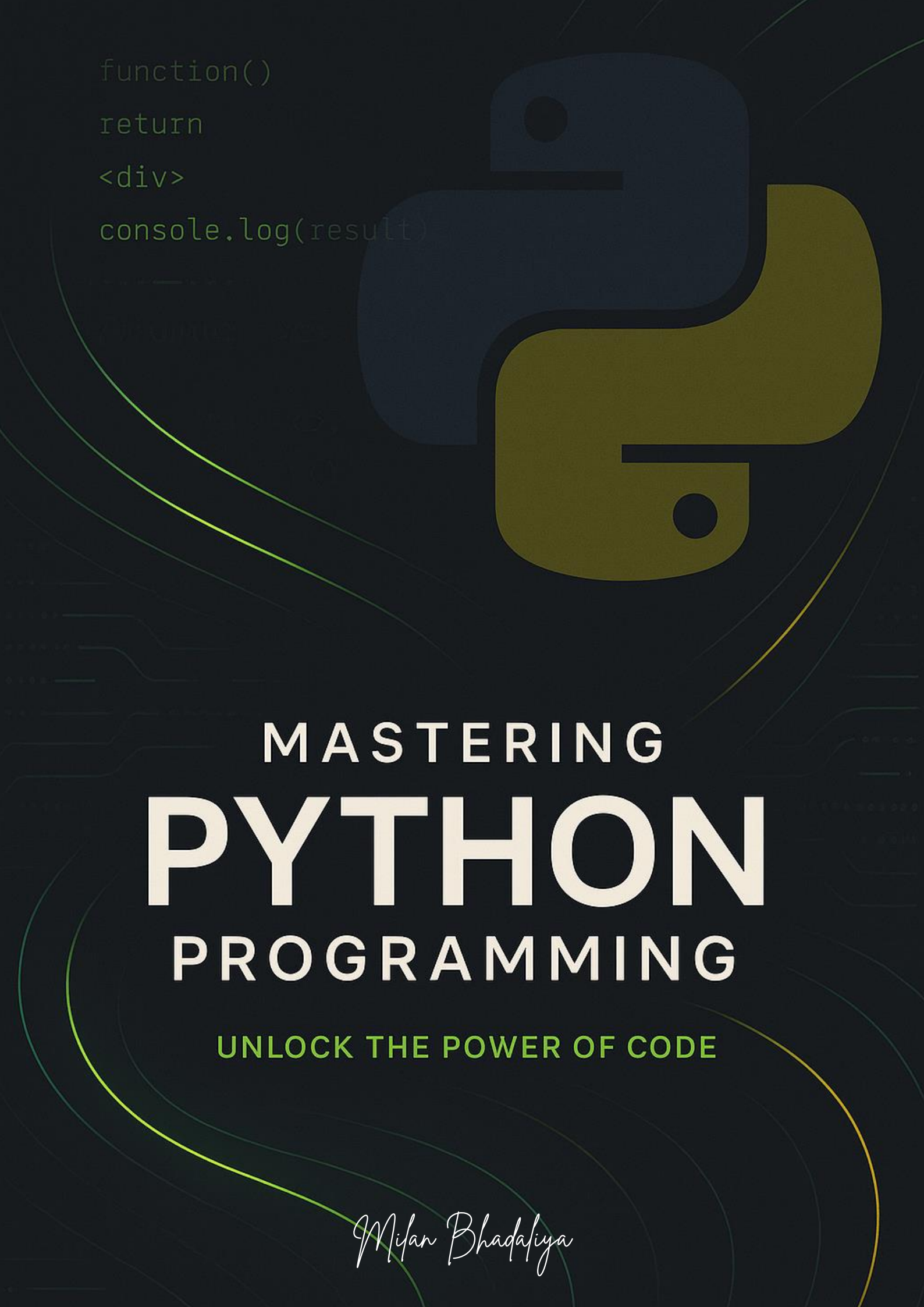


MASTERING PYTHON PROGRAMMING

UNLOCK THE POWER OF CODE

Milan Bhadaliya

Chapter 1

Basics of Python

Chapter 2

String & Lists

Chapter 3

Dictionaries, Tuples and Sets

Chapter 4

Control Flow Statements

Chapter 5

Files

Chapter 6

Introduction to Data science

6.1 Introduction to Libraries

6.2 What is PIP?

6.3 NumPy: Creating, accessing, manipulating array, performing various operations on array.

6.4 Pandas: Functions to create, access and manipulate sequence and data frame from file

6.5 Matplotlib: Line Graphs, Scatter Graph, Pie Charts, Bar Charts, Figures and Subplot

6.6 Scikitlearn: Linear regression, K-Nearest neighbour classifier, Logistic regression, Decision Tree, Random Forest classifier Clustering and anomaly detection

6.7 What are the reasons Python is considered the top choice for data visualization?

1.1 Python Introduction

→ Python is a popular programming language.

→ It was created by Guido van Rossum, and released in 1991.

→ Python is a versatile language with a wide range of **use cases.**

Here are some key areas where Python excels:

1. **Web Development:** Python is popular for building web applications. Frameworks like Django and Flask help developers create robust, scalable web applications quickly and efficiently.
2. **Data Analysis and Visualization:** Python is widely used in data science. Libraries like Pandas, NumPy, and Matplotlib make it easy to analyze, manipulate, and visualize data.
3. **Machine Learning and AI:** Python is a leading language in the field of artificial intelligence and machine learning. Libraries such as TensorFlow, PyTorch, and Scikit-Learn provide tools for building and training models.
4. **Automation and Scripting:** Python is great for automating repetitive tasks and writing scripts. It can be used for everything from simple file manipulations to complex workflows.
5. **Game Development:** Python can be used for game development with libraries like Pygame, which provides tools for creating 2D games.
6. **Desktop GUI Applications:** Python can be used to build cross-platform desktop applications with libraries like Tkinter, PyQt, and Kivy.
7. **Networking:** Python is useful for network programming. Libraries like Socket and Twisted provide tools for creating networked applications and services.
8. **Cybersecurity:** Python is used in cybersecurity for tasks such as penetration testing, malware analysis, and writing security tools and scripts.
9. **Scientific Computing:** Python is used in scientific research and engineering for tasks that involve numerical simulations and complex computations. Libraries like SciPy and SymPy are commonly used.
10. **Web Scraping:** Python can be used to extract data from websites using libraries like BeautifulSoup and Scrapy.
11. **Database Interaction:** Python can interact with various types of databases, both SQL (e.g., SQLite, PostgreSQL) and NoSQL (e.g., MongoDB), using libraries like SQLAlchemy and PyMongo.

12. **Embedded Systems:** Python can be used in embedded systems and IoT (Internet of Things) projects with libraries like MicroPython and CircuitPython.
13. **Finance and Trading:** Python is used for financial analysis, algorithmic trading, and quantitative analysis. Libraries like QuantLib and Zipline are popular in this domain.

These use cases illustrate Python's broad applicability across various fields and industries.

1.2 Python Installation

To check if we have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
python --version
```

If we find that we do not have Python installed on our computer, then we can download it for free from the following website: **https://www.python.org/**

1.3 Python Editors

A Python editor is a software application designed to facilitate writing, editing, and managing Python code.

1. **IDLE**

→ Developer: Python Software Foundation
→ Features: Simple and lightweight, included with Python installation, basic editor and interpreter.

2. **PyCharm**

→ Developer: JetBrains
→ Features: Code completion, debugging, version control integration, extensive plugins, support for web frameworks.
→ Versions: Community (free), Professional (paid).

3. **Visual Studio Code (VS Code)**

→ Developer: Microsoft
→ Features: Lightweight, customizable with extensions, integrated terminal, Git support, IntelliSense.
→ Extensions: Python extension by Microsoft adds features like debugging and linting.

4. **Spyder**

→ Developer: Anaconda Inc.
→ Features: Scientific Python development, integrated IPython console, variable explorer, support for data science libraries.
→ Target Audience: Data scientists, engineers.

5. **Sublime Text** (Text Editor)

→ Developer: Sublime HQ
→ Features: Fast, customizable, supports many languages with plugins, great for quick edits.
→ Plugins: Python-related packages available via Package Control.

6. **Notepad++** (Text Editor)

→ Developer: Don Ho
→ Features: Lightweight, syntax highlighting, supports plugins.
→ Plugins: Python plugins available for added functionality.

7. **Jupyter Notebook / JupyterLab**

→ Developer: Project Jupyter
→ Features: Interactive computing environment, support for rich media, integrates code and documentation.
→ Use Cases: Data analysis, machine learning, and interactive computing.

8. **Google Colab**

→ Developer: Google

→ Features: Cloud-based Jupyter notebook environment, free access to GPUs, easy sharing.

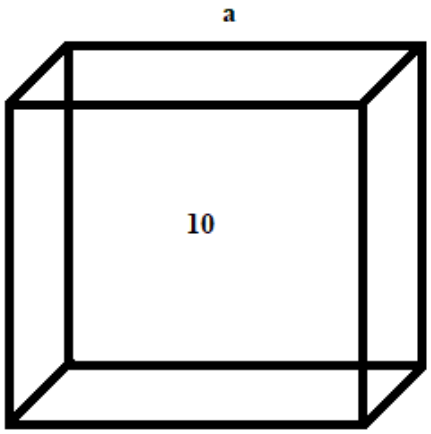
→ Use Cases: Data science, collaborative projects.

1.4 Variables

→ Variables are containers for storing data values.

Example:

a=10



Creating Variables

→ Python has no command for declaring a variable.

→ A variable is created as soon as we first assign a value to it.

→ Variables do not need to be declared with any particular type, and can even change type after they have been set.

Example:

```
x = 10
y = "Milan"
print(x)
print(y)
```

Casting

→ If we want to specify the data type of a variable, this can be done with casting.

Example:

```
x = str(10)    # x will be '10'
y = int(10)    # y will be 10
z = float(10)  # z will be 10.0
```

Get the Type

→ We can get the data type of a variable with the type() function.

Example:

```
x = 10
y = "Milan"
print(type(x))
print(type(y))
```

Single or Double Quotes?

→ String variables can be declared either by using single or double quotes.

Example:

```
x = "Milan"
# is the same as
x = 'Milan'
```

Case-Sensitive

→ Variable names are case-sensitive.

Example:

```
a = 10
A = "Milan"
#A will not overwrite a both variables are different
```

★ Identifier

→ An identifier is a name used to identify a variable, function, class, or other user-defined item.

Rules for Python variables declaration:

→ A variable name must start with a letter or the underscore character
→ A variable name cannot start with a number
→ A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
→ Variable names are case-sensitive (age, Age and AGE are three different variables)
→ A variable name cannot be any of the Python keywords.

Valid variable names:

Example:

```
myvar = "Milan"
my_var = "Milan"
_my_var = "Milan"
myVar = "Milan"
MYVAR = "Milan"
myvar2 = "Milan"
```

Invalid variable names:

Example:

```
2myvar = "Milan"
my-var = "Milan"
my var = "Milan"
```

Note: Remember that variable names are case-sensitive

★Extra:

Multi Words Variable Names

→ Variable names with more than one word can be difficult to read.
→ There are several techniques we can use to make them more readable.

1 Camel Case

→ Each word, except the first, starts with a capital letter.

Example:

```
myVariableName = "Milan"
```

2 Pascal Case

→ Each word starts with a capital letter.

Example:

```
MyVariableName = "Milan"
```

3 Snake Case

→ Each word is separated by an underscore character.

Example:

```
my_variable_name = "Milan"
```

Assign Multiple Values to Multiple Variables at a same time

→ Python allows us to assign values to multiple variables in one line.

Example:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

Note: Make sure the number of variables matches the number of values, or else we will get an error.

One Value to Multiple Variables

→ And we can assign the same value to multiple variables in one line.

Example:

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Unpack a Collection

→ If we have a collection of values in a list, tuple etc. Python allows us to extract the values into variables. This is called unpacking.

Example:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Output Variables

→ The Python print() function is often used to output variables.

Example:

```
x = "Python is awesome"
print(x)
```

→ In the print() function, we can output multiple variables by separating them with commas.

Example:

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

→ We can also use the + operator to output multiple variables.

Example:

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

Notice the space character after "Python " and "is ", without them the result would be "Pythonisawesome".

→ For numbers, the + character works as a mathematical operator.

Example:

```
x = 5
y = 10
print(x + y)
```

→ In the print() function, when we try to combine a string and a number with the + operator, Python will give us an error.

Example:

```
x = 5
y = "Milan"
print(x + y)
```

→ The best way to output multiple variables in the print() function is to separate them with commas, which even support different data types.

Example:

```
x = 5
y = "Milan"
print(x, y)
```

1.5 Data Types

Built-in Data Types

→ Variables can store data of different types, and different types can do different things.

→ Python has the following data types built-in by default, in these categories:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview
None Type:	NoneType

Example:

x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "Milan", "age" : 18}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview
x = None	NoneType

→ If we want to specify the data type, we can use the following constructor functions:

x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="Milan", age=20)	dict
x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

1.6 Operators

→ Operators are used to perform operations on variables and values.

→ Python divides the operators in the following groups:

1 Arithmetic operators
2 Assignment operators
3 Comparison operators
4 Logical operators
5 Identity operators
6 Membership operators
7 Bitwise operators

1 Arithmetic operators

→ Arithmetic operators are used with numeric values to perform common mathematical operations.

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

2 Assignment operators

→ Assignment operators are used to assign values to variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3
:=	print(x := 3)	x = 3 print(x)

3 Comparison operators

→ Comparison operators are used to compare two values.

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

4 Logical operators

→ Logical operators are used to combine conditional statements.

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

5 Identity operators

→ Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

6 Membership operators

→ Membership operators are used to test if a sequence is presented in an object.

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

7 Bitwise operators

→ Bitwise operators are used to compare (binary) numbers.

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Operator Precedence

→ Operator precedence describes the order in which operations are performed.

Example:

→ Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

```
print((6 + 3) - (6 + 3))
```

Example:

→ Multiplication * has higher precedence than addition +, and therefor multiplications are evaluated before additions:

```
print(100 + 5 * 3)
```

→ The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
()	Parentheses
**	Exponentiation
+x -x ~x	Unary plus, unary minus, and bitwise NOT
* / // %	Multiplication, division, floor division, and modulus
+ -	Addition and subtraction
<< >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
== != > >= < <= is is not in not in	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

1.7 Python Indentation

→ Indentation in Python refers to the spaces at the start of a line of code.

→ Unlike many other programming languages where indentation is used merely for readability, in Python it has a crucial role.

→ Python relies on indentation to define blocks of code, making it essential for the proper execution and structure of the program.

Example:

True Syntax:	
1.	if 5 > 2: print("Five is greater than two!")

2.	if 5 > 2:
	  print("Five is greater than two!")
	if 5 > 2:
	  print("Five is greater than two!")

False Syntax:	
1.	if 5 > 2: print("Five is greater than two!")
2.	if 5 > 2: print("Five is greater than two!") print("Five is greater than two!")

1.8 Comments

→ Python supports comments for in-code documentation.

→ Comments begin with a # symbol, and everything following this symbol on the same line is treated as a comment by Python.

→ Comments can be used to explain Python code.

→ Comments can be used to make the code more readable.

→ Comments can be used to prevent execution when testing code.

Single line Comments:

→ A single-line comment is a short note in our code that helps explain what the code does. It starts with a special symbol so that the computer ignores it.

Example:

```
#Example of comment.
print("Hello, World!")
```

Multiline Comments:

→ A multiline comment is a note in our code that can span several lines. It helps explain parts of our code in more detail and is ignored by the computer when running the program.

→ Python does not really have a syntax for multiline comments.

→ To add a multiline comment we could insert a # for each line:

Example:

```
#Example of Multiline comment
#written in
#more than one line
print("Hello, World!")
```

→ But we can use a multiline string as a multiline comment. Because Python will ignore string literals that are not assigned to a variable, we can add a multiline string (triple quotes) in our code, and place our comment inside it:

```
"""
Example of Multiline comment
written in
more than one line
"""
print("Hello, World!")
```

1.9 Python end parameter in print()

By default Python‘s print() function ends with a newline. Python’s print() function comes with a parameter called ‘end‘. By default, the value of this parameter is ‘\n’, i.e. the new line character.

Example 1:

To prevent a newline character from being added after a print statement, set the end parameter to an empty string:

```
print("Hello", end="")
print("world!")
```

Output:

```
Helloworld!
```

Example 2:
Here, we can end a print statement with any character/string using this parameter.

```
# ends the output with a space
print("Milan", end = ' ')
print("Bhadaliya")
```

Output:

```
Milan Bhadaliya
```

Example 3:

```
# ends the output with '@'
print("Milan", end='@')
print("Bhadaliya")
```

Output:

```
Milan@Bhadaliya
```

Example 4:
The print() function uses the sep parameter to separate the arguments and ends after the last argument.

```
print("Milan", end='@')
print("Bhadaliya")
print('M','J', sep='', end='')
print('B')
#\n provides new line after printing the year
print('21','12','2012', sep='-', end='\n')
print('Milan','Bhadaliya', sep='.', end='@')
print('Professor')
```

Output:

```
Milan@Bhadaliya
MJB
21-12-2012
Milan.Bhadaliya@Professor
```

1.10 Python Casting (Typecasting)

In Python, "casting" refers to the process of converting a variable from one type to another. This is often necessary when you need to perform operations that require specific data types or when you're dealing with data from various sources that need to be in a particular format.

Example:

```
#int casting
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
print("x= ",x)
print("y= ",y)
print("z= ",z)

#float casting
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
print("x= ",x)
print("y= ",y)
print("z= ",z)
print("w= ",w)

#String casting
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
print("x= ",x)
print("y= ",y)
print("z= ",z)
```

Output:

```
x=  1
y=  2
z=  3
x=  1.0
y=  2.8
z=  3.0
w=  4.2
x=  s1
y=  2
z=  3.0
```

1 int(): Converts a value to an integer.

```
x = "123"
y = int(x)  # y is now an integer 123
```

2 float(): Converts a value to a floating-point number.

```
x = "123.45"
y = float(x)  # y is now a float 123.45
```

3 str(): Converts a value to a string.

```
x = 123
y = str(x)  # y is now the string "123"
```

4 list(): Converts a value to a list.

```
x = (1, 2, 3)
y = list(x)  # y is now [1, 2, 3]
```

5 tuple(): Converts a value to a tuple.

```
x = [1, 2, 3]
y = tuple(x)  # y is now (1, 2, 3)
```

6 set(): Converts a value to a set.

```
x = [1, 2, 2, 3]
y = set(x)  # y is now {1, 2, 3}
```

Output:

```
y=  123
y=  123.45
y=  123
y=  [1, 2, 3]
y=  (1, 2, 3)
y=  {1, 2, 3}
```

1.11 Python input() Function

Syntax

```
input(prompt)
```

Example:

```
print('Enter your name:')
x = input()
print('Hello, ' + x)
```

Output:

```
Enter your name:
milan
Hello, milan
```

Example:

Use the prompt parameter to write a message before the input:

```
x = input('Enter your name: ')
print('Hello, ' + x)
```

Output:

```
Enter your name: Milan
Hello, Milan
```

1.12 Hybrid Nature of Python (Interpreted and Compiled)

Python is primarily an interpreted language, but it has characteristics of both interpreted and compiled languages. Here's a breakdown:

→ **Interpreted Nature**

- **Execution:** Python code is executed line-by-line by an interpreter, which means that the code is read and executed in real-time. This allows for a more interactive development process.
- **Interpreter:** The standard Python interpreter is CPython, which compiles Python code into bytecode and then interprets it at runtime. Other Python interpreters include Jython (Python on the JVM), IronPython (Python for .NET), and PyPy (an optimized Python interpreter).

→ **Compiled Aspects**

- **Bytecode Compilation:** When you run a Python program, the source code (.py files) is compiled into bytecode (.pyc files), which is a lower-level, platform-independent representation of the source code. This bytecode is then interpreted by the Python virtual machine.
- **JIT Compilation:** Some Python implementations, like PyPy, use Just-In-Time (JIT) compilation techniques to improve execution speed by compiling Python code into machine code at runtime.

→ **Hybrid Nature**

- Compilation and Interpretation: Python involves both compilation (to bytecode) and interpretation (execution of bytecode), blending aspects of both approaches. This hybrid nature allows Python to offer ease of development with dynamic features, while also striving for performance improvements.

Overall, Python is best described as an interpreted language, with compilation steps that enhance execution efficiency.

Python Code Flow

1. **Source Code (.py)**

   Write Python code in a `.py` file.

2. **Compilation to Bytecode (.pyc)**

   Python interpreter compiles the `.py` file into bytecode (`.pyc`).

3. **Execution by Python Virtual Machine**

   The Python Virtual Machine executes the bytecode during runtime.

Comparison to Java

- Java Source Code (.java): Written in '.java' files.
- Compilation to Bytecode (.class): Compiled into bytecode ('.class' files) by the Java compiler.
- Execution by Java Virtual Machine (JVM): JVM executes the bytecode.

Both languages have a multi-stage process involving source code, compilation to an intermediate representation, and execution by a virtual machine, but the specifics differ in terms of the intermediate forms and execution mechanisms.

1. Source Code File (.py)

→ File Extension: Python source code is written in files with a `.py` extension. For example, `example.py`.

→ Contents: This file contains human-readable Python code.

2. Compilation to Bytecode (.pyc)

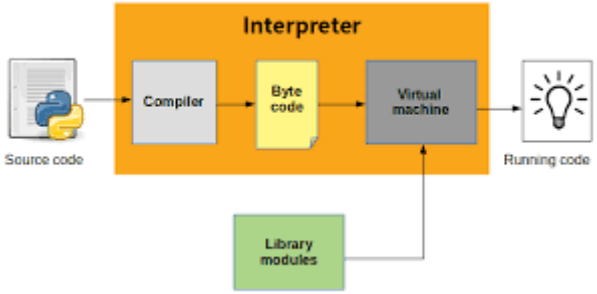
→ Compilation: When you run a Python script, the Python interpreter compiles the source code into bytecode. Bytecode is a lower-level, platform-independent representation of the source code. This compilation step is typically done automatically by the interpreter.

→ Bytecode File: The compiled bytecode is stored in `.pyc` files, which are usually found in the `__pycache__` directory. For example, `example.cpython-38.pyc` (where `38` corresponds to Python version 3.8).

3. Execution by the Python Virtual Machine

→ Python Virtual Machine (PVM): The Python interpreter includes a virtual machine that executes the bytecode. The PVM interprets the bytecode and performs the operations as specified in the source code.

→ Runtime Execution: The Python interpreter reads and executes the bytecode line-by-line or in blocks, performing operations such as function calls, control flow, and interaction with external systems.



2.1 String representation

→ Strings in python are surrounded by either single quotation marks, or double quotation marks.

→ 'hello' is the same as "hello".

You can display a string literal with the print() function:

```
print("Hello")
print('Hello')
```

Assign String to a Variable

→ Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

```
a = "Hello"
print(a)
```

Multiline Strings

→ You can assign a multiline string to a variable by using three quotes:

Code:

```
a = """o stree kal aana
aaj shanivar hai"""
print(a)
```

Output:

```
o stree kal aana
aaj shanivar hai
```

Note: in the result, the line breaks are inserted at the same position as in the code.

Strings are Arrays

→ Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

→ However, Python does not have a character data type, a single character is simply a string with a length of 1.

→ Square brackets can be used to access elements of the string.

Example:

Get the character at position 1 (remember that the first character has the position 0)

```
a = "Hello, World!"
print(a[1])
```

Output:

```
e
```

Looping Through a String

→ Since strings are arrays, we can loop through the characters in a string, with a for loop.

Example:
Loop through the letters in the sentence “o stree kal aana”:

```
for x in “o stree kal aana”:
        print(x)
```

Output:

```
o stree kal aana
```

2.2 String Operations

Check String

→ To check if a certain phrase or character is present in a string, we can use the keyword in.

Example:
Check if "stree" is present in the following text:

```
txt = "o stree kal aana"
print("stree" in txt)
```

Use it in an if statement

Example:
Print only if "stree" is present:

```
txt = "o stree kal aana"
if "stree" in txt:
  print("Yes, 'stree' is present.")
```

Check if NOT

→ To check if a certain phrase or character is NOT present in a string, we can use the keyword not in.

Example:
Check if "stree" is NOT present in the following text:

```
txt = "o stree kal aana"
print("stree" not in txt)
```

Use it in an if statement

Example:
print only if "stree" is NOT present:

```
txt = "o stree kal aana"
if "stree" not in txt:
  print("No, 'stree' is NOT present. Stree kal aayegi")
```

2.3 Indexing and slicing

→ You can return a range of characters by using the slice syntax.

→ Specify the start index and the end index, separated by a colon, to return a part of the string.

Example:

Get the characters from position 2 to position 5 (not included):

```
a = "Hello, World!"
print(a[2:5])
```

Output:

```
llo
```

Note: The first character has index 0.

Slice From the Start

→ By leaving out the start index, the range will start at the first character:

Example:

Get the characters from the start to position 5 (not included):

```
a = "Hello, World!"
print(a[:5])
```

Output:

```
Hello
```

Slice To the End

→ By leaving out the end index, the range will go to the end:

Example:

Get the characters from position 2, and all the way to the end:

```
a = "Hello, World!"
print(a[2:])
```

Negative Indexing

→ Use negative indexes to start the slice from the end of the string:

Example:

Get the characters

From: "o" in "World!" (Position -5)

To, but not included: "d" in "World!" (Position -2):

```
a = "Hello, World!"
print(a[-5:-2])
```

Output:

```
orl
```

2.4 other string operations

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

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

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts

rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Escape Character

→ To insert characters that are illegal in a string, use an escape character.

→ An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
txt = "O "Stree" kal aana." #error
```

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

2.5 Creating Lists

→ Lists are used to store multiple items in a single variable.
→ Lists are created using square brackets.
→ List items are ordered, changeable, and allow duplicate values.
→ List items are indexed, the first item has index [0], the second item has index [1] etc.

Example:

```
demolist = ["1010", "123.23", "MB"]
print(demolist)
```

Ordered

→ When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
→ If you add new items to a list, the new items will be placed at the end of the list.

Changeable | Mutable

→ The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

→ Since lists are indexed, lists can have items with the same value.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "John Wick", "John Wick"]
print(demolist)
```

List Length

→ To determine how many items a list has, use the len() function:

Example:

Print the number of items in the list.

```
demolist = ["Spiderman", "John Wick", "batman"]
print(len(demolist))
```

type()

→ From Python's perspective, lists are defined as objects with the data type 'list'.

```
<class 'list'>
```

Example:

```
demolist = ["Spiderman", "John Wick", "batman"]
print(type(demolist))
```

2.6 Basic List Indexing and slicing

Access Items

→ List items are indexed and you can access them by referring to the index number:

Example:

Print the second item of the list:

```
demolist = ["Spiderman", "John Wick", "batman"]
print(demolist[1])
```

Note: The first item has index 0.

Negative Indexing

→ Negative indexing means start from the end

→ -1 refers to the last item, -2 refers to the second last item etc.

Example:

Print the last item of the list.

```
demolist = ["Spiderman", "John Wick", "batman"]
print(demolist[-1])
```

Range of Indexes

→ You can specify a range of indexes by specifying where to start and where to end the range.

→ When specifying a range, the return value will be a new list with the specified items.

Example:

Return the third, fourth, and fifth item.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor",
"Black Widow", "Captain America"]
print(demolist[2:5])
```

Note: The search will start at index 2 (included) and end at index 5 (not included).
Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item

Example:

This example returns the items from the beginning to, but NOT including, "Wonder Woman"

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor",
"Black Widow", "Captain America"]
print(demolist[:4])
```

By leaving out the end value, the range will go on to the end of the list

Example:

This example returns the items from "batman" to the end.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor",
"Black Widow", "Captain America"]
print(demolist[2:])
```

Range of Negative Indexes

→ Specify negative indexes if you want to start the search from the end of the list

Example:

This example returns the items from "Wonder Woman" (-4) to, but NOT including "Captain America" (-1)

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
print(demolist[-4:-1])
```

Check if Item Exists

→ To determine if a specified item is present in a list use the in keyword.

Example:

Check if "John Wick" is present in the list.

```
demolist = ["Spiderman", "John Wick", "batman"]
if "John Wick" in demolist:
  print("Yes, 'John Wick' is in the demolist")
```

2.7 Built-In Functions Used on Lists

Change Item Value

→ To change the value of a specific item, refer to the index number.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist[1] = "the amazing spider-man"
print(demolist)
```

Change a Range of Item Values

→ To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist[5:6] = ["Loki", "Superman"]
print(demolist)
```

→ If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist[5:6] = ["Loki", "Superman", "Deadpool"]
print(demolist)
```

Note: The length of the list will change when the number of items inserted does not match the number of items replaced.

→ If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist[5:6] = ["Loki"]
print(demolist)
```

Insert Items

→ To insert a new list item, without replacing any of the existing values, we can use the insert() method.

→ The insert() method inserts an item at the specified index.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.insert(2, "Loki")
print(demolist)
```

Append Items

→ To add an item to the end of the list, use the append() method.

Example:

Using the append() method to append an item.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.append("Loki")
print(demolist)
```

Extend List

To append elements from another list to the current list, use the extend() method.

Example:

```
thislist = ["Thor", "Black Widow", "Captain America", "Loki"]
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman"]
thislist.extend(demolist)
print(thislist)
```

Remove Specified Item

The remove() method removes the specified item.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.remove("Captain America")
print(demolist)
```

If there are more than one item with the specified value, the remove() method removes the first occurrence.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", , "Captain America", "Iron Man", , "Captain America", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.remove("Captain America")
print(demolist)
```

Remove Specified Index

The pop() method removes the specified index.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.pop(6)
print(demolist)
```

If you do not specify the index, the pop() method removes the last item

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.pop()
print(demolist)
```

The del keyword also removes the specified index

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
del demolist[6]
print(demolist)
```

The del keyword can also delete the list completely

Example:

Delete the entire list.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
del demolist
```

Clear the List

→ The clear() method empties the list, The list still remains, but it has no content.

Example:

Clear the list content.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.clear()
print(demolist)
```

Loop Through a List

→ You can loop through the list items by using a for loop.

Example:

Print all items in the list, one by one.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
for x in demolist:
  print(x)
```

Loop Through the Index Numbers

→ You can also loop through the list items by referring to their index number.

→ Use the range() and len() functions to create a suitable iterable.

Example:

Print all items by referring to their index number.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
for i in range(len(demolist)):
  print(demolist[i])
```

Sort List

→ List objects have a sort() method that will sort the list alphanumerically, ascending, by default.

Example:

Sort the list alphabetically

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.sort()
print(demolist)
```

Example:

Sort the list numerically.

```
demolist = [100, 50, 65, 82, 23]
demolist.sort()
print(demolist)
```

Sort Descending

To sort descending, use the keyword argument **reverse = True.**

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.sort(reverse = True)
print(demolist)
```

Example:

```
demolist = [100, 50, 65, 82, 23]
demolist.sort(reverse = True)
print(demolist)
```

Case Insensitive Sort

→ By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.sort()
print(demolist)
```

Note: So if you want a case-insensitive sort function, use str.lower as a key function.

Example:

Perform a case-insensitive sort of the list.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "thor", "Black Widow", "captain America"]
demolist.sort(key = str.lower)
print(demolist)
```

Reverse Order

→ What if you want to reverse the order of a list, regardless of the alphabet?

→ The reverse() method reverses the current sorting order of the elements.

Example:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
demolist.reverse()
print(demolist)
```

Copy a List

→ You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

Use the copy() method

→ You can use the built-in List method copy() to copy a list.

Example:

Make a copy of a list with the copy() method.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
mylist = demolist.copy()
print(mylist)
```

Use the list() method

→ Another way to make a copy is to use the built-in method list().

Example:

Make a copy of a list with the list() method:

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
mylist = list(demolist)
print(mylist)
```

Use the slice Operator

→ You can also make a copy of a list by using the **: (slice)** operator.

Example:

Make a copy of a list with the **:** operator.

```
demolist = ["Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America"]
mylist = demolist[:]
print(mylist)
```

Join Two Lists

→ There are several ways to join, or concatenate, two or more lists in Python.

→ One of the easiest ways are by using the **+ operator**.

Example:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
```

→ Another way to join two lists is by **appending** all the items from list2 into list1, one by one:

Example:

Append list2 into list1.

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

→ Or you can use the **extend()** method, where the purpose is to add elements from one list to another list:

Example:

Use the extend() method to add list2 at the end of list1.

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

List Built-In Functions summary:

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

List Related Programs:

1. Write a Python program to remove duplicate values from a list.

Example Code:

```
OGlist = [2, 4, 10, 20, 5, 2, 20, 4]
finallist = []
for num in OGlist:
        if num not in finallist:
                finallist.append(num)

print(finallist)
```

Output:

```
[2, 4, 10, 20, 5]
```

2. Write a Python program to find the transpose of the matrix using list.

Example Code:

```
l1 = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
l2 = []
for i in range(len(l1[0])):
    row =[]
    for item in l1:
        row.append(item[i])
    l2.append(row)

print(l2)
```

Output:

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

3. **Consider the list lst = [9,8,7,6,5,4,3]. Make the Python program which performs the following operation without using built-in methods.**
   i. Insert element 10 at beginning of the list.
   ii. Insert element 2 at end of the list.
   iii. Delete the element at index position 5.
   iv. Print all elements in reverse order.

Code:

```
lst = [9, 8, 7, 6, 5, 4, 3]

# 1. Insert element 10 at the beginning of the list
lst = [10] + lst
print(lst)

# 2. Insert element 2 at the end of the list
lst = lst + [2]
print(lst)

# 3. Delete the element at index position 5
# Create a new list excluding the element at index 5
lst = lst[:5] + lst[6:]
print(lst)

# 4. Print all elements in reverse order
for i in range(len(lst) - 1, -1, -1):
    print(lst[i])
```

Output:

```
[10, 9, 8, 7, 6, 5, 4, 3]
[10, 9, 8, 7, 6, 5, 4, 3, 2]
[10, 9, 8, 7, 6, 4, 3, 2]
2
3
4
6
7
8
9
10
```

4. **Construct a Python program to count the number of strings where the string length is 2 or more and the first and last character are same from a given list of strings. Sample List: ['abc', 'xyz', 'aba','1221'] Expected Result: 2**

Code:

```
sample_list = ['abc', 'xyz', 'aba', '1221','Mom','DaD']
count = 0
for s in sample_list:
   # Check if the string length is 2 or more
   if len(s) >= 2:
      # Check if the first and last character are the same
      if s[0] == s[-1]:
         count += 1
print("Number of strings where the first and last character are the same: ",count)
```

Output:

```
3
```

5. **Consider Str1, Str2 the two different string given as**

Str1 = 'Welcome to python programming'

Str2 ='hello how are you?'

Make a short note on results of the following expressions?

1.len(Str1)

2.Str1[-5]

3.Str1==Str2

4.Str2[5:6]

5.Str1[2:4:1]

Output:

```
29
m
False

lc
```

If you want to get elements at specified intervals, you can do it by using two [::].

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
print(my_list[1:10:3])
```

```
[2, 5, 8]
```

3.1 Python Tuples

→ Tuples are used to store multiple items in a single variable.
→ Tuple items are ordered, unchangeable, and allow duplicate values.
→ Tuples are written with () round brackets.

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a)
```

Output:

```
('Spiderman', 'John Wick', 'batman', 'Iron Man', 'Wonder Woman', 'Thor', 'Black Widow', 'Captain America', 'Loki', 'Superman', 'Deadpool')
```

★ Tuple Length

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(len(a))
```

Output:

```
11
```

★ Create Tuple with One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Code:

```
a=("Spiderman",)
print(type(a))

b=("Spiderman") #Not a tuple
print(type(b))
```

Output:

```
<class 'tuple'>
<class 'str'>
```

3.2 Working with Tuples in Python

★ Access Tuples

You can access tuple items by referring to the index number, inside square brackets.

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a[1])
```

Output:

```
John Wick
```

Note: The first item has index 0.

⋆ Negative Indexing

→ Negative indexing means start from the end.

→ -1 refers to the last item, -2 refers to the second last item

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a[-10])
```

Output:

```
John Wick
```

⋆ Range of Indexes

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a[3:7])
```

Output:

```
('Iron Man', 'Wonder Woman', 'Thor', 'Black Widow')
```

Note: The search will commence at the specified starting index (inclusive) and will terminate at the specified ending index (exclusive).

⋆ By leaving out the start value, the range will start at the first item

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a[:7])
```

Output:

```
('Spiderman', 'John Wick', 'batman', 'Iron Man', 'Wonder Woman', 'Thor', 'Black Widow')
```

⭐ By leaving out the end value, the range will go on to the end of the tuple

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a[3:])
```

Output:

```
('Iron Man', 'Wonder Woman', 'Thor', 'Black Widow', 'Captain America', 'Loki', 'Superman', 'Deadpool')
```

⭐ Range of Negative Indexes

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a[-3:-1])
```

Output:

```
('Loki', 'Superman')
```

★ Update Tuples

Tuples are immutable, which means you cannot modify, add, or remove elements once a tuple has been created.

→ You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
b=list(a)
b[1]="Aquaman"
a=tuple(b)
print(a)
```

Output:

```
('Spiderman', 'John Wick', 'batman', 'Iron Man', 'Aquaman', 'Thor', 'Black Widow', 'Captain America', 'Loki', 'Superman', 'Deadpool')
```

⭐ Add Items

Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple.

Convert into a list

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
b = list(a)
b.append("Aquaman")
a = tuple(b)
print(a)
```

Output:

```
('Spiderman', 'John Wick', 'batman', 'Iron Man', 'Wonder Woman', 'Thor', 'Black Widow', 'Captain America', 'Loki', 'Superman', 'Deadpool', 'Aquaman')
```

Add tuple to a tuple

Code:

```
a = ("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
b = ("Aquaman",)
a += b
print(a)
```

Output:

```
('Spiderman', 'John Wick', 'batman', 'Iron Man', 'Wonder Woman', 'Thor', 'Black Widow', 'Captain America', 'Loki', 'Superman', 'Deadpool', 'Aquaman')
```

⭐ Remove Items

Note: You cannot remove items in a tuple.

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
b = list(a)
b.remove("Wonder Woman")
a = tuple(b)
print(a)
```

Output:

```
('Spiderman', 'John Wick', 'batman', 'Iron Man', 'Thor', 'Black Widow', 'Captain America', 'Loki', 'Superman', 'Deadpool')
```

⭐ Delete the tuple completely

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
del a
print(a)
```

Output:

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

★ Tuple Packing and Unpacking

→ Packing: This refers to the process of combining multiple values into a single tuple.

→ Unpacking: This involves extracting the values from a tuple into individual variables.

→ This feature allows for convenient grouping and distribution of data in Python.

Code:

```
a=('With great power comes great responsibility.', 'Why so Serious?') # Packing values into a tuple
(Spiderman, batman)=b # Unpacking values from a tuple
print(Spiderman)
print(batman)
```

Output:

```
With great power comes great responsibility.
Why so Serious?
```

⭐ Using Asterisk*

→ If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list.

Code:

```
a=('With great power comes great responsibility.', 'Why so Serious?', 'I am vengeance, I am the night.', 'I can do this all day.')
(Spiderman, *batman)=a
print(Spiderman)
print(batman)
```

Output:

```
With great power comes great responsibility.
['Why so Serious?', 'I am vengeance, I am the night.', 'I can do this all day.']
```

→ If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Code:

```
a=('With great power comes great responsibility.', 'Why so Serious?', 'I am vengeance, I am the night.', 'I can do this all day.')
(Spiderman, *batman, other_quotes)=a
print(Spiderman)
print(batman)
```

Output:

```
With great power comes great responsibility.
['Why so Serious?', 'I am vengeance, I am the night.']
I can do this all day.
```

★ Loop Tuples

Code:

```
a=("Spiderman", "John Wick", "batman", "Loki")
for i in a:
  print(i)
```

Output:

```
Spiderman
John Wick
batman
Loki
```

Code:

```
a=("Spiderman", "John Wick", "batman", "Loki")
for i in range(len(a)):
  print(a[i])
```

Output:

```
Spiderman
John Wick
batman
Loki
```

★ Join Tuples

Code:

```
a=("Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman")
b=("Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
c=a+b
print(c)
```

Output:

```
('Spiderman', 'John Wick', 'batman', 'Iron Man', 'Wonder Woman', 'Thor', 'Black Widow', 'Captain America', 'Loki', 'Superman', 'Deadpool')
```

Code:

```
a=("Spiderman", "John Wick", "Loki")
b=a*3
print(b)
```

Output:

```
('Spiderman', 'John Wick', 'Loki', 'Spiderman', 'John Wick', 'Loki', 'Spiderman', 'John Wick', 'Loki')
```

★ zip() Function

→ The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.

→ If the passed iterables have different lengths, the iterable with the least items decides the length of the new iterator.

Code:

```
a=('With great power comes great responsibility.', 'Why so Serious?')
b=('Spiderman', 'batman')
c=zip(a,b)
print(tuple(c))
```

Output:

```
(('With great power comes great responsibility.', 'Spiderman'), ('Why so Serious?', 'batman'))
```

Code:

```
a=('With great power comes great responsibility.', 'Why so Serious?')
b=('Spiderman', 'batman', 'Krish')
c=zip(a,b)
print(tuple(c))
```

Output:

```
(('With great power comes great responsibility.', 'Spiderman'), ('Why so Serious?', 'batman'))
```

★ Tuple Methods

⭐ index() Method

→ The index() method finds the first occurrence of the specified value.

→ The index() method raises an exception if the value is not found.

Code:

```
a=("Spiