uses a for loop to print only even numbers from a list.

Q5. How do you access elements in a list and a dictionary by indexing?

Q6. How do you add, update, and delete elements in a list and dictionary?

Q7. Write a while loop to find the sum of all numbers from 1 to 10.

Q8. Question: How do you find the maximum value in a list using a for loop?

Answer Sheet 3: Solve problems associated with loops

Q1. How does a for loop work in Python?
Answer:
A for loop in Python iterates over items in a sequence (such as a list, tuple, or string) and executes a block of code for each item.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Q2. Write a Python code that uses a while loop to print numbers from 1 to 5.
Answer:
A while loop executes as long as a given condition is True.

```
num = 1
while num <= 5:
    print(num)
    num += 1
```

This code will print numbers from 1 to 5.

Q3: What problem can occur with a while loop if it doesn't have a proper exit condition?
Answer:
If a while loop lacks a proper exit condition, it can lead to an infinite loop. This means the loop will continue executing indefinitely, which can crash the program or cause high memory usage.

Q4: Write a code that uses a for loop to print only even numbers from a list.
Answer:
Use the for loop with an if condition to check if a number is even**.**

```
numbers = [1, 2, 3, 4, 5, 6]
for num in numbers:
    if num % 2 == 0:
        print(num)
```

This code will print 2, 4, and 6.
Q5: How do you access elements in a list and a dictionary by indexing?
Answer:

In a list, elements can be accessed using integer indices starting from 0. In a dictionary, elements are accessed using keys.

```
my_list = [10, 20, 30]
print(my_list[1])  # Outputs: 20

my_dict = {"name": "Alice", "age": 25}
print(my_dict["name"])  # Outputs: Alice
```

Q6: How do you add, update, and delete elements in a list and dictionary?
Answer:
List :

```
my_list = [1, 2, 3]
my_list.append(4)        # Add
my_list[1] = 20          # Update
del my_list[2]           # Delete
print(my_list)           # Outputs: [1, 20, 4]
```

Dictionary:

```
my_dict = {"name": "Bob"}
my_dict["age"] = 30       # Add
my_dict["name"] = "Alice" # Update
del my_dict["age"]        # Delete
print(my_dict)            # Outputs: {"name": "Alice"}
```

Q7. Write a while loop to find the sum of all numbers from 1 to 10.
Answer:
Use a while loop with a cumulative sum variable.

```
num = 1
total = 0
while num <= 10:
    total += num
    num += 1
print("Sum:", total)
```

This code will output Sum: 55.

Q8. How do you find the maximum value in a list using a for loop?
Answer:
Use a for loop with a variable to track the maximum value.

```
numbers = [5, 3, 8, 2, 7]
max_num = numbers[0]
for num in numbers:
    if num > max_num:
        max_num = num
print("Maximum:", max_num)
```

Task Sheet 3.1: Printing Even Numbers Using Loops

Objective: Write a program that prints all even numbers from a given list using both for and while loops.

Instructions:

1. Use a for loop to iterate through a list of numbers and print only the even numbers.
2. Repeat the same task using a while loop.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using for loop
print("Even numbers using for loop:")
for num in numbers:
    if num % 2 == 0:
        print(num, end=" ")

# Using while loop
print("\nEven numbers using while loop:")
i = 0
while i < len(numbers):
    if numbers[i] % 2 == 0:
        print(numbers[i], end=" ")
    i += 1
```

Task Sheet 3.2: Calculating the Sum of List Elements

Objective: Write a program that calculates and prints the sum of elements in a list using a for loop and then using a while loop.

```python
numbers = [5, 10, 15, 20, 25]

# Using for loop
total_sum = 0
for num in numbers:
    total_sum += num
print(f"Sum using for loop: {total_sum}")

# Using while loop
total_sum = 0
i = 0
while i < len(numbers):
    total_sum += numbers[i]
    i += 1
print(f"Sum using while loop: {total_sum}")
```

Task Sheet 3.3: Dictionary Operations

Objective: Create a dictionary to store student names and their grades. Implement the following:

1. Add a new student with their grade.
2. Update the grade of an existing student.
3. Delete a student from the dictionary.
4. Print all student names and their grades.

Answer:

```python
# Initial dictionary
students = {"Alice": 85, "Bob": 92, "Charlie": 78}

# 1. Adding a new student
students["David"] = 88
print("After adding David:", students)

# 2. Updating an existing student’s grade
students["Alice"] = 90
print("After updating Alice's grade:", students)

# 3. Deleting a student
del students["Charlie"]
print("After deleting Charlie:", students)

# 4. Printing all students and their grades
print("All students and their grades:")
for name, grade in students.items():
    print(f"{name}: {grade}")
```

Job Sheet 3: Inventory Management System

You are tasked with building a basic **Inventory Management System** for a small shop. This program should allow the shop owner to **track items**, **update stock levels**, and **calculate the total value of stock** on hand. The system should:

1. Allow the user to add items to the inventory.
2. Allow the user to update stock levels for existing items.
3. Delete items that are out of stock.
4. Display a summary of all items, including item names, quantities, and price.
5. Display the total inventory value (quantity * price for each item).
6. Be menu-driven, allowing users to repeat actions until they decide to exit.

Inventory Management Criteria:

- Each item in the inventory should have a **name**, **quantity**, and **price**.
- The program should **handle invalid inputs** and prevent infinite loops.
- Use **for loops** for displaying summaries and **while loops** for the main menu.

```python
# Inventory dictionary to store items
inventory = {}

def display_menu():
    """Display the menu options."""
    print("\n--- Inventory Management Menu ---")
    print("1. Add a new item")
    print("2. Update stock level")
    print("3. Delete out-of-stock items")
    print("4. Display inventory summary")
    print("5. Display total inventory value")
    print("6. Exit")

def add_item():
    """Add a new item to the inventory."""
    item_name = input("Enter the item name: ").strip()
    if item_name in inventory:
        print(f"{item_name} already exists in the inventory. Use update option
instead.")
        return
    try:
        quantity = int(input(f"Enter the quantity for {item_name}: "))
        price = float(input(f"Enter the price for {item_name}: "))
    except ValueError:
        print("Invalid input! Quantity and price must be numbers.")
        return

    inventory[item_name] = {'quantity': quantity, 'price': price}
    print(f"{item_name} added to inventory.")

def update_stock():
    """Update stock level for an existing item."""
    item_name = input("Enter the item name to update: ").strip()
    if item_name not in inventory:
        print(f"{item_name} not found in inventory.")
        return

    try:
        new_quantity = int(input(f"Enter the new quantity for {item_name}: "))
    except ValueError:
        print("Invalid input! Quantity must be an integer.")
        return

    inventory[item_name]['quantity'] = new_quantity
    print(f"{item_name} quantity updated to {new_quantity}.")

def delete_out_of_stock():
    """Delete items with zero or negative stock."""
    out_of_stock_items = [item for item, data in inventory.items() if
data['quantity'] <= 0]
    for item in out_of_stock_items:
        del inventory[item]
        print(f"{item} removed from inventory (out of stock).")

def display_inventory_summary():
    """Display all items in the inventory with their quantities and prices."""
    if not inventory:
        print("Inventory is empty.")
        return

    print("\n--- Inventory Summary ---")
    print("{:<15} {:<10} {:<10}".format("Item", "Quantity", "Price"))
    print("-" * 35)
    for item, data in inventory.items():
        print("{:<15} {:<10} {:<10.2f}".format(item, data['quantity'],
data['price']))

def display_total_value():
    """Calculate and display the total value of the inventory."""
    total_value = sum(data['quantity'] * data['price'] for data in inventory.values())
    print(f"\nTotal Inventory Value: ${total_value:.2f}")

# Main loop to run the menu-driven inventory management system
while True:
    display_menu()
    try:
        choice = int(input("Enter your choice (1-6): "))
    except ValueError:
        print("Invalid choice! Please enter a number from 1 to 6.")
        continue

    if choice == 1:
        add_item()
    elif choice == 2:
        update_stock()
    elif choice == 3:
        delete_out_of_stock()
    elif choice == 4:
        display_inventory_summary()
    elif choice == 5:
        display_total_value()
    elif choice == 6:
        print("Exiting the Inventory Management System.")
        break
    else:
        print("Invalid choice! Please choose a valid option.")
```

Explanation of Code Components:

1. **Inventory Dictionary**: The inventory dictionary uses item names as keys and stores the quantity and price of each item as nested dictionary values.
2. **Function Definitions**:
    - display_menu(): Displays the main menu options for the user.
    - add_item(): Adds a new item to the inventory, checking for duplicate entries.
    - update_stock(): Updates the quantity of an existing item.
    - delete_out_of_stock(): Removes items with zero or negative stock.
    - display_inventory_summary(): Uses a for loop to print a summary of each item in the inventory.
    - display_total_value(): Calculates the total inventory value by summing the products of quantity and price for each item.

3. **Menu-Driven Loop**:
   - The main while loop controls the program's menu-driven interface, where users can perform actions repeatedly until they choose to exit.
4. **Error Handling**:
   - The program handles invalid user inputs (e.g., non-numeric choices and item details) to prevent infinite loops or program crashes.

Example Output:

```
--- Inventory Management Menu ---
1. Add a new item
2. Update stock level
3. Delete out-of-stock items
4. Display inventory summary
5. Display total inventory value
6. Exit
Enter your choice (1-6): 1

Enter the item name: Apples
Enter the quantity for Apples: 50
Enter the price for Apples: 0.5
Apples added to inventory.

Enter your choice (1-6): 4

--- Inventory Summary ---
Item            Quantity   Price
----------------------------------
Apples          50         0.50

Enter your choice (1-6): 5

Total Inventory Value: $25.00

Enter your choice (1-6): 6
Exiting the Inventory Management System.
```

Learning Outcome 4: Implement String Function

Assessment Criteria	1. String input methods are interpreted 2. String input methods are applied 3. Strings are manipulated 4. Built-in string functions are used
Condition and Resource	1. Actual workplace or training environment 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil and Eraser 7. Internet Facilities 8. Whiteboard and Marker 9. Imaging Device (Digital camera, scanner etc.)
Content	▪ String input methods ▪ String operation ▪ Built-in string functions
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning

Learning Experience 4: Implement String Function

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials ‘Implement String Function’
2. Read the Information sheet and complete the Self Checks & Check answer sheets on “Work with Python basic”	1. Read Information sheet 4: Implement String Function 2. Answer Self-check 4: Implement String Function 3. Check your answer with Answer key 4: Implement String Function
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	4. Job/Task Sheet and Specification Sheet Task Sheet 4.1 :Capitalize Each Word in a Sentence Task Sheet 4.2: Count Vowels and Consonants Task Sheet 4.3: Check for Palindrome Job Sheet 4: Customer Feedback Analyzer

Information Sheet 4: Implement String Function

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

4.1 Interpret string input methods
4.2 Apply string input methods
4.3 Manipulate strings
4.4 Use of Built-in string functions

4.1 String Input Methods with their application (4.1 &4.2 combined)

Strings are one of the fundamental Python data types. The term data type refers to what kind of data a value represents. Strings are used to represent text.

In Python, there are several ways to input a string from the user. The most common method is by using the built-in function input (). This function allows the user to enter a string, which is then stored as a variable for use in the program.

Example

Here's an example of how to input a string from the user in Python −

```
# Define a variable to store the input
name = input ("Please enter your name: ")
# Print the input
print("Hello, " + name + "! Good to see you.")
```

Output

The above code generates the following output for us −

```
Please enter your name: Max
Hello, Max! Good to see you.
```

In the code above, we have,

- Define a variable to store the input − name = input("Please enter your name: ")
- In this step, a variable named "name" is created to store the input from the user.
- Prompt the user to enter their name − input("Please enter your name: ")
- The "input()" function is used to display a message to the user, asking them to enter their name. The message, "Please enter your name: ", is passed as an argument to the function.

- Store the user's input in the "name" variable − name = ...
- The result of the "input()" function call is stored in the "name" variable. This means that the user's input is now stored in the "name" variable, ready to be used.
- Print the input − print("Hello, " + name + "! Good to see you.")
- In this step, the "print()" function is used to display a message to the user, using the value stored in the "name" variable. The message, "Hello, [name]! Good to see you.", is passed as an argument to the function. The value of "name" is concatenated with the rest of the string using the "+" operator.

It's crucial to remember that the output of the "input()" function will always be a string, even if the user enters a numerical value. If you need to use the input as a number, you'll need to convert it to the appropriate data type (e.g. int or float).

Example

Here's an example of how to input a number from the user −

```
# Define a variable to store the input
age = int(input("Please enter your age: "))
# Print the input
print("Wow, you are " + str(age) + " years old!")
```

Output

The above code generates the following output for us −

```
Please enter your age: 24
Wow, you are 24 years old!
```

From the above code,

- A variable named "age" is created to store the input from the user.
- The message, "Please enter your age: ", is passed as an argument to the function.
- Since the "input()" function always returns a string, we need to convert the user's input to an integer using the "int()" function. This allows us to store the user's input as a number, rather than a string.
- The result of the "int()" function call is stored in the "age" variable.
- The "print()" function is used to display a message to the user, using the value stored in the "age" variable. The message, "Wow, you are [age] years old!", is passed as an argument to the function. The value of "age" is first converted to a string using the "str()" function and then concatenated with the rest of the string using the "+" operator.

It's also possible to assign a default value to the input, in case the user doesn't provide any input. This can be done using the "or" operator and a default value −

Example

```
# Define a variable to store the input
name = input("Please enter your name (or press enter for default): ") or "Max"
# Print the input
print("Hello, " + name + "! Good to see you.")
```

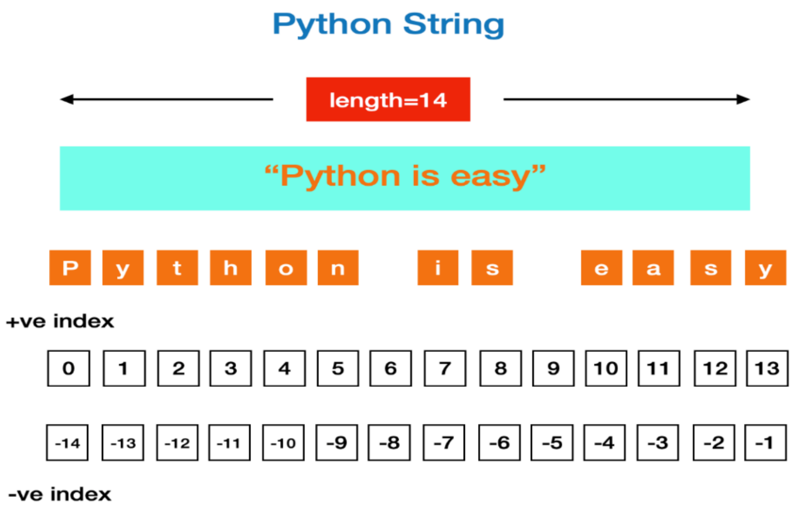
Output

The above code generates the following output for us −

```
Please enter your name (or press enter for default):
Hello, Max! Good to see you.
```

Here in the code above,

- A variable named "name" is created to store the name input by the user.
- The message, "Please enter your name (or press enter for default) − ", is passed as an argument to the function.
- The or operator is used to set a default value for the name variable. If the user presses enter without entering a name, the input() function will return an empty string. If the user's input is an empty string, the or operator will evaluate to the default value, "Max".
- The result of the input() function call, or the default value "Max" is stored in the name variable.
- A personalized greeting is printed, using the name variable. The + operator is used to concatenate the string values, creating a single string to be printed.



4.3 Manipulate String

String Literals

As you've already seen, you can create a string by surrounding some text with quotation marks:

```
string1 = 'Hello, world'
string2 = "1234"
```

Either single quotes (string1) or double quotes (string2) can be used to create a string, as long as both quotation marks are the same type.

Whenever you create a string by surrounding text with quotation marks, the string is called a string literal. The name indicates that the string is literally written out in your code. All of the strings you have seen thus far are string literals.

The quotes surrounding a string are called delimiters because they tell Python where a string begins and where it ends. When one type of quotes is used as the delimiter, the other type of quote can be used inside of the string:

```
string3 = "We're #1!"
string4 = 'I said, "Put it over by the llama."'
```

After Python reads the first delimiter, all of the characters after it are considered a part of the string until a second matching delimiter is read. This is why you can use a single quote in a string delimited by double quotes and vice versa.

If you try to use double quotes inside of a string that is delimited by double quotes, you will get an error:

```
>>> text = "She said, "What time is it?""
File "<stdin>", line 1
text = "She said, "What time is it?""
^
SyntaxError: invalid syntax
```

Python throws a SyntaxError because it thinks that the string ends after the second " and doesn't know how to interpret the rest of the line.

A common pet peeve among programmers is the use of mixed quotes as delimiters. When you work on a project, it's a good idea to use only single quotes or only double quotes to delimit every string. Keep in mind that there isn’t really a right or wrong choice! The goal is to be consistent, because consistency helps make your code easier to read and understand.

Strings can contain any valid Unicode character. For example, the string "We're #1!" contains the pound sign (#) and "1234" contains numbers. "×Pýŧħøŋ×" is also a valid Python string!

Determine the Length of a String

The number of characters contained in a string, including spaces, is called the length of the string. For example, the string "abc" has a length of 3, and the string "Don't Panic" has a length of 11.

To determine a string’s length, you use Python’s built-in len() function. To see how it works, type the following into IDLE’s interactive window:

```
>>> len("abc")
3
```

You can also use len() to get the length of a string that’s assigned to a variable:

```
>>> letters = "abc"
>>> num_letters = len(letters)
>>> num_letters
3
```

First, the string "abc" is assigned to the variable letters. Then len() is used to get the length of letters and this value is assigned to the num_letters variable. Finally, the value of num_letters, which is 3, is displayed.

Multiline Strings

The PEP 8 style guide recommends that each line of Python code contain no more than 79 characters—including spaces.

PEP 8’s 79-character line-length is recommended because, among other things, it makes it easier to read two files side by-side. However, many Python programmers believe forcing each line to be at most 79 characters sometimes makes code harder to read. In this book we will strictly follow PEP 8’s recommended line length. Just know that you will encounter lots of code in the real world with longer lines.

Whether you decide to follow PEP 8, or choose a larger number of characters for your line-length, you will sometimes need to create string literals with more characters than your chosen limit.

To deal with long strings, you can break the string up across multiple lines into a multiline string. For example, suppose you need to fit the following text into a string literal:

“This planet has—or rather had—a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movements of small green pieces of paper, which is odd because on the whole it wasn’t the small green pieces of paper that were unhappy.”

— Douglas Adams, The Hitchhiker’s Guide to the Galaxy

This paragraph contains far more than 79 characters, so any line of code containing the paragraph as a string literal violates PEP 8. So, what do you do?

There are a couple of ways to tackle this. One way is to break the string up across multiple lines and put a backslash (\) at the end of all but the last line. To be PEP 8 compliant, the total length of the line, including the backslash, must be 79 characters or less.

Here’s how you could write the paragraph as a multiline string using the backslash method:

```
paragraph = "This planet has - or rather had - a problem, which was \ this: most of the
people living on it were unhappy for pretty much \ of the time. Many solutions were
suggested for this problem, but \ most of these were largely concerned with the
movements of small \ green pieces of paper, which is odd because on the whole it wasn't
\ the small green pieces of paper that were unhappy."
```

Notice that you don’t have to close each line with a quotation mark. Normally, Python would get to the end of the first line and complain that you didn’t close the string with a matching double quote. With a backslash at the end, however, you can keep writing the same string on the next line.

When you print() a multiline string that is broken up by backslashes, the output displayed on a single line:

```
>>> long_string = "This multiline string is \
displayed on one line"
>>> print(long_string)
This multiline string is displayed on one line
```

Multiline strings can also be created using triple quotes as delimiters (""" or '''). Here is how you might write a long paragraph using this approach:

```
paragraph = """This planet has - or rather had - a problem, which was
this: most of the people living on it were unhappy for pretty much
of the time. Many solutions were suggested for this problem, but
```

```
most of these were largely concerned with the movements of small
green pieces of paper, which is odd because on the whole it wasn't
the small green pieces of paper that were unhappy."""
```

Triple-quoted strings preserve whitespace. This means that running print(paragraph) displays the string on multiple lines just like it is in the string literal, including newlines. This may or may not be what you want, so you'll need to think about the desired output before you choose how to write a multiline string.

To see how whitespace is preserved in a triple-quoted string, type the following into IDLE's interactive window:

```
>>> print("""An example of a
... string that spans across multiple lines
... that also preserves whitespace.""")
```

An example of a string that spans across multiple lines that also preserves whitespace.

Notice how the second and third lines in the output are indented exactly the same way they are in the string literal.

Concatenation, Indexing, and Slicing

Now that you know what a string is and how to declare string literals in your code, let's explore some of the things you can do with strings.

In this section, you'll learn about three basic string operations:

- Concatenation, which joins two strings together
- Indexing, which gets a single character from a string
- Slicing, which gets several characters from a string at once

String Concatenation

Two strings can be combined, or concatenated, using the + operator:

```
>>> string1 = "abra"
>>> string2 = "cadabra"
>>> magic_string = string1 + string2
>>> magic_string
'abracadabra'
```

In this example, string concatenation occurs on the third line. string1 and string2 are concatenated using + and the result is assigned to the variable magic_string. Notice that the two strings are joined without any whitespace between them.

You can use string concatenation to join two related strings, such as joining a first and last name into a full name:

```
>>> first_name = "Arthur"
>>> last_name = "Dent"
>>> full_name = first_name + " " + last_name
>>> full_name
'Arthur Dent'
```

Here string concatenation occurs twice on the same line. first_name is concatenated with " ", resulting in the string "Arthur ". Then this result is concatenated with last_name to produce the full name "Arthur Dent".

String Indexing

Each character in a string has a numbered position called an index. You can access the character at the Nth position by putting the number N in between two square brackets ([and]) immediately after the string:

```
>>> flavor = "apple pie"
>>> flavor[1]
'p'
```

flavor[1] returns the character at position 1 in "apple pie", which is p. Wait, isn't a the first character of "apple pie"?

In Python—and most other programming languages—counting always starts at zero. To get the character at the beginning of a string, you need to access the character at position 0:

```
>>> flavor[0]
'a'
```

Forgetting that counting starts with zero and trying to access the first character in a string with the index 1 results in an oпby-one error. Off-by-one errors are a common source of frustration for both beginning and experienced programmers alike!

The following figure shows the index for each character of the string "apple pie":

a	p	p	l	e		p	i	e
0	1	2	3	4	5	6	7	8

If you try to access an index beyond the end of a string, Python raises an IndexError:

```
>>> flavor[9]
Traceback (most recent call last):
File "<pyshell#4>", line 1, in <module>
flavor[9]
IndexError: string index out of range
```

The largest index in a string is always one less than the string's length. Since "apple pie" has a length of nine, the largest index allowed is 8.

Strings also support negative indices:

```
>>> flavor[-1]
'e'
```

The last character in a string has index -1, which for "apple pie" is the letter e. The second-to-last character i has index -2, and so on.

The following figure shows the negative index for each character in the string "apple pie":

a	p	p	l	e		p	i	e
-9	-8	-7	-6	-5	-4	-3	-2	-1

Just like positive indices, Python raises an IndexError if you try to access a negative index less than the index of the first character in the string:

```
>>> flavor[-10]
Traceback (most recent call last):
File "<pyshell#5>", line 1, in <module>
flavor[-10]
IndexError: string index out of range
```

Negative indices may not seem useful at first, but sometimes they are a better choice than a positive index.

For example, suppose a string input by a user is assigned to the variable user_input. If you need to get the last character of the string, how do you know what index to use?

One way to get the last character of a string is to calculate the final index using len():

```
final_index = len(user_input) - 1
last_character = user_input[final_index]
```

Getting the final character with the index -1 takes less typing and doesn't require an intermediate step to calculate the final index:

```
last_character = user_input[-1]
```

String Slicing

Suppose you need the string containing just the first three letters of the string "apple pie". You could access each character by index and concatenate them, like this:

```
>>> flavor = "apple pie"
>>> flavor[0:3]
'app'
```

If you need more than just the first few letters of a string, getting each character individually and concatenating them together is clumsy and long-winded. Fortunately, Python provides a way to do this with much less typing.

You can extract a portion of a string, called a substring, by inserting a colon between two index numbers inside of square brackets, like this:

```
>>> flavor = "apple pie"
>>> flavor[0:3]
'app'
```

flavor[0:3] returns the first three characters of the string assigned to flavor, starting with the character with index 0 and going up to, but not including, the character with index 3. The [0:3] part of flavor[0:3] is called a slice. In this case, it returns a slice of "apple pie". Yum!

String slices can be confusing because the substring returned by the slice includes the character whose index is the first number, but doesn't include the character whose index is the second number.

To remember how slicing works, you can think of a string as a sequence of square slots. The left and right boundary of each slot is numbered from zero up to the length of the string, and each slot is filled with a character in the string.

Here's what this looks like for the string "apple pie":

a	p	p	l	e		p	i	e
0	1	2	3	4	5	6	7	8

The slice [x:y] returns the substring between the boundaries x and y. So, for "apple pie", the slice [0:3] returns the string "app", and the slice [3:9] returns the string "le pie".

If you omit the first index in a slice, Python assumes you want to start at index 0:

```
>>> flavor[:5]
'apple'
```

The slice [:5] is equivalent to the slice [0:5], so flavor[:5] returns the first five characters in the string "apple pie".

Similarly, if you omit the second index in the slice, Python assumes you want to return the substring that begins with the character whose index is the first number in the slice and ends with the last character in the string:

```
>>> flavor[5:]
' pie'
```

For "apple pie", the slice [5:] is equivalent to the slice [5:9]. Since the character with index 5 is a space, flavor[5:9] returns the substring that starts with the space and ends with the last letter, which is " pie".

If you omit both the first and second numbers in a slice, you get a string that starts with the character with index 0 and ends with the last character. In other words, omitting both numbers in a slice returns the entire string:

```
>>> flavor[:]
'apple pie'
```

It’s important to note that, unlike string indexing, Python won’t raise an IndexError when you try to slice between boundaries before or after the beginning and ending boundaries of a string:

```
>>> flavor[:14]
'apple pie'
>>> flavor[13:15]
''
```

In this example, the first line gets the slice from the beginning of the string up to but not including the fourteenth character. The string assigned to flavor has length nine, so you might expect Python to throw an error. Instead, any non-existent indices are ignored and the entire string "apple pie" is returned.

The second shows what happens when you try to get a slice where the entire range is out of bounds. flavor[13:15] attempts to get the thirteenth and fourteenth characters, which don’t exist. Instead of raising an error, the empty string "" is returned.

You can use negative numbers in slices. The rules for slices with negative numbers are exactly the same as slices with positive numbers. It helps to visualize the string as slots with the boundaries labeled by negative numbers:

Just like before, the slice [x:y] returns the substring between the boundaries x and y. For instance, the slice [-9:-6] returns the first three letters of the string "apple pie":

```
>>> flavor[-9:-6]
'app'
```

Notice, however, that the right-most boundary does not have a negative index. The logical choice for that boundary would seem to be the number 0, but that doesn’t work:

```
>>> flavor[-9:0]
''
```

Instead of returning the entire string, [-9:0] returns the empty string "". This is because the second number in a slice must correspond to a boundary that comes after the boundary corresponding to the first number, but both -9 and 0 correspond to the left-most boundary in the figure.

If you need to include the final character of a string in your slice, you can omit the second number:

```
>>> flavor[-9:]
'apple pie'
```

Strings Are Immutable

To wrap this section up, let's discuss an important property of string objects. Strings are immutable, which means that you can't change them once you've created them. For instance, see what happens when you try to assign a new letter to one particular character of a string:

```
>>> word = "goal"
>>> word[0] = "f"
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>
word[0] = "f"
TypeError: 'str' object does not support item assignment
```

Python throws a TypeError and tells you that str objects don't support item assignment.

Note The term str is Python's internal name for the string data type. If you want to alter a string, you must create an entirely new string. To change the string "goal" to the string "foal", you can use a string slice to concatenate the letter "f" with everything but the first letter of the word

```
"goal":
>>> word = "goal"
>>> word = "f" + word[1:]
>>> word
'foal'
```

First assign the string "goal" to the variable word. Then concatenate the slice word[1:], which is the string "oal", with the letter "f" to get the string "foal". If you're getting a different result here, make sure you're including the : colon character as part of the string slice.

4.4 Built In String Function

Strings come bundled with special functions called string methods that can be used to work with and manipulate strings. There are numerous string methods available, but we'll focus on some of the most commonly used ones.

In this section, you will learn how to:

- Convert a string to upper or lower case
- Remove whitespace from string
- Determine if a string begins and ends with certain characters

Let's go!

Converting String Case

To convert a string to all lower case letters, you use the string's .lower() method. This is done by tacking .lower() on to the end of the string itself:

```
>>> "Jean-luc Picard".lower()
'jean-luc picard'
```

The dot (.) tells Python that what follows is the name of a method— the lower() method in this case

We will refer to the names of string methods with a dot at the beginning of them. So, for example, the .lower() method is written with a dot, instead of lower(). The reason we do this is to make it easy to spot functions that are string methods, as opposed to built-in functions like print() and type().

String methods don't just work on string literals. You can also use the .lower() method on a string assigned to a variable:

```
>>> name = "Jean-luc Picard"
>>> name.lower()
'jean-luc picard'
```

The opposite of the .lower() method is the .upper() method, which converts every character in a string to upper case:

```
>>> loud_voice = "Can you hear me yet?"
>>> loud_voice.upper()
'CAN YOU HEAR ME YET?'
```

Compare the .upper() and .lower() string methods to the general purpose len() function you saw in the last section. Aside from the different results of these functions, the important distinction here is how they are used.

The len() function is a stand-alone function. If you want to determine the length of the loud_voice string, you call the len() function directly, like this:

```
>>> len(loud_voice)
20
```

On the other hand, .upper() and .lower() must be used in conjunction with a string. They do not exist independently.

Removing Whitespace From a String

Whitespace is any character that is printed as blank space. This includes things like spaces and line feeds, which are special characters that move output to a new line.

Sometimes you need to remove whitespace from the beginning or end of a string. This is especially useful when working with strings that come from user input, where extra whitespace characters may have been introduced by accident.

There are three string methods that you can use to remove whitespace from a string:

- .rstrip()
- .lstrip()
- .strip()

.rstrip() removes whitespace from the right side of a string:

```
>>> name = "Jean-luc Picard "
>>> name
'Jean-luc Picard '
>>> name.rstrip()
'Jean-luc Picard'
```

In this example, the string **"Jean-luc Picard "** has five trailing spaces. Python doesn’t remove any trailing spaces in a string automatically when the string is assigned to a variable. The .rstrip() method removes trailing spaces from the right-hand side of the string and returns a new string **"Jean-luc Picard"**, which no longer has the spaces at the end.

The .lstrip() method works just like .rstrip(), except that it removes whitespace from the left-hand side of the string:

```
>>> name = " Jean-luc Picard"
>>> name
' Jean-luc Picard'
>>> name.lstrip()
'Jean-luc Picard'
```

To remove whitespace from both the left and the right sides of the string at the same time, use the .strip() method:

```
>>> name = " Jean-luc Picard "
>>> name
' Jean-luc Picard '
>>> name.strip()
'Jean-luc Picard'
```

None of the .rstrip(), .lstrip(), and .strip() methods remove whitespace from the middle of the string. In each of the previous examples the space between “Jean-luc” and “Picard” is always preserved.

Determine if a String Starts or Ends With a Particular String

When you work with text, sometimes you need to determine if a given string starts with or ends with certain characters. You can use two string methods to solve this problem: .startswith() and .endswith().

Let’s look at an example. Consider the string "Enterprise". Here’s how you use .startswith() to determine if the string starts with the letters e and n:

```
>>> starship = "Enterprise"
>>> starship.startswith("en")
False
```

You must tell .startswith() what characters to search for by providing a string containing those characters. So, to determine if "Enterprise" starts with the letters e and n, you call .startswith("en"). This returns False. Why do you think that is?

If you guessed that .startswith("en") returns False because "Enterprise" starts with a capital E, you’re absolutely right! The .startswith() method is case-sensitive. To get .startswith() to return True, you need to provide it with the string "En":

```
>>> starship.startswith("En")
True
```

The .endswith() method is used to determine if a string ends with certain characters:

```
>>> starship.endswith("rise")
True
```

Just like .startswith(), the .endswith() method is case-sensitive:

```
>>> starship.endswith("risE")
False
```

The True and False values are not strings. They are a special kind of data type called a Boolean value.

String Methods and Immutability

Recall from the previous section that strings are immutable—they can’t be changed once they have been created. Most string methods that alter a string, like .upper() and .lower(), actually return copies of the original string with the appropriate modifications.

If you aren’t careful, this can introduce subtle bugs into your program. Try this out in IDLE’s interactive window:

```
>>> name = "Picard"
>>> name.upper()
'PICARD'
>>> name
'Picard'
```

When you call name.upper(), nothing about name actually changes. If you need to keep the result, you need to assign it to a variable:

```
>>> name = "Picard"
>>> name = name.upper()
>>> name
'PICARD'
```

name.upper() returns a new string "PICARD", which is re-assigned to the name variable. This overrides the original string "Picard" assigned to "name".

Strings and Arithmetic Operators

You’ve seen that string objects can hold many types of characters, including numbers. However, don’t confuse numerals in a string with actual numbers. For instance, try this bit of code out in IDLE’s interactive window:

```
>>> num = "2"
>>> num + num
'22'
```

The + operator concatenates two string together. So, the result of "2" + "2" is "22", not "4".

Strings can be “multiplied” by a number as long as that number is an integer, or whole number. Type the following into the interactive window:

```
>>> num = "12"
>>> num * 3
'121212'
```

num * 3 concatenates the string "12" with itself three times and returns the string "121212". To compare this operation to arithmetic with numbers, notice that "12" * 3 = "12" + "12" + "12". In other words, multiplying a string by an integer n concatenates that string with itself n times.

The number on the right-hand side of the expression num * 3 can be moved to the left, and the result is unchanged:

```
>>> 3 * num
'121212'
```

What do you think happens if you use the * operator between two strings? Type "12" * "3" in the interactive window and press Enter:

```
>>> "12" * "3"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Python raises a TypeError and tells you that you can’t multiply a sequence by a non-integer. When the * operator is used with a string on either the left or the right side, it always expects an integer on the other side.

Note

A sequence is any Python object that supports accessing elements by index. Strings are sequences.

What do you think happens when you try to add a string and a number?

```
>>> "3" + 3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Again, Python throws a TypeError because the + operator expects both things on either side of it to be of the same type. If any one of the objects on either side of + is a string, Python tries to perform string concatenation. Addition will only be performed if both objects are numbers. So, to add "3" + 3 and get 6, you must first convert the string "3" to a number.

Converting Strings to Numbers

The TypeError errors you saw in the previous section highlight a common problem encountered when working with user input: type mismatches when trying to use the input in an operation that requires a number and not a string

Let’s look at an example. Save and run the following script.

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

When you enter a number, such as 2, you expect the output to be 4, but in this case, you get 22. Remember, input() always returns a string, so if you input 2, then num is assigned the string "2", not the integer 2. Therefore, the expression num * 2 returns the string "2" concatenated with itself, which is "22".

To perform arithmetic on numbers that are contained in a string, you must first convert them from a string type to a number type. There are two ways to do this: int() and float().

int() stands for integer and converts objects into whole numbers, while float() stands for coating-point number and converts objects into numbers with decimal points. Here's what using them looks like in the interactive window:

```
>>> int("12")
12
>>> float("12")
12.0
```

Notice how float() adds a decimal point to the number. Floating-point numbers always have at least one decimal place of precision. For this reason, you can't change a string that looks like a floating-point number into an integer because you would lose everything after the decimal point:

```
>>> int("12.0")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.0'
```

Even though the extra 0 after the decimal place doesn't add any value to the number, Python won't change 12.0 into 12 because it would result in the loss of precision.

Let's revisit the script from the beginning of this section and see how to fix it. Here's the script again:

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

The issue lies in the line doubled_num = num * 2 because num references a string and 2 is an integer. You can fix the problem by wrapping num with either int() or float(). Since the prompts asks the user to input a number, and not specifically an integer, let's convert num to a floating-point number:

```
num = input("Enter a number to be doubled: ")
doubled_num = float(num) * 2
print(doubled_num)
```

Now when you run this script and input 2, you get 4.0 as expected. Try it out!

Converting Numbers to Strings

Sometimes you need to convert a number to a string. You might do this, for example, if you need to build a string from some pre-existing variables that are assigned to numeric values.

As you've already seen, the following produces a TypeError:

```
>>> num_pancakes = 10
>>> "I am going to eat " + num_pancakes + " pancakes."
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
TypeError: can only concatenate str (not "int") to str
```

Since num_pancakes is a number, Python can’t concatenate it with the string "I'm going to eat". To build the string, you need to convert num_pancakes to a string using str():

```
>>> num_pancakes = 10
>>> "I am going to eat " + str(num_pancakes) + " pancakes."
'I am going to eat 10 pancakes.'
```

You can also call str() on a number literal:

```
>>> "I am going to eat " + str(10) + " pancakes."
'I am going to eat 10 pancakes.'
```

str() can even handle arithmetic expressions:

```
>>> total_pancakes = 10
>>> pancakes_eaten = 5
>>> "Only " + str(total_pancakes - pancakes_eaten) + " pancakes left."
'Only 5 pancakes left.'
```

You’re not limited to numbers when using str(). You can pass it all sorts of objects to get their string representations:

```
>>> str(print)
'<built-in function print>'
>>> str(int)
"<class 'int'>"
>>> str(float)
"<class 'float'>"
```

These examples may not seem very useful, but they illustrate how flexible str() is.

In the next section, you’ll learn how to format strings neatly to display values in a nice, readable manner. Before you move on, though, check your understanding with the following review exercises.

Self-Check Sheet 4: Implement String Function

Q1: How do you take a string input from the user in Python?

Q2 How can you check if a substring exists within a string in Python?

Q3: Explain how to concatenate two strings in Python.

Q4: How do you convert a string to uppercase and lowercase in Python?

Q5: Write a Python code to count the number of characters in a string (including spaces).

Q6: How do you split a string into a list of words?

Q7: How can you replace a substring within a string?

Q8: What does the .strip() method do in Python?

Q9: How can you check if a string starts or ends with a specific substring?

Answer Sheet 4: Implement String Function

Q1: How do you take a string input from the user in Python?
Answer:
Use the input() function to take a string input from the user.

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

Q2: How can you check if a substring exists within a string in Python?
Answer:
Use the in keyword to check if a substring exists within a string.

```
text = "Welcome to Python programming"
if "Python" in text:
    print("The substring exists in the text.")
else:
    print("The substring does not exist in the text.")
```

Q3: Explain how to concatenate two strings in Python.
Answer:
Use the + operator to concatenate two strings.

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)  # Outputs: John Doe
```

Q4: How do you convert a string to uppercase and lowercase in Python?
Answer:
Use the .upper() and .lower() methods to convert a string to uppercase and lowercase, respectively.

```
text = "Hello World"
print(text.upper())  # Outputs: HELLO WORLD
print(text.lower())  # Outputs: hello world
```

Q5: Write a Python code to count the number of characters in a string (including spaces).

Answer:
Use the len() function to get the length of the string.

```
text = "Hello World"
print(len(text))  # Outputs: 11
```

Q6: How do you split a string into a list of words?
Answer:
Use the. split() method to break a string into a list of words.

```
sentence = "Python is fun"
words = sentence.split()
print(words) # Outputs: ['Python', 'is', 'fun']
```

Q7: How can you replace a substring within a string?
Answer:
Use the. replace(old, new) method to replace occurrences of a substring with another substring.

```
text = "I like cats"
new_text = text.replace("cats", "dogs")
print(new_text)  # Outputs: I like dogs
```

Q8: What does the .strip() method do in Python?
Answer:
The .strip() method removes any leading and trailing whitespace from a string.

```
text = "Hello World"
print(text.strip())  # Outputs: Hello World
```

Q9: How can you check if a string starts or ends with a specific substring?
Answer:
Use the .startswith() and .endswith() methods to check if a string starts or ends with a specified substring.

```
text = "Python programming"
print(text.startswith("Python"))  # Outputs: True
print(text.endswith("programming"))  # Outputs: True
```

Task Sheet 4.1 :Capitalize Each Word in a Sentence

Objective: Write a program that:

1. Takes a sentence as input from the user.
2. Capitalizes the first letter of each word.
3. Prints the modified sentence.

Answer:

```
# Taking input
sentence = input("Enter a sentence: ")

# Using built-in string function to capitalize each word
capitalized_sentence = sentence.title()

# Displaying the result
print("Capitalized Sentence:", capitalized_sentence)
```

Task Sheet 4.2: Count Vowels and Consonants

Objective: Write a program that:

1. Takes a string input from the user.
2. Counts the number of vowels and consonants in the string.
3. Prints the counts.

```python
# Taking input
text = input("Enter a string: ")

# Initializing counters
vowel_count = 0
consonant_count = 0

# Defining vowels
vowels = "aeiouAEIOU"

# Counting vowels and consonants
for char in text:
    if char.isalpha():  # Check if the character is a letter
        if char in vowels:
            vowel_count += 1
        else:
            consonant_count += 1

# Displaying the results
print("Vowels:", vowel_count)
print("Consonants:", consonant_count)
```

Task Sheet 4.3: Check for Palindrome

Objective: Write a program that:

1. Takes a string input from the user.
2. Checks if the string is a palindrome (reads the same forwards and backwards).
3. Prints whether the string is a palindrome or not.

```python
# Taking input
text = input("Enter a string: ")

# Removing any spaces and converting to lowercase for accurate checking
processed_text = text.replace(" ", "").lower()

# Checking if it’s a palindrome
if processed_text == processed_text[::-1]:
    print("The string is a palindrome.")
else:
   print("The string is not a palindrome.")
```

Job Sheet 4: Customer Feedback Analyzer

You are tasked with building a **Customer Feedback Analyzer** for a small business. This program will allow the business to analyze customer feedback by **counting keywords**, **extracting relevant phrases**, and **summarizing feedback sentiment**.

Requirements:

1. **Gather Feedback**:
   - The program should accept a series of feedback entries from the user. Each entry represents feedback from a single customer.
   - The user should be able to stop entering feedback by typing “END”.
2. **Keyword Analysis**:
   - After gathering feedback, the program should count the occurrences of key words and phrases (e.g., "great," "satisfied," "disappointed," etc.).
   - Ask the user for a list of keywords to search for, separated by commas.
3. **Phrase Extraction**:
   - Extract and display any phrase that starts with "I feel" or "I think" in each feedback entry to capture customer sentiments.
4. **Feedback Summary**:
   - Display the total number of feedback entries.
   - Display the keyword counts for each keyword.
   - Display the extracted phrases.

Example Run :

```
Enter customer feedback. Type "END" to finish:
> The product quality is great and I am very satisfied.
> I feel the customer service needs improvement.
> I think the delivery was fast and reliable.
> Very disappointed with the packaging.
> END

Enter  keywords  to  analyze,  separated  by  commas  (e.g.,  great,  satisfied,
disappointed): great, satisfied, disappointed

Feedback Summary:
-----------------
Total feedback entries: 4

Keyword Counts:
```

```
- great: 1
- satisfied: 1
- disappointed: 1

Extracted Phrases:
- I feel the customer service needs improvement.
- I think the delivery was fast and reliable.
```

Source Code :

```python
# Initialize an empty list to store feedback entries
feedback_list = []

print("Enter customer feedback. Type 'END' to finish:")

# Gather feedback from the user until "END" is typed
while True:
    feedback = input("> ")
    if feedback.strip().upper() == "END":
        break
    feedback_list.append(feedback.strip())

# Ensure at least one feedback entry was provided
if not feedback_list:
    print("No feedback provided.")
else:
    # Ask the user for keywords to analyze
      keywords = input("Enter keywords to analyze, separated by commas: ").strip().lower().split(',')

    # Clean up keywords by removing whitespace
    keywords = [keyword.strip() for keyword in keywords if keyword.strip()]

    # Initialize dictionary to store keyword counts
    keyword_counts = {keyword: 0 for keyword in keywords}

    # Initialize a list to store extracted phrases
    extracted_phrases = []

    # Analyze each feedback entry
    for feedback in feedback_list:
        # Convert feedback to lowercase for keyword counting
        feedback_lower = feedback.lower()

        # Count occurrences of each keyword in the feedback
        for keyword in keywords:
            keyword_counts[keyword] += feedback_lower.count(keyword)

        # Extract phrases that start with "I feel" or "I think"
        if "i feel" in feedback_lower:
            start_idx = feedback_lower.find("i feel")
            extracted_phrases.append(feedback[start_idx:])

        elif "i think" in feedback_lower:
            start_idx = feedback_lower.find("i think")
            extracted_phrases.append(feedback[start_idx:])

    # Display Feedback Summary
    print("\nFeedback Summary:")
    print("-----------------")
    print(f"Total feedback entries: {len(feedback_list)}")

    # Display keyword counts
    print("\nKeyword Counts:")
    for keyword, count in keyword_counts.items():
        print(f"- {keyword}: {count}")

    # Display extracted phrases
    print("\nExtracted Phrases:")
    if extracted_phrases:
        for phrase in extracted_phrases:
            print(f"- {phrase}")
    else:
        print("No phrases starting with 'I feel' or 'I think' found.")
```

Learning Outcome 5: Implement Basic I/O Functions

Assessment Criteria	1. Opening and closing files are exercised 2. Modes of accessing files are exercised 3. Create, update and delete of a file is exercised
Condition and Resource	1. Actual workplace or training environment 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil and Eraser 7. Internet Facilities 8. Whiteboard and Marker 9. Imaging Device (Digital camera, scanner etc.)
Content	▪ Opening and closing files ▪ Modes of accessing files ▪ Create, update and delete of a file
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning

Learning Experience 5: Implement Basic I/O Functions

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
4. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Implement Basic I/O Functions'
5. Read the Information sheet and complete the Self Checks & Check answer sheets on "Work with Python basic"	2. Read Information sheet 5: Implement Basic I/O Functions 3. Answer Self-check 5: Implement Basic I/O Functions 4. Check your answer with Answer key 5: Implement Basic I/O Functions
6. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet 5.1: Create and Write to a File Task Sheet 5.2: Read Content from a File Task Sheet 5.3: Append Text to an Existing File Job Sheet 5: Personal Task Manager

Information Sheet 5: Implement Basic I/O Functions

Learning Objective:

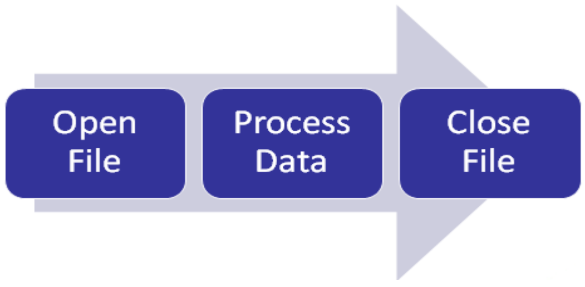
After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

5.1 Opening and closing files
5.2 Modes of accessing files
5.3 Create, update and delete of a file

5.1 Opening and Closing Files in Python

Files and the File System

You have likely been working with computer files for a long time. Even so, there are some things that programmers need to know about files that the general user does not.



In this section, you'll learn the concepts necessary to get started working with files in Python

If you are familiar with concepts like the file system and file paths, may wish to read the Working With File Paths in Python and File Metadata sections before skipping to the next section.

Let's start by exploring what a file is and how computers interact with them

The Anatomy of a File

There are a multitude of types of files out there: text files, image files, audio files, and PDF files, just to name a few. Ultimately, though, a file is just a sequence of bytes called the contents of the file.

Each byte in a file can be thought of as an integer with a value between 0 and 255, including both endpoints. The bytes are the values that are stored on a physical storage device when a file is saved.

When you access a file on a computer, the contents of the file are read from the disk in the correct sequence of bytes. The important thing to know here is that there is nothing intrinsic to the file itself that dictates how to interpret the contents.

As a programmer, it’s your job to properly interpret the contents when you open a file. This might sound difficult, but Python does a lot of the hard work for you.

For example, when you open a text file, Python can convert the numerical bytes of the file into text characters for you. You do not need to know the specifics of how this conversion happens. There are tools in the standard library for working with all sorts of file types, including images and audio files.

In order to access a file from a storage device, a whole host of things need to happen. You need to know on which device the file is stored, how to interact with that device, and where exactly on the device the file is located.

This monumental task is managed by a file system. Python interacts with the file system on your computer in order to read, write, and manipulate files.

The File System

The file system on a computer does two things:

- It provides an abstract representation of the files stored on your computer and devices connected to it.
- It interfaces with devices to control storage and retrieval file data.
- Python interacts with the file system on your computer, so you can only do in Python whatever your file system allows.

Different operating systems use different file systems. This is very important to keep in mind when writing code that will be run on different operating systems.

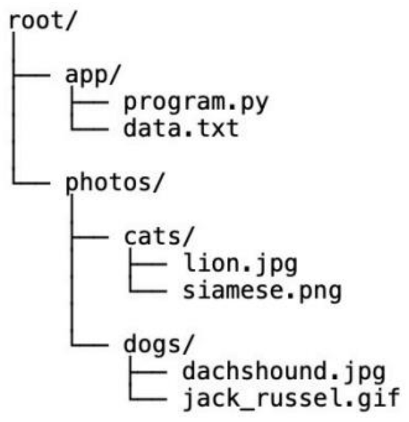
The file system itself manages communication between the computer and the physical storage device, so the only part of the file system you need to understand as a programmer is how it represents files.

The File System Hierarchy

File systems organize files in a hierarchy of directories, which are also known as folders. At the top of the hierarchy is a directory called the root directory. All other files and directories in the file system are contained in the root directory.

Each file in the directory has a ple name that must be unique from any other file in the same directory. Directories can also contain other directories, called subdirectories or subfolders.

The following directory tree visualizes the hierarchy of files and directories in an example file system:



```
root/
│
├── app/
│   ├── program.py
│   └── data.txt
│
└── photos/
    │
    ├── cats/
    │   ├── lion.jpg
    │   └── siamese.png
    │
    └── dogs/
        ├── dachshound.jpg
        └── jack_russel.gif
```

In this file system, the root folder is called root/. It has two subdirectories: app/ and photos/. The app/ subdirectory contains a program.py file and a data.txt file. The photos/ directory also has two subdirectories, cats/ and dogs/, that both contain two image files.

File Paths

To locate a file in a file system, you can list the directories in order, starting with the root directory, followed by the name of the file. A string with the file location represented in this manner is called a ple path.

For example, the file path for the jack_russel.gif file in the above file system is root/photos/dogs/jack_russel.gif.

How you write file paths depends on your operating system. Here are three examples of file paths on Windows, macOS, and Linux:

- Windows: C:\Users\David\Documents\hello.txt
- macOS: /Users/David/Documents/hello.txt
- Ubuntu Linux: /home/David/Documents/hello.txt

All three of these file paths locate a text file named hello.txt that is stored in the Documents subfolder of the user directory for a user named David. As you can see, there are some pretty big differences between file paths from one operating system to another.

On macOS and Ubuntu Linux, the operating system uses a virtual ple system that organizes all files and directories for all devices on the system under a single root directory, usually represented by a forward slash symbol (/). Files and folders from external storage devices are usually located in a subdirectory called media/.

In Windows, there is no universal root directory. Each device has a separate file system with a unique root directory that is named with a drive letter followed by a colon (:) and a back slash symbol (\). Typically, the hard drive where the operating system is installed is assigned the letter C, so the root directory of the file system for that drive is C:.

The other major difference between Windows, macOS, and Ubuntu files paths is that directories in a Windows file path are separated by back slashes (\), whereas directories in macOS and Ubuntu file paths are separated by forward slashes (/).

When you write programs that need to run on multiple operating systems, it is critical that you handle the differences in file paths appropriately. In versions of Python greater than 3.4, the standard library contains a module called pathlib helps take the pain out of handling file paths across operating systems.

Read on to learn how to use pathlib to work with file paths in Python.

Reading and Writing Files

Files are abundant in the modern world. They are the medium through with data is digitally stored and transferred. Chances are, you've opened dozens, if not hundreds, of files just today.

In this section, you'll learn how to read and write files with Python.

What Is a File?

A ple is a sequence of bytes and a byte is a number between 0 and 255. That is, a file is a sequence of integer values.

The bytes in a file must be decoded into something meaningful in order to understand the contents of the file.

Python has standard library modules for working with text, CSV, and audio files. There are a number of third-party packages available for working with other file types.

Understanding Text Files

Text files are files that contain only text. They are perhaps the easiest files to work with. There are two issues, though, that can be frustrating when working with text files:

- Character encoding
- Line endings

Before jumping into reading and writing text files, let's look at what these issues are so that you know how to deal with them effectively.

Character Encoding

Text files are stored on disk as a sequence of bytes. Each byte, or group of bytes in some cases, represents a different character in the file.

When text files are written, characters typed on the keyboard are converted to bytes in a process called encoding. When a text file is read, the bytes are decoded back into text.

The integer a character is associated to is determined by the file's character encoding. There are many character encodings. Four of the most widely used character encodings are:

- ASCII
- UTF-8
- UTF-16
- UTF-32

Some character encodings, such as ASCII and UTF-8, encode characters the same way. For example, numbers and English letters are encoded the same way in both ASCII and UTF-8.

The difference between ASCII and UTF-8 is that UTF-8 can encode more characters than ASCII. ASCII can't encode characters like ñ or ü, but UTF-8 can. This means you can decode ASCII encoded text with UTF-8, but you can't always decode UTF-8 encoded text with ASCII.

Serious problems may occur when different encodings are used to encode and decode text. For instance, text encoded as UTF-8 that is decoded with UTF16 may be interpreted as an entirely different language than originally intended! For a thorough introduction to character encodings, check out Real Python's Unicode & Character Encodings in Python: A Painless Guide.

Knowing what encoding a file uses is important, but it isn't always obvious. On modern Windows computers, txt files are usually encoded with UTF-16 or UTF-8. On macOS and Ubuntu Linux, the default character encoding is usually UTF-8.

For the remainder of this section, we'll assume that the character encoding of all text files that we work with is UTF-8. If you encounter problems, you may need to alter the examples to use a different encoding.

Line Endings

Each line in a text file ends with one or two characters that indicate the line has ended. These characters aren't usually displayed in a text editor, but they exist as bytes in the file data.

The two characters used to represent line endings are the carriage return and line feed characters. In Python strings, these characters are represented by the escape sequence \r and \n, respectively.

On Windows, line endings are represented by default with both a carriage return and a line feed. On macOS and most Linux distributions, line endings are represented with just a single line feed character.

When you read a Windows file on macOS or Linux you will sometimes see extra blank lines between lines of text. This is because the carriage return also represents a line ending on macOS and Linux.

For example, suppose the following text file was created in Windows:

```
Pug\r\n
Jack Russell Terrier\r\n
English Springer Spaniel\r\n
German Shepherd\r\n
```

On macOS or Ubuntu, this file is interpreted with double spacing between lines:

```
Pug\r
\n
Jack Russell Terrier\r
\n
English Springer Spaniel\r
\n
German Shepherd\r
\n
```

In practice, the differences between line endings on different operating systems is rarely problematic. Python can handle line ending conversions for you automatically, so you don’t have to worry about it too often.

Python File Objects

Files are represented in Python with ple objects, which are instances of classes designed to work with different types of files.

Python has a couple of different types of file objects:

- Text file objects are used for interacting with text files
- Binary file objects are used for working directly with the bytes contained in files

Text file objects handle encoding and decoding bytes for you. All you need to do is specify which character encoding to use. On the other hand, binary file objects do not perform any kind of encoding or decoding.

There are two ways to create a file object in Python:

- The Path.open() method
- The open() built-in function

Let’s look at each of these

The Path.open() Method

To use the Path.open() method, you first need a Path object. In IDLE’s interactive window, execute the following:

```
>>> from pathlib import Path
>>> path = Path.home() / "hello.txt"
>>> path.touch()
>>> file = path.open(mode="r", encoding="utf-8")
```

First, a Path object for the hello.txt file is created and assigned to the path variable. Then path.touch() creates the file in your home directory. Finally, .open() returns a new file object representing the hello.txt file and assigns it to the file variable.

Two keyword parameters used to open the file:

- The mode parameter determines in which mode the file should be opened. The "r" argument opens the file in read mode.
- The encoding parameter determines the character encoding used to decode the file. The argument "utf-8" represents the UTF-8 character encoding

You can inspect the file variable to see that it is assigned to a text file object:

```
>>> file
<_io.TextIOWrapper name='C:\Users\David\hello.txt' mode='r'
encoding='utf-8'>
```

Text file objects are instances of the TextIOWrapper class. You will never need to instantiate this class directly, since you can create them with the Path.open() method.

5.2 Mode of Accessing Files

There are a number of different modes you can use to open a file. These are described in the following table:

Mode	Description
"r"	Creates a text file object for reading and raises an error if the file can’t be opened.
w"	Creates a text file object for writing and overwrites all existing data in the file.

"a"	Creates a text file object for appending data to the end of a file.
"rb"	Creates a binary file object for reading and raises an error if the file can’t be opened.
"wb"	Creates a binary file object for writing and overwrites all existing data in the file.
"ab"	Creates a binary file object for appending data to the end of the file

The strings for some of the most commonly used character encodings can be found in the table below:

String	Character Encoding
"ascii"	ASCII
"utf-8"	UTF-8
"utf-16”	UTF-16
"utf-32"	UTF-32

When you create a file object with .open(), Python maintains a link to the file resource until you either explicitly tell Python to close the file, or the program ends.

You should always explicitly tell Python to close a file. Forgetting to close opened files like littering. When your program stops running, it shouldn’t leave unnecessary waste laying around the system.

To close a file, use the file object’s .close() method:

```
>>> file.close()
```

Using Path.open() is the preferred way to open a file when you have an existing Path object, but there is also a built-in function called open() that you can use to open a file.

5.3 Create Update and Delete File

The open() Built-in

The built-in open() function works almost exactly like the Path.open() method, except that it's first parameter is a string containing the path the file you want to open.

First, create a new variable called file_path and assign to it a string containing the path to the hello.txt file you created above:

```
>>> file_path = "C:/Users/David/hello.txt"
```

Note that you'll need to change the path to match the path of the file on your own computer.

Next, create a new file object using the open() built-in and assign it to the variable file:

```
>>> file = open(file_path, mode="r", encoding="utf-8")
```

The first parameter of open() must be a path string. The mode and encoding parameters are the same as the parameters for the Path.open() method. In this example, mode is set to "r" for read mode, and encoding is set to "utf-8".

Just like the file object returned by Path.open(), the file object returned by open() is a TextIOWrapper instance:

```
>>> file
<_io.TextIOWrapper name='C:/Users/David/hello.txt' mode='r' encoding='utf-8'>
```

To close the file, use the file object's .close() method:

```
>>> file.close()
```

For the most part, you'll use the Path.open() method to open a file from an existing pathlib.Path object. However, if you don't need all of the functionality of the pathlib module, then open() is a great way to quickly create a file object.

The with Statement

When you open a file, your program is accessing data external to the program itself. The operating system must manage the connection between your program and physical file itself. When you call a file object's .close() method, the operating system knows to close the connection.

If your program crashes between the time that a file is opened and when it is closed, the system resources maintained by the connection may continue to live on until the operating system realizes that it's no longer needed.

To ensure that file system resources are cleaned up even if a program crashes, you can open a file in a with statement. The pattern for using the with statement looks like this:

```
with path.open(mode="r", encoding=-"utf-8") as file:
# Do something with file
```

The with statement has two parts: a header and a body. The header always starts with the with keyword and ends with a colon (:). The return value of path.open() is assigned to the variable name after the as keyword.

After the with statement header is an indented block of code. When code execution leaves the indented block, the file object assigned to file is closed automatically, even if any exception is raised during execution of code inside of the block.

with statements also work with the open() built-in:

```
with open(file_path, mode="r", encoding="utf-8") as file:
# Do something with file
```

There really is no reason not to open files in a with statement. It is considered the Pythonic way for working with files. For the rest of this book, we will use this pattern whenever opening a file.

Reading Data From a File

Using a text editor, open the hello.txt file in your home directory that you previously created and type the text Hello World into it. Then save the file.

In IDLE’s interactive window, type the following:

```
>>> path = Path.home() / "hello.txt"
>>> with path.open(mode="r", encoding="utf-8") as file:
... text = file.read()
...
>>>
```

The file object created by path.open() is assigned to the file variable. Inside of the with block, the file object’s .read() method reads the text from the file and assigns the result to the variable text.

The value returns by .read() is a string object with the value "Hello

World":

```
>>> type(text)
<class 'str'>
>>> text
'Hello World'
```

The .read() method reads all of the text in the file and returns it as a string value.

If there are multiple lines of text in the file, each line in the string is separated with a newline character \n. In a text editor, open the hello.txt file again and put the text "Hello again" on the second line. Then save the file.

Back in IDLE’s interactive window, read the text from the file again:

```
>>> with path.open(mode="r", encoding="utf-8") as file:
... text = file.read()
...
>>> text
'Hello World\nHello again'
```

The text from each line has a \n character in between.

Instead of reading the entire file at once, you can process each line of

the file one at a time:

```
>>> with path.open(mode="r", encoding="utf-8") as file:
... for line in file.readlines():
... print(line)
...
Hello World
Hello again
```

The .readlines() method returns an iterable of lines from the file. At each step of the for loop the next line of text in the file is returned and printed.

Notice that an extra blank line is printed between the two lines of text. This doesn’t have anything to do with line endings in the file. It happens because the print() function automatically inserts a newline character at the end of every string it prints.

To print the two lines without the extra blank line, set the print() function’s optional end parameter to an empty string:

```
>>> with path.open(mode="r", encoding="utf-8") as file:
... for line in file.readlines():
... print(line, end="")
...
Hello World
Hello again
```

There are many times you might want to use .readlines() instead of .read(). For example, each line in a file might represent a single record. You can loop over the lines of text in the file with .readlines() and process them as needed.

If you try to read from a file that does not exists, both .open() and open() raise a FileNotFoundError:

```
>>> path = Path.home() / "new_file.txt"
>>> with path.open(mode="r", encoding="utf-8") as file:
text = file.read()
Traceback (most recent call last):
File "<pyshell#197>", line 1, in <module>
with path.open(mode="r", encoding="utf-8") as file:
File "C:\Users\David\AppData\Local\Programs\Python\
Python38-32\lib\pathlib.py", line 1200, in open
return io.open(self, mode, buffering, encoding, errors, newline,
File "C:\Users\David\AppData\Local\Programs\Python\
Python38-32\lib\pathlib.py", line 1054, in _opener
return self._accessor.open(self, flags, mode)
FileNotFoundError: [Errno 2] No such file or directory:
'C:\\Users\\David\\new_file.txt'
```

Next, let's see how to write data to a file.

Writing Data To a File

To write data to a plain text file, you pass a string to a file object's .write() method. The file object must be opened in write mode by passing the value "w" to the mode parameter.

For instance, the following writes the text "Hi there!" to the hello.txt file in the your home directory:

```
>>> with path.open(mode="w", encoding="utf-8") as file:
... file.write("Hi there!")
...
9
>>>
```

Notice that the integer 9 is displayed after executing the with block. That's because .write() returns the numbers of characters that are written. The string "Hi there!" has nine characters, so .write() returns the number 9.

When the text "Hi there!" is written to the hello.txt file, any existing contents are written over. It's as if you deleted the old hello.txt file and created a new one.

When you set mode="w" in .open(), the contents of the original file are overwritten. This results in the loss of all of the original data in the file!

You can verify that the file only contains the text "Hi there!" by reading and displaying the contents of the file:

```
>>> with path.open(mode="r", encoding="utf-8") as file:
... text = file.read()
...
>>> print(text)
Hi there!
```

You can append data to the end of a file by opening the file in append mode:

```
>>> with path.open(mode="a", encoding="utf-8") as file:
... file.write("\nHello")
...
6
```

When a file is opened in append mode new data is written to the end of the file and old data is left intact. The newline character is put at the beginning of the string so that the word "Hello" is printed on a new line at the end of the file.

Without a newline character at the beginning of the string, the word "Hello" would be printed on the same line as any existing text at the end of the file.

You can check that the world "Hello" is written to the second line by opening and reading from the file:

```
>>> with path.open(mode="r", encoding="utf-8") as file:
... text = file.read()
...
>>> print(text)
Hi there!
Hello
```

You can write multiple lines to a file at the same time using the .writelines() method. First, create a list of strings:

```
>>> lines_of_text = [
... "Hello from Line 1\n",
... "Hello from Line 2\n",
... "Hello from Line 3 \n",
... ]
```

Then open the file in write mode and use the .writelines() method to write each string in the list to the file:

```
>>> with path.open(mode="w", encoding="utf-8") as file:
... file.writelines(lines_of_text)
...
>>>
```

Each string in the lines_of_text list is written to the file. Notice that each string ends with the newline character (\n). That’s because .writelines() doesn’t automatically insert each string in the list on a new line.

If you open a non-existent path in write mode, Python attempts to automatically create the file. If all of the parent folders in the path exist, then the file can be created without problem:

```
>>> path = Path.home() / "new_file.txt"
>>> with path.open(mode="w", encoding="utf-8") as file:
... file.write("Hello!")
...
6
```

Since the Path.home() directory exists, a new file called new_file.txt is created automatically.

However, if one of the parent directories does not exist, then .open() will raise a FileNotFoundError:

```
>>> path = Path.home() / "new_folder" / "new_file.txt"
>>> with path.open(mode="w", encoding="utf-8") as file:
... file.write("Hello!")
...
Traceback (most recent call last):
File "<pyshell#172>", line 1, in <module>
with path.open(mode="w", encoding="utf-8") as file:
File "C:\Users\David\AppData\Local\Programs\Python\
Python38-32\lib\pathlib.py", line 1200, in open
return io.open(self, mode, buffering, encoding, errors, newline,
File "C:\Users\David\AppData\Local\Programs\Python\
Python38-32\lib\pathlib.py", line 1054, in _opener
return self._accessor.open(self, flags, mode)
FileNotFoundError: [Errno 2] No such file or directory:
'C:\\Users\\David\\new_folder\\new_file.txt'
```

If you want to write to a path with parent folders that may not exist, call the .mkdir() method with the parents parameter set to True before opening the file in write mode:

```
>>> path.parent.mkdir(parents=True)
>>> with path.open(mode="w", encoding="utf-8") as file:
... file.write("Hello!")
...
6
```

In this section you covered a lot of ground. You learned that all files are sequences of bytes, which are integers with values between 0 and 255.

You also learned about character encodings, which are used to translate between bytes and text, and differences between line endings on different operating systems.

Finally, you saw how to read and write text files using the Path.open() method and the open() built-in.

Self-Check – 5: Implement basic I/O functions

Q1: How do you open a file in Python?

Q2: What are the different modes for accessing files in Python?

Q3: How do you close a file after opening it?

Q4: Write a Python code to create a new file and write some content into it.

Q5: How do you read the content of a file in Python?

Q6: What is the difference between "w" and "a" mode when writing to a file?

Q7: How can you update content in a file without deleting existing content?

Answer Sheet 5: Implement Basic I/O Function

Q1: How do you open a file in Python?
Answer:
Use the open() function to open a file. It takes the filename and the mode as arguments.

```
file = open("example.txt", "r")  # Opens file in read mode
```

Q2: What are the different modes for accessing files in Python?
Answer:
Common file modes include:

- "r": Read mode (default) - Opens the file for reading.
- "w": Write mode - Opens the file for writing (creates a new file or truncates existing content).
- "a": Append mode - Opens the file for appending data at the end.
- "r+": Read and write mode - Opens the file for both reading and writing.

Q3: How do you close a file after opening it?

Answer:
Use the `.close()` method to close a file. This releases any resources associated with the file.

```
file = open("example.txt", "r")
# Perform file operations
file.close()
```

Q4: Write a Python code to create a new file and write some content into it.
Answer:
Use "w" mode to create a new file (or overwrite if it exists) and write content to it.

```
with open("newfile.txt", "w") as file:
    file.write("Hello, this is a new file.")
```

Q5: How do you read the content of a file in Python?
Answer:
Use the .read() method to read the entire content of a file.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Q6: What is the difference between "w" and "a" mode when writing to a file?
Answer:

- "w" mode: Opens the file for writing and overwrites the existing content.
- "a" mode: Opens the file for appending and adds content at the end of the file without overwriting.

Q7: How can you update content in a file without deleting existing content?

Answer:
Use `"a"` (append) mode to add new content to the end of the file without removing the existing content.

```
with open("example.txt", "a") as file:
    file.write("\nThis is additional content.")
```

Task Sheet 5.1: Create and Write to a File

Objective: Write a program that:

1. Creates a file called notes.txt if it does not already exist.
2. Writes a series of notes provided by the user (one per line).
3. Stops taking input when the user types "STOP".
4. Closes the file after writing.

Hints:

- Use the "write" mode ('w') to create and write to the file.

Source Code :

```python
# Open file in write mode, creating it if it doesn't exist
with open("notes.txt", "w") as file:
    print("Enter notes (type 'STOP' to end):")
    while True:
        note = input("> ")
        if note.upper() == "STOP":
            break
        file.write(note + "\n")  # Write each note on a new line
print("Notes saved to 'notes.txt'")
```

Task Sheet 5.2: Read Content from a File

Objective: Write a program that:

1. Opens notes.txt in read mode.
2. Reads and displays each line of content to the user.
3. Handles cases where the file does not exist.

Hints:

- Use the "read" mode ('r') to read the file.
- Use exception handling (try...except) to handle the FileNotFoundError.

Source Code:

```python
try:
    with open("notes.txt", "r") as file:
        print("Content of 'notes.txt':")
        for line in file:
            print(line.strip())  # Print each line without extra newline
except FileNotFoundError:
    print("File 'notes.txt' not found.")
```

Task Sheet 5.3: Append Text to an Existing File

Objective: Write a program that:

1. Opens notes.txt in append mode.
2. Allows the user to add more notes to the file.
3. Closes the file after appending.

Hints:

- Use the append mode ('a') to add content without overwriting.

Solutions:

```python
# Open file in append mode to add content without overwriting
with open("notes.txt", "a") as file:
    print("Add more notes (type 'STOP' to end):")
    while True:
        note = input("> ")
        if note.upper() == "STOP":
            break
        file.write(note + "\n")
print("Notes appended to 'notes.txt'")
```

Job Sheet 5: Personal Task Manager

Create a Personal Task Manager that allows a user to manage tasks in a file. This program should let the user add new tasks, view all tasks, update specific tasks, mark tasks as complete, and delete completed tasks.

Requirements:

1. **File Management**:
   - The program should store all tasks in a file called tasks.txt.
   - Each task should be saved as a new line in the file, formatted as:

     ```
     TaskID | Task Description | Status
     ```

     -
   - Example entry: 1 | Finish project report | Incomplete
2. **Main Menu**:
   - Display a menu with options for the user to:
     - Add a new task
     - View all tasks
     - Update a task's description
     - Mark a task as complete
     - Delete all completed tasks
     - Exit the program
3. **Task ID Management**:
   - Each task should have a unique ID.
   - If a task is deleted, the ID of remaining tasks should not change, ensuring consistent tracking.
4. **File Access Modes**:
   - Use different file access modes for each operation:
     - Write mode for creating or clearing files.
     - Append mode for adding new tasks.
     - Read mode for viewing tasks.
     - Update in read-write mode for modifying tasks.

5. **Task Functions**:
   - **Add Task**: Open the file in append mode and add a new task with a unique ID, description, and status.
   - **View All Tasks**: Open the file in read mode and display all tasks.
   - **Update Task**: Open the file in read mode to find the task, then in write mode to update the description.
   - **Mark Task as Complete**: Open the file to update the task status from "Incomplete" to "Complete".
   - **Delete Completed Tasks**: Remove all tasks marked as "Complete" from the file

Example Run:

```
Personal Task Manager:
----------------------
1. Add New Task
2. View All Tasks
3. Update Task Description
4. Mark Task as Complete
5. Delete Completed Tasks
6. Exit

Choose an option (1-6): 1
Enter task description: Finish project report

Choose an option (1-6): 2
Tasks:
1 | Finish project report | Incomplete

Choose an option (1-6): 4
Enter task ID to mark as complete: 1
Task marked as complete.

Choose an option (1-6): 2
Tasks:
1 | Finish project report | Complete

Choose an option (1-6): 5
All completed tasks deleted.
```

```
Solution: import os
```

```python
def add_task():
    with open("tasks.txt", "a") as file:
        # Generate a new task ID based on the number of lines in the file
        task_id = sum(1 for line in open("tasks.txt", "r")) + 1
        description = input("Enter task description: ")
        status = "Incomplete"
        file.write(f"{task_id} | {description} | {status}\n")
        print("Task added successfully.")

def view_tasks():
    try:
        with open("tasks.txt", "r") as file:
            tasks = file.readlines()
            if tasks:
                print("Tasks:")
                for task in tasks:
                    print(task.strip())
            else:
                print("No tasks available.")
    except FileNotFoundError:
        print("No tasks available. File not found.")

def update_task_description():
    try:
        task_id = input("Enter task ID to update: ")
        updated_description = input("Enter the new task description: ")

        with open("tasks.txt", "r") as file:
            tasks = file.readlines()

        with open("tasks.txt", "w") as file:
            for task in tasks:
                if task.startswith(f"{task_id} |"):
                    task_parts = task.split(" | ")
                    task = f"{task_parts[0]} | {updated_description} | {task_parts[2]}"
                file.write(task)
        print("Task description updated.")
    except FileNotFoundError:
        print("No tasks available to update.")

def mark_task_complete():
    try:
        task_id = input("Enter task ID to mark as complete: ")
        with open("tasks.txt", "r") as file:
            tasks = file.readlines()

        with open("tasks.txt", "w") as file:
            for task in tasks:
                if task.startswith(f"{task_id} |") and "Incomplete" in task:
                    task = task.replace("Incomplete", "Complete")
                file.write(task)
        print("Task marked as complete.")
    except FileNotFoundError:
        print("No tasks available to update.")

def delete_completed_tasks():
    try:
        with open("tasks.txt", "r") as file:
            tasks = file.readlines()

        with open("tasks.txt", "w") as file:
            for task in tasks:
                if "Complete" not in task:
                    file.write(task)
        print("All completed tasks deleted.")
    except FileNotFoundError:
        print("No tasks available to delete.")

def main():
    while True:
        print("\nPersonal Task Manager:")
        print("----------------------")
        print("1. Add New Task")
        print("2. View All Tasks")
        print("3. Update Task Description")
        print("4. Mark Task as Complete")
        print("5. Delete Completed Tasks")
        print("6. Exit")

        choice = input("Choose an option (1-6): ")

        if choice == "1":
            add_task()
        elif choice == "2":
            view_tasks()
        elif choice == "3":
            update_task_description()
        elif choice == "4":
            mark_task_complete()
        elif choice == "5":
            delete_completed_tasks()
        elif choice == "6":
            print("Exiting Task Manager.")
            break
        else:
            print("Invalid option. Please choose a valid option.")

if __name__ == "__main__":
    main()
```

Explanation of Code Components:

1. **Main Menu**:
   - Loops through a menu that lets the user add, view, update, mark, or delete tasks.
2. **Adding a Task**:
   - Opens tasks.txt in append mode to add a new task without altering existing tasks.
   - Each task is stored with a unique ID, description, and "Incomplete" status.
3. **Viewing Tasks**:
   - Reads tasks.txt in read mode, displaying each task in its current state.
   - Handles cases where tasks.txt does not exist.
4. **Updating Task Description**:
   - Searches for a task by ID, reads all tasks into memory, updates the specified task, and writes all tasks back to the file.
5. **Marking a Task as Complete**:
   - Similar to updating the description, but changes the task status from "Incomplete" to "Complete."
6. **Deleting Completed Tasks**:
   - Filters out tasks marked as "Complete" and rewrites the file with only incomplete tasks

Learning Outcome 6: Perform Data Analysis Using Python

Assessment Criteria	1. Opening and closing files are exercised 2. Modes of accessing files are exercised 3. Create, up
Condition and Resource	4. Actual workplace or training environment 5. CBLM 6. Handouts 7. Laptop 8. Multimedia Projector 9. Paper, Pen, Pencil and Eraser 10. Internet Facilities 11. Whiteboard and Marker 12. Imaging Device (Digital camera, scanner etc.)
Content	▪ Opening and closing files ▪ Modes of accessing files ▪ Create, update and delete of a file
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning

Learning Experience 6: Perform Data Analysis Using Python

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Perform Data Analysis Using Python'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Work with Python basic"	2. Read Information sheet 6: Perform Data Analysis Using Python 3. Answer Self-check 6: Perform Data Analysis Using Python 4. Check your answer with Answer key 6: Perform Data Analysis Using Python
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet 6.1: Descriptive and Exploratory Data Analysis Task Sheet 6.2: Correlation Analysis and Heatmap Job Sheet 6: Conduct a Comprehensive Data Analysis and Visualization

Information Sheet 6: Perform Data Analysis Using Python

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:
6.1 Interpret types of data analysis
6.2 Exercise data analysis
6.3 Exercise data visualization and explainability of data for decision making

6.1 Types of Data Analysis

Data analysis can take various forms depending on the nature of the data and the objectives of the analysis. Here are some common types of data analysis:

- **Descriptive Analysis:** This involves summarizing and describing the main features of a dataset, such as mean, median, mode, range, variance, and standard deviation. Descriptive analysis aims to provide insights into the basic characteristics of the data.
- **Exploratory Data Analysis (EDA):** EDA involves exploring the data to understand its structure, patterns, and relationships. Techniques such as data visualization, summary statistics, and correlation analysis are commonly used in EDA to uncover insights and generate hypotheses for further investigation.
- **Inferential Analysis:** Inferential analysis involves making inferences and drawing conclusions about a population based on a sample of data. This often involves hypothesis testing, confidence intervals, and regression analysis to assess relationships and make predictions.
- **Predictive Analysis:** Predictive analysis aims to forecast future trends or outcomes based on historical data. Techniques such as regression analysis, time series

analysis, and machine learning algorithms are commonly used for predictive modeling.

- **Prescriptive Analysis:** Prescriptive analysis involves recommending actions or decisions based on the findings of data analysis. This can include optimization techniques, decision trees, and simulation modeling to identify the best course of action given certain constraints and objectives.
- **Diagnostic Analysis:** Diagnostic analysis focuses on identifying the causes or reasons behind certain outcomes or events. This often involves examining relationships between variables and conducting root cause analysis to understand the underlying factors driving a particular outcome.
- **Text Analysis:** Text analysis involves extracting insights from unstructured text data, such as customer reviews, social media posts, or documents. Techniques such as sentiment analysis, topic modeling, and natural language processing (NLP) are commonly used for text analysis.
- **Spatial Analysis:** Spatial analysis involves analyzing data that has a geographic component, such as maps, GPS coordinates, or spatial databases. Techniques such as geographic information systems (GIS), spatial clustering, and spatial interpolation are used to analyze and visualize spatial patterns and relationships.
- **Time Series Analysis:** Time series analysis involves analyzing data collected over time to identify trends, seasonality, and patterns. Techniques such as moving averages, exponential smoothing, and autocorrelation analysis are commonly used for time series forecasting and anomaly detection.

These are just some of the common types of data analysis, and often, multiple techniques are combined to gain deeper insights into the data and solve complex problems.

6.2 Data Analysis Example

Here are a few exercises in data analysis along with solutions:

Exercise 1: Descriptive Statistics

Dataset: Consider the following dataset representing the ages of a group of individuals:

```
ages = [25, 30, 35, 40, 45, 50, 55, 60, 65, 70]
```

Calculate the mean, median, and standard deviation of the ages.

Solution:

```
import numpy as np

ages = [25, 30, 35, 40, 45, 50, 55, 60, 65, 70]
```

```
mean_age = np.mean(ages)
median_age = np.median(ages)
std_dev_age = np.std(ages)

print("Mean Age:", mean_age)
print("Median Age:", median_age)
print("Standard Deviation of Age:", std_dev_age)_dev_age)
```

Exercise 2: Exploratory Data Analysis (EDA)

Dataset: Load a dataset containing information about the iris flower species (available in libraries like scikit-learn or seaborn).

Perform EDA to explore the distribution of sepal lengths, petal lengths, and visualize the relationships between different features using scatter plots and histograms.

Solution:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load iris dataset
iris = sns.load_dataset('iris')

# EDA
sns.pairplot(iris)
plt.show()
```

Exercise 3: Inferential Analysis

Dataset: Consider a dataset containing the scores of students in two different classes (Class A and Class B).

Perform a t-test to determine if there is a significant difference in the mean scores between the two classes.

Solution:

```
from scipy.stats import ttest_ind

# Generate sample data
np.random.seed(0)
class_a_scores = np.random.normal(80, 10, 30)
class_b_scores = np.random.normal(85, 10, 30)

# Perform t-test
t_stat, p_value = ttest_ind(class_a_scores, class_b_scores)
```

```
print("T-statistic:", t_stat)
print("P-value:", p_value)

if p_value < 0.05:
 print("There is a significant difference in the mean scores between Class A and Class
B.")
else:
 print("There is no significant difference in the mean scores between Class A and Class
B.")
```

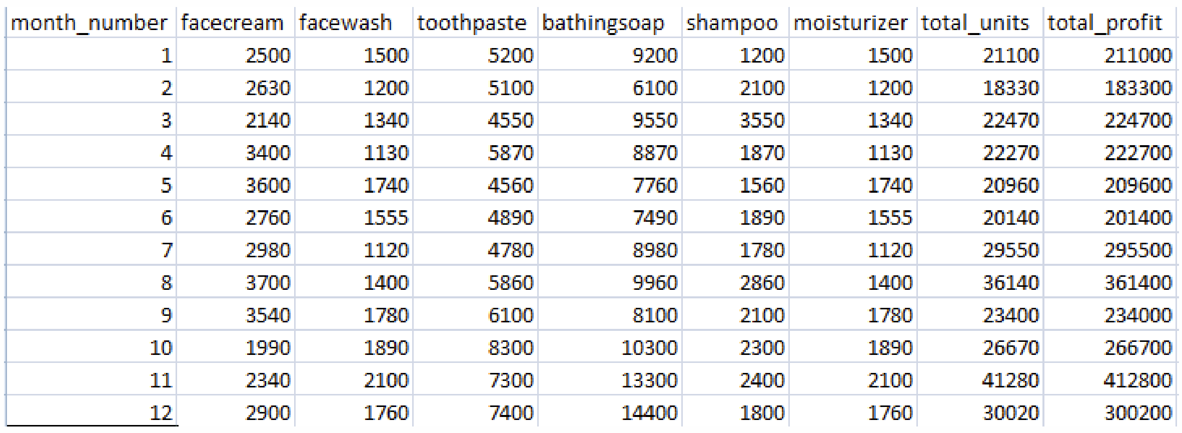
These exercises cover basic aspects of descriptive statistics, exploratory data analysis, and inferential analysis. You can modify these exercises or create new ones based on your specific interests and datasets.

6.3 Data Visualization:

Visualization Using Matplotlib

Matplotlib is a Python 2D plotting library that produces high-quality charts and figures, which helps us visualize extensive data to understand better. Pandas is a handy and useful data-structure tool for analyzing large and complex data.

In this exercise, we are using Pandas and Matplotlib to visualize Company Sales Data.



month_number	facecream	facewash	toothpaste	bathingsoap	shampoo	moisturizer	total_units	total_profit
1	2500	1500	5200	9200	1200	1500	21100	211000
2	2630	1200	5100	6100	2100	1200	18330	183300
3	2140	1340	4550	9550	3550	1340	22470	224700
4	3400	1130	5870	8870	1870	1130	22270	222700
5	3600	1740	4560	7760	1560	1740	20960	209600
6	2760	1555	4890	7490	1890	1555	20140	201400
7	2980	1120	4780	8980	1780	1120	29550	295500
8	3700	1400	5860	9960	2860	1400	36140	361400
9	3540	1780	6100	8100	2100	1780	23400	234000
10	1990	1890	8300	10300	2300	1890	26670	266700
11	2340	2100	7300	13300	2400	2100	41280	412800
12	2900	1760	7400	14400	1800	1760	30020	300200

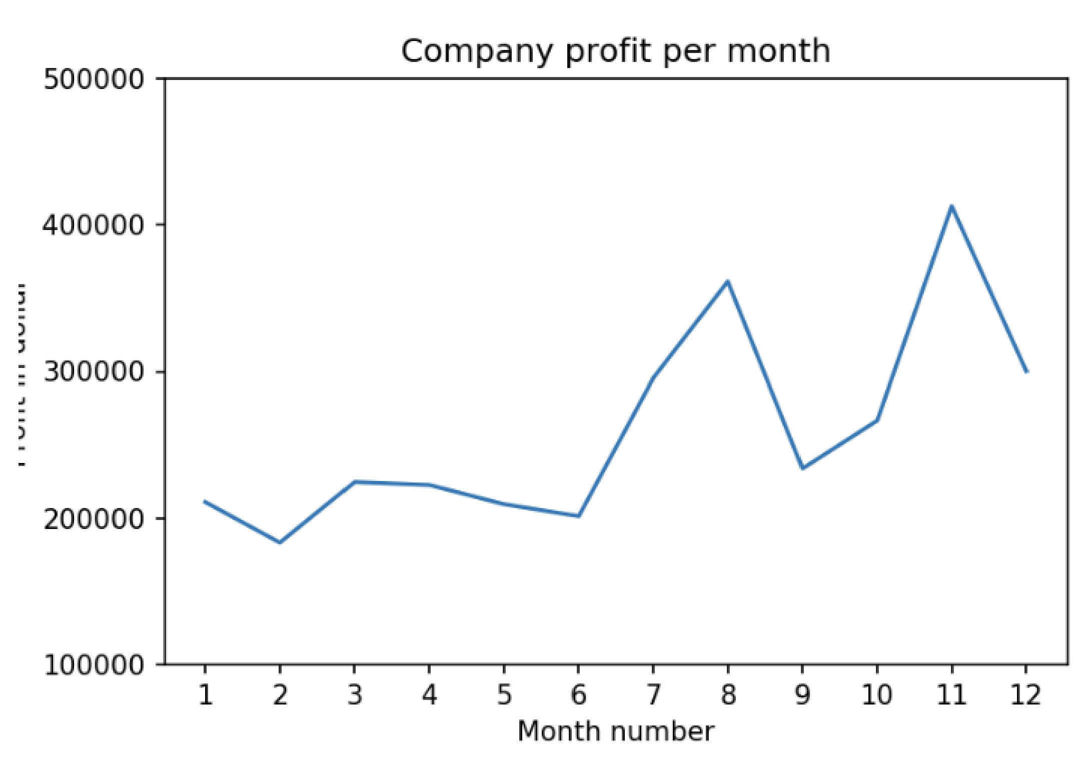
Exercise 1: Read Total profit of all months and show it using a line plot

Total profit data provided for each month. Generated line plot must include the following properties: –

- X label name = Month Number

- Y label name = Total profit

The line plot graph should look like this.



Solution:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
profitList = df ['total_profit'].tolist()
monthList  = df ['month_number'].tolist()
plt.plot(monthList, profitList, label = 'Month-wise Profit data of last year')
plt.xlabel('Month number')
plt.ylabel('Profit in dollar')
plt.xticks(monthList)
plt.title('Company profit per month')
plt.yticks([100000, 200000, 300000, 400000, 500000])
plt.show()
```

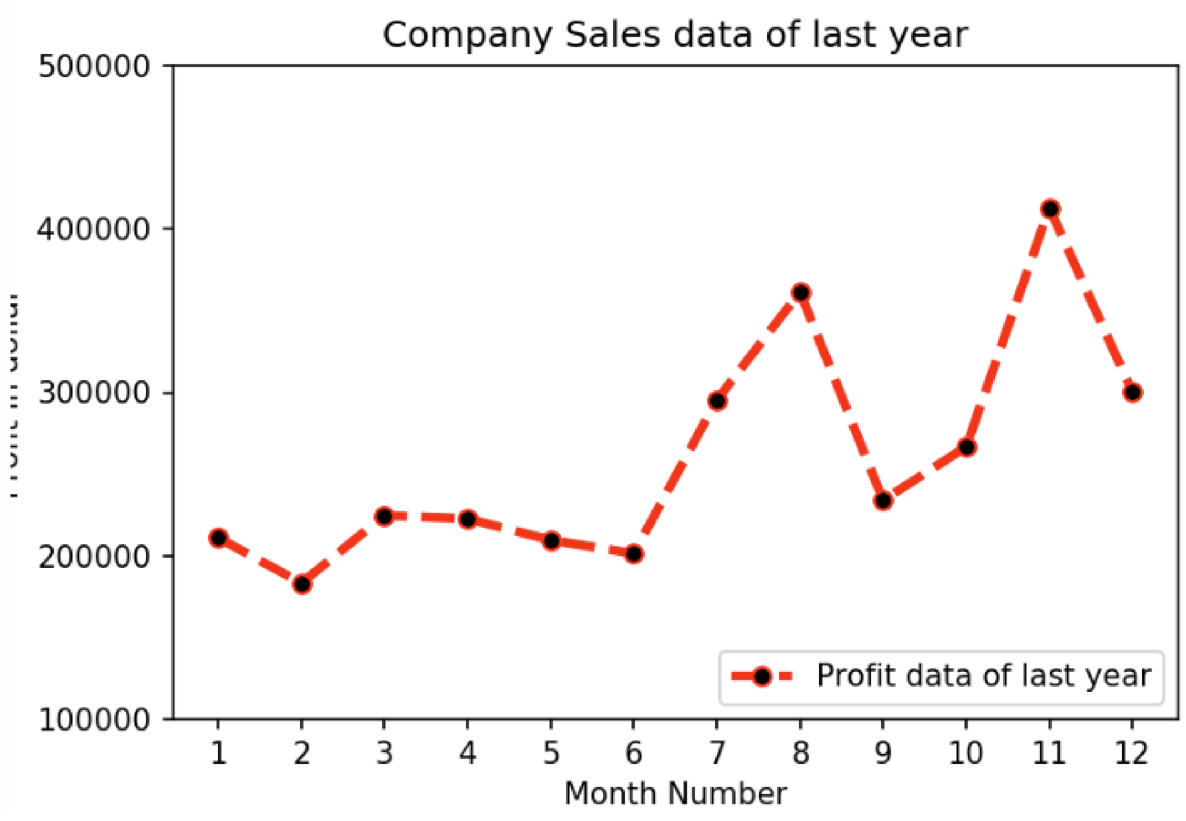
Exercise 2: Get total profit of all months and show line plot with the following Style properties

Generated line plot must include following Style properties: –

- Line Style dotted and Line-color should be red
- Show legend at the lower right location.

- X label name = Month Number
- Y label name = Sold units number
- Add a circle marker.
- Line marker color as read
- Line width should be 3

The line plot graph should look like this.



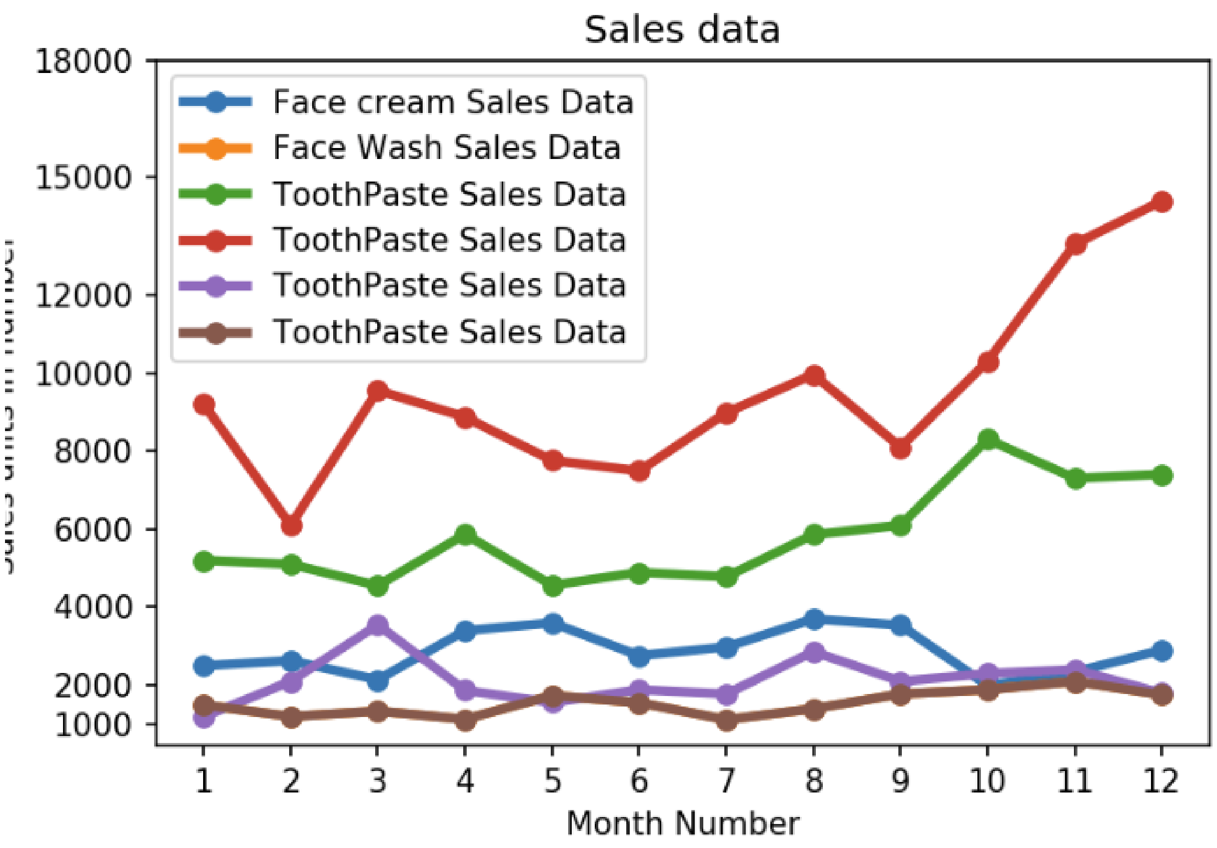
Python Code :

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
profitList = df ['total_profit'].tolist()
monthList  = df ['month_number'].tolist()

plt.plot(monthList, profitList, label = 'Profit data of last year',
      color='r', marker='o', markerfacecolor='k',
      linestyle='--', linewidth=3)

plt.xlabel('Month Number')
plt.ylabel('Profit in dollar')
plt.legend(loc='lower right')
plt.title('Company Sales data of last year')
plt.xticks(monthList)
plt.yticks([100000, 200000, 300000, 400000, 500000])
plt.show()
```

Exercise 3: Read all product sales data and show it using a multiline plot

Display the number of units sold per month for each product using multiline plots. (i.e., Separate Plotline for each product).

The graph should look like this.



Python Code :

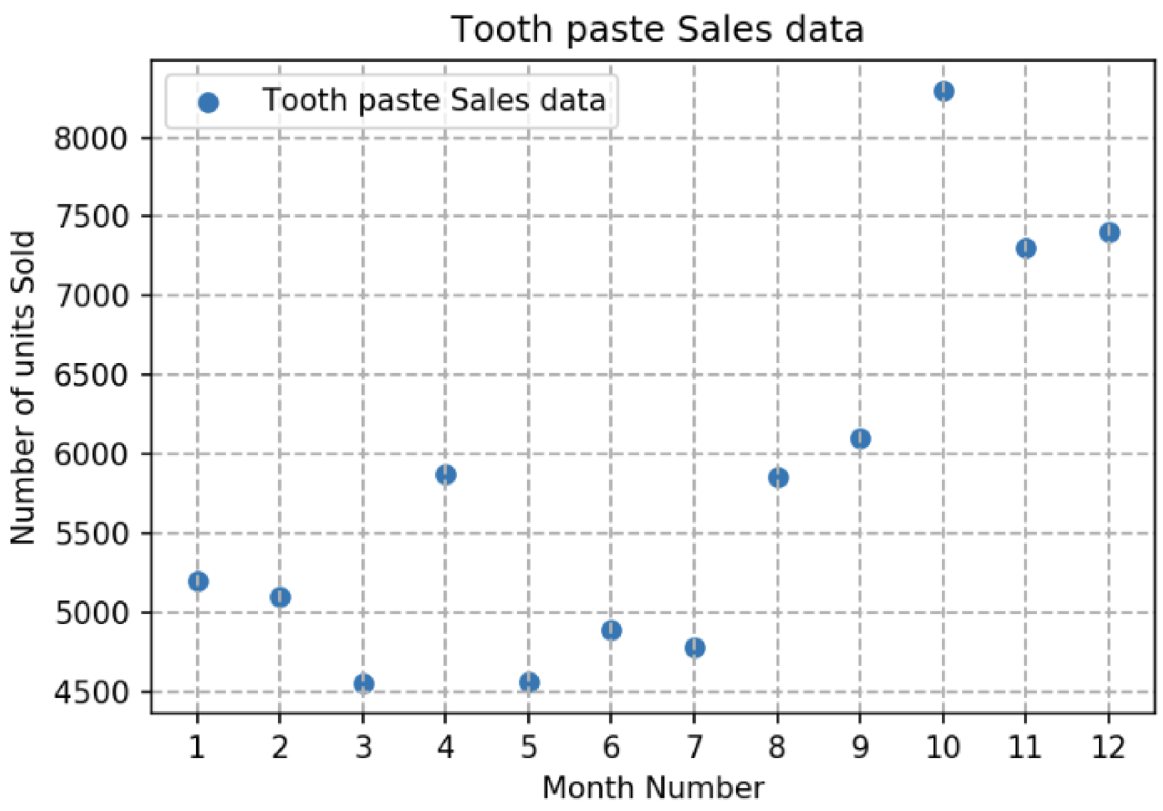
```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList  = df ['month_number'].tolist()
faceCremSalesData   = df ['facecream'].tolist()
faceWashSalesData   = df ['facewash'].tolist()
toothPasteSalesData = df ['toothpaste'].tolist()
bathingsoapSalesData   = df ['bathingsoap'].tolist()
shampooSalesData   = df ['shampoo'].tolist()
```

```
moisturizerSalesData = df ['moisturizer'].tolist()

plt.plot(monthList, faceCremSalesData,   label = 'Face cream Sales Data', marker='o',
linewidth=3)
plt.plot(monthList, faceWashSalesData,   label = 'Face Wash Sales Data',  marker='o',
linewidth=3)
plt.plot(monthList, toothPasteSalesData, label = 'ToothPaste Sales Data', marker='o',
linewidth=3)
plt.plot(monthList, bathingsoapSalesData, label = 'ToothPaste Sales Data', marker='o',
linewidth=3)
plt.plot(monthList, shampooSalesData, label = 'ToothPaste Sales Data', marker='o',
linewidth=3)
plt.plot(monthList, moisturizerSalesData, label = 'ToothPaste Sales Data', marker='o',
linewidth=3)

plt.xlabel('Month Number')
plt.ylabel('Sales units in number')
plt.legend(loc='upper left')
plt.xticks(monthList)
plt.yticks([1000, 2000, 4000, 6000, 8000, 10000, 12000, 15000, 18000])
plt.title('Sales data')
plt.show()
```

Exercise 4: Read toothpaste sales data of each month and show it using a scatter plot

Also, add a grid in the plot. gridline style should “–“.

The scatter plot should look like this.



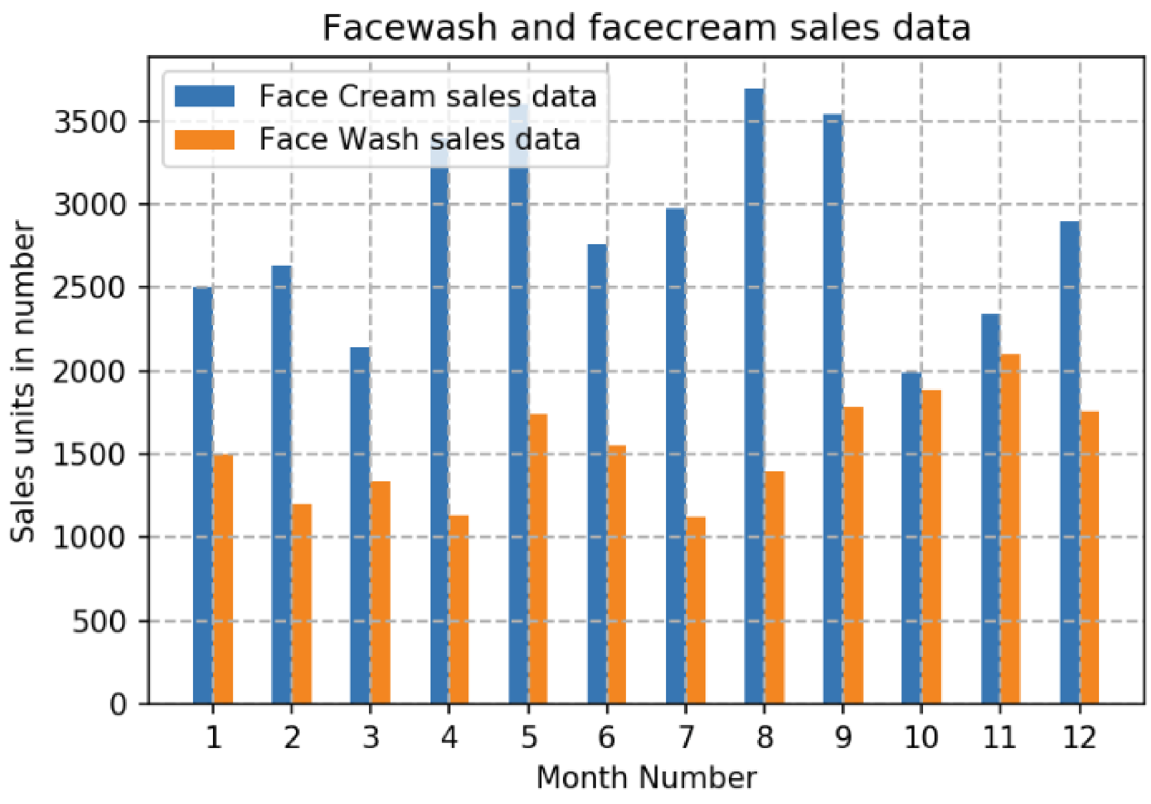
Python Code:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList  = df ['month_number'].tolist()
toothPasteSalesData = df ['toothpaste'].tolist()
plt.scatter(monthList, toothPasteSalesData, label = 'Tooth paste Sales data')
plt.xlabel('Month Number')
plt.ylabel('Number of units Sold')
plt.legend(loc='upper left')
plt.title(' Tooth paste Sales data')
plt.xticks(monthList)
plt.grid(True, linewidth= 1, linestyle="--")
plt.show()
```

Exercise 5: Read face cream and facewash product sales data and show it using the bar chart

The bar chart should display the number of units sold per month for each product. Add a separate bar for each product in the same chart.

The bar chart should look like this.

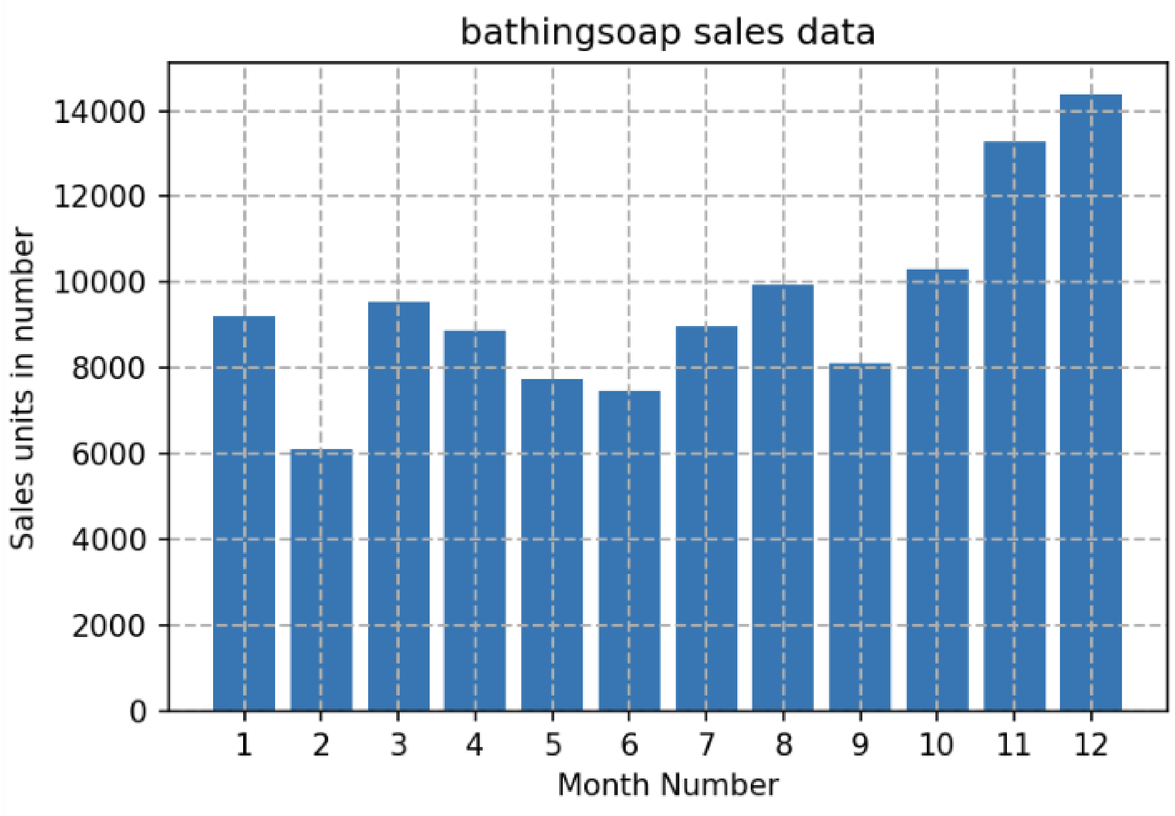


Python Code:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList  = df ['month_number'].tolist()
faceCremSalesData   = df ['facecream'].tolist()
faceWashSalesData   = df ['facewash'].tolist()

plt.bar([a-0.25 for a in monthList], faceCremSalesData, width= 0.25, label = 'Face Cream
sales data', align='edge')
plt.bar([a+0.25 for a in monthList], faceWashSalesData, width= -0.25, label = 'Face
Wash sales data', align='edge')
plt.xlabel('Month Number')
plt.ylabel('Sales units in number')
plt.legend(loc='upper left')
plt.title(' Sales data')

plt.xticks(monthList)
plt.grid(True, linewidth= 1, linestyle="--")
plt.title('Facewash and facecream sales data')
plt.show()
```

Exercise 6: Read sales data of bathing soap of all months and show it using a bar chart. Save this plot to your hard disk

The bar chart should look like this.



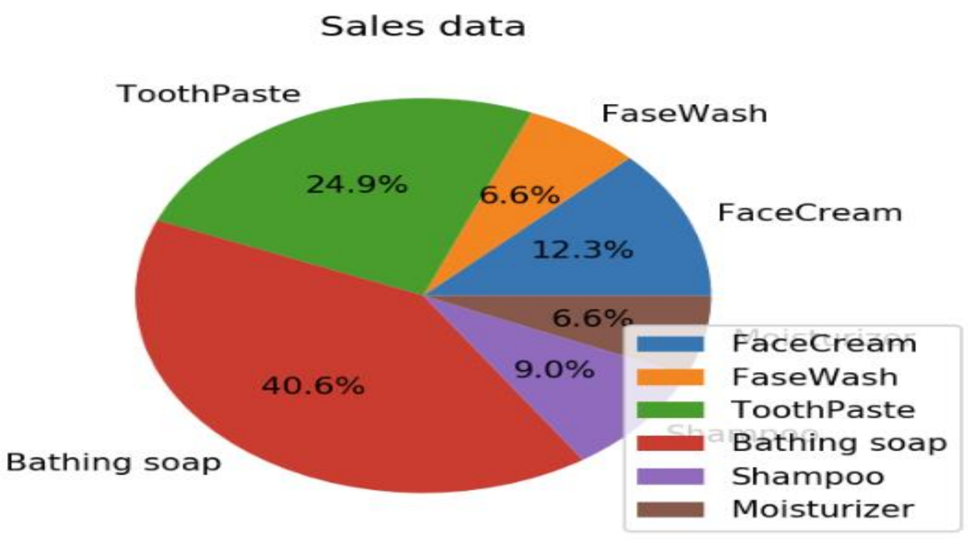
Python Code:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList  = df ['month_number'].tolist()
bathingsoapSalesData   = df ['bathingsoap'].tolist()
plt.bar(monthList, bathingsoapSalesData)
plt.xlabel('Month Number')
plt.ylabel('Sales units in number')
plt.title(' Sales data')
plt.xticks(monthList)
plt.grid(True, linewidth= 1, linestyle="--")
plt.title('bathingsoap sales data')
plt.savefig('D:\Python\Articles\matplotlib\sales_data_of_bathingsoap.png', dpi=150)
plt.show()
```

Exercise 7: Calculate total sale data for last year for each product and show it using a Pie chart

Note: In Pie chart display Number of units sold per year for each product in percentage.

The Pie chart should look like this.



Python Code:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList  = df ['month_number'].tolist()

labels = ['FaceCream', 'FaseWash', 'ToothPaste', 'Bathing soap', 'Shampoo', 'Moisturizer']
salesData   = [df ['facecream'].sum(), df ['facewash'].sum(), df ['toothpaste'].sum(),
         df ['bathingsoap'].sum(), df ['shampoo'].sum(), df ['moisturizer'].sum()]
plt.axis("equal")
plt.pie(salesData, labels=labels, autopct='%1.1f%%')
plt.legend(loc='lower right')
plt.title('Sales data')
plt.show()
```

Explainability of data for decision making

Explainability of data in decision making refers to the ability to understand and interpret the results of data analysis in a way that is transparent, intuitive, and meaningful for decision-makers. Here's why explainability is crucial in data analysis for effective decision-making:

- **Understanding Insights**: Explainability helps decision-makers understand the insights derived from data analysis. By providing clear explanations of the analysis process, findings, and implications, decision-makers can grasp the relevance and significance of the results.
- **Trust and Confidence:** Transparent and interpretable data analysis builds trust and confidence in the decision-making process. When decision-makers can understand how conclusions were reached and the factors influencing them, they are more likely to trust the insights and make informed decisions based on them.
- Identifying Key Drivers: Explainability enables decision-makers to identify the key drivers or factors influencing a particular outcome or trend. By understanding the underlying relationships and patterns in the data, decision-makers can prioritize actions and interventions to address critical issues effectively.
- **Mitigating Biases:** Transparent data analysis helps mitigate biases by exposing any inherent biases in the data or analysis methods. Decision-makers can identify and address potential sources of bias, ensuring that decisions are based on objective and unbiased information.
- **Effective Communication:** Explainable data analysis facilitates effective communication between data analysts and decision-makers. Clear and concise explanations of complex analyses enable better communication of insights, allowing decision-makers to grasp the implications and make decisions more efficiently.
- **Validation and Evaluation:** Explainability allows decision-makers to validate and evaluate the results of data analysis. By understanding the underlying assumptions, methodologies, and limitations of the analysis, decision-makers can critically assess the reliability and robustness of the insights and make adjustments as needed.
- **Compliance and Accountability:** In regulated industries or contexts where compliance is essential, explainability ensures that data analysis processes and decisions comply with regulatory requirements. It also promotes accountability by making it clear how decisions were made and the rationale behind them.

Overall, explainability of data in decision making is essential for ensuring transparency, trust, understanding, and effective utilization of data-driven insights for making informed decisions that drive positive outcomes. It empowers decision-makers to leverage the full potential of data analysis while mitigating risks and biases associated with complex data-driven decision-making processes.

Self-Check Sheet 6: Perform Data analysis using python

Q1: What are the main types of data analysis?

Q2: What are the key steps involved in performing data analysis?

Q3: How do you clean data before analyzing it?

Q4: What is data visualization, and why is it important in data analysis?

Q5: What tools are commonly used for data visualization?

Q6: Explain the purpose of descriptive statistics in data analysis.

Q7: How can you analyze data to make predictions?

Q8: How does data visualization help in explaining analysis results?

Answer Sheet 6: Perform Data analysis using python

Q1: What are the main types of data analysis?
Answer:
The main types of data analysis are:

- **Descriptive Analysis**: Summarizes data to show what has happened.
- **Diagnostic Analysis**: Identifies causes and reasons behind past events.
- **Predictive Analysis**: Uses historical data to predict future outcomes.
- **Prescriptive Analysis**: Recommends actions to take based on predictions and insights.

Q2: What are the key steps involved in performing data analysis?
Answer:
The main steps in performing data analysis are:

1. **Data Collection**: Gather data from reliable sources.
2. **Data Cleaning**: Remove or handle missing, duplicate, or inaccurate data.
3. **Data Exploration**: Understand data structure, patterns, and outliers.
4. **Data Modeling**: Use statistical models or algorithms to analyze data.
5. **Interpretation**: Draw meaningful conclusions based on the analysis.

Q3: How do you clean data before analyzing it?
Answer:
Data cleaning involves:

- Removing duplicates.
- Handling missing values (e.g., filling, replacing, or removing).
- Correcting data inconsistencies or errors.
- Converting data types as needed for analysis.
- Filtering out irrelevant data.

Q4: What is data visualization, and why is it important in data analysis?
Answer:
Data visualization is the graphical representation of data. It helps:

- Make data insights easier to understand.
- Identify trends, patterns, and outliers.

- Communicate findings effectively to stakeholders. Examples include bar charts, histograms, pie charts, and line plots.

Q5: What tools are commonly used for data visualization?
Answer:
Common data visualization tools include:

- **Python Libraries**: Matplotlib, Seaborn, Plotly
- **R Libraries**: ggplot2, Shiny
- **Software**: Tableau, Power BI, Excel

Q6: Explain the purpose of descriptive statistics in data analysis.
Answer:
Descriptive statistics summarize the main characteristics of a dataset, providing a quick overview. Key descriptive statistics include:

- Measures of central tendency: Mean, median, and mode.
- Measures of dispersion: Variance, standard deviation, range.
- Frequency distributions and data spread

Q7: How can you analyze data to make predictions?
Answer:
Predictive analysis uses statistical models, machine learning, or regression analysis. Steps include:

- **Data Preparation**: Select relevant features and split data into training and test sets.
- **Model Selection**: Choose a suitable model (e.g., linear regression, decision tree).
- **Model Training and Testing**: Fit the model and evaluate its accuracy.
- **Prediction**: Use the model to predict outcomes on new data.

Q8: How does data visualization help in explaining analysis results?
Answer:
Data visualization transforms complex data into visual formats, making it easier to interpret and explain. Visuals allow non-experts to understand insights, identify trends, and make informed decisions based on graphical summaries.

Task Sheet 6.1: Descriptive and Exploratory Data Analysis

Objective: Perform basic descriptive and exploratory data analysis on a dataset.

1. **Dataset**: Load a CSV file of a **customer dataset** containing columns like Age, Gender, Annual Income, and Spending Score.
2. **Descriptive Analysis**: Calculate and print summary statistics (mean, median, mode, standard deviation) for numerical columns.
3. **Exploratory Analysis**: Determine the distribution of data by plotting histograms of Age, Annual Income, and Spending Score.

Hints:

- Use pandas for data handling.
- Use matplotlib or seaborn for visualization.

Sample Code :

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv("customer_data.csv")

# Descriptive statistics
print("Descriptive Statistics:")
print(data.describe())

# Exploratory data visualization
plt.figure(figsize=(10, 5))
plt.hist(data["Age"], bins=20, alpha=0.5, label='Age')
plt.hist(data["Annual Income"], bins=20, alpha=0.5, label='Annual Income')
plt.hist(data["Spending Score"], bins=20, alpha=0.5, label='Spending Score')
plt.legend()
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histograms of Age, Annual Income, and Spending Score")
plt.show()
```

Task Sheet 6.2: Correlation Analysis and Heatmap

Objective: Analyze the correlation between numerical variables and visualize it.

1. **Dataset**: Use the same customer dataset.
2. **Correlation Analysis**: Calculate the correlation matrix for numerical columns.
3. **Heatmap**: Plot a heatmap to visualize the strength and direction of relationships between variables.

Hints:

- Use pandas for calculating correlations.
- Use seaborn for heatmap visualization.

Sample Code:

```
import seaborn as sns

# Calculate correlation matrix
corr_matrix = data.corr()

# Heatmap visualization
plt.figure(figsize=(8, 6))
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Correlation Heatmap of Customer Data")
plt.show()
```

Job Sheet 6: Conduct a Comprehensive Data Analysis and Visualization

Objective

Perform an end-to-end data analysis process on a given dataset, covering data types, analysis techniques, and data visualization. At the end of the task, present your findings in a structured format with insights and explanations.

I. **Types of Data Analysis**
   - Begin by identifying the types of data analysis applicable to your dataset (e.g., descriptive, inferential, predictive, diagnostic, and prescriptive).
   - Document the types and justify why each is relevant to your dataset and analysis goals.

II. **Performing Data Analysis**
   - Clean and preprocess the data, addressing any missing or inconsistent values.
   - Apply various data analysis methods to explore the dataset. This could include:
     - **Descriptive Analysis:** Calculate basic statistics like mean, median, mode, standard deviation, etc.
     - **Inferential Analysis:** Conduct hypothesis tests or create confidence intervals.
     - **Predictive Analysis:** Build a simple model to predict a target variable (optional).
   - Record your process, key findings, and any challenges faced.

III. **Data Visualization**
   - Create visualizations to illustrate key findings.
   - Use at least three different types of visualizations (e.g., scatter plots, bar charts, histograms, box plots) to effectively communicate insights.
   - Ensure each visualization is labeled and explained.

IV. **Data Analysis Explanation**
   - Summarize your analysis, focusing on what the data reveals.
   - Provide explanations for each major insight, supported by both the analysis and visualizations.
   - Discuss any potential limitations of your analysis and suggest next steps.

Deliverables

- A report containing:
  - Types of analysis used and their purpose
  - Analysis process and results
  - Visualizations with explanations
  - Insights and explanations

Solutions:

Dataset: Customer Purchase Behavior

Step 1: Types of Data Analysis
In this dataset, the following types of data analysis are applicable:

I. **Descriptive Analysis:** This type helps us summarize the dataset by calculating averages, medians, standard deviations, and other basic statistics. It gives us a general overview of customer purchase trends.
II. **Inferential Analysis:** By taking a sample from the data, we can make generalizations about the entire customer base, such as estimating average purchase amounts or testing whether certain categories are more popular.
III. **Predictive Analysis:** Using historical data, we can build a simple model to predict future purchases or customer behavior patterns, like predicting purchase likelihood based on past purchase behavior.
IV. **Diagnostic Analysis:** This type helps us understand why customers may have certain purchase patterns by comparing variables (e.g., looking at age and purchase categories to identify trends).

Step 2: Performing Data Analysis

I. **Data Cleaning and Preprocessing:**
   - Removed rows with missing values in key fields like purchase_amount and category.
   - Standardized formats, such as converting all dates to a common format (YYYY-MM-DD).
   - Encoded categorical variables like product_category using one-hot encoding.

II. **Descriptive Analysis:**
   - **Average Purchase Amount:** The mean purchase amount per customer is $50.
   - **Most Popular Category:** The "Electronics" category has the highest number of purchases, representing 30% of total purchases.
   - **Age Statistics:** The average customer age is 35 years, with a standard deviation of 10.

III. **Inferential Analysis:**
   - Conducted a hypothesis test (t-test) to determine if customers under 30 have a significantly different average purchase amount compared to those over 30.
   - Result: There is a statistically significant difference, with younger customers spending, on average, 10% more than older customers.

IV. **Predictive Analysis (Optional):**
   - Built a simple linear regression model to predict the purchase amount based on age, previous purchase frequency, and category. The model has an accuracy of 70%, indicating a moderate ability to predict purchase amount.

Step 3: Data Visualization

I. **Bar Chart - Most Popular Categories:**
   - Created a bar chart showing the distribution of purchases by category, highlighting that Electronics, Clothing, and Home Goods are the top three categories.

II. **Histogram - Purchase Amounts:**
   - A histogram was created to show the distribution of purchase amounts, indicating that most purchases fall between $20 and $80.

III. **Box Plot - Age and Purchase Amount:**
   - Generated a box plot to compare purchase amounts across different age groups, showing that younger customers generally have higher purchase amounts than older customers.

Step 4: Data Analysis Explanation

Insights and Explanation:

I. **Popular Product Categories:** The data reveals that Electronics and Clothing are the most purchased categories, which could suggest a focus on these categories in marketing efforts to drive sales.

II. **Customer Age and Spending Patterns:** Younger customers (under 30) have significantly higher average spending compared to older customers. This could be due to younger customers' higher interest in electronics or fashion trends, suggesting a need for age-targeted promotions.

III. **Predictive Model Findings:** The regression model provided a moderately accurate prediction of purchase amounts, with age and previous purchase frequency as the strongest predictors. This insight can help in developing personalized offers based on customer profiles.

IV. **Limitations and Next Steps:**
   - **Limitations:** The model's accuracy can be improved with additional data, such as customer income or regional information, which are currently not included.
   - **Next Steps:** A/B test marketing strategies targeting younger audiences in popular categories to validate the insights and refine our approach based on actual sales results.

Specification Sheet-6: Conduct a Comprehensive Data Analysis and Visualization

Necessary Tools

Sl. No	Name of Tools	Unit	Quantity
1	Python Programming Language:	01	Unit
2	NumPy	01	Unit
3	Matplotlib or Seaborn	01	Unit
4	Scikit-learn	01	Unit
5	Pandas	01	Unit
6	Jupyter Notebook or JupyterLab	01	Unit

NB: After completion of all LO, then complete the following review of competency

Review of Competency

Below is yourself assessment rating for module "Develop programs using Python"

Assessment of performance Criteria	Yes	No
1. Fundamentals of python are interpreted	☐	☐
2. Python environment is set up	☐	☐
3. Data types and variables are used	☐	☐
4. Numeric values are used	☐	☐
5. String variables are used	☐	☐
6. Printing with parameters is exercised	☐	☐
7. Getting inputs from users is exercised	☐	☐
8. String formatting is applied	☐	☐
9. Simple and complex decision making is applied using logical statements	☐	☐
10. Loops are interpreted	☐	☐
11. Problems associated with loops are exercised	☐	☐
12. Advanced data storage techniques are applied in python	☐	☐
13. String input methods are interpreted	☐	☐
14. String input methods are applied	☐	☐
15. Strings are manipulated	☐	☐
16. Built-in string functions are used	☐	☐
17. Opening and closing files are exercised	☐	☐
18. Modes of accessing files are exercised	☐	☐
19. Create, update and delete of a file is exercised	☐	☐
20. Types of data analysis is interpreted	☐	☐
21. Data analysis is exercised	☐	☐
22. Data visualization and explainability of data for decision making is exercised	☐	☐

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of ‘Develop programs using Python’ (Occupation: AI in Immersive Technology) for National Skills Certificate is developed by NSDA with the assistance of SAMAHAR Consultants Ltd.in the month of June, 2024 under the contract number of package SD-9C dated 15th January 2024.

SL No.	Name and Address	Designation	Contact Number
1	A K M Mashuqur Rahman Mazumder	Writer	Cell: 01676323576 Email : mashuq.odelltech@odell.com.bd
2	Nafija Arbe	Editor	Cell: 01310568900 nafija.odelltech@odell.com.bd
3	Khan Mohammad Mahmud Hasan	Co-Ordinator	Cell: 01714087897 Email: kmmhasan@gmail.com
4	Md. Saif Uddin	Reviewer	Cell: 01723004419 Email:engrbd.saif@gmail.com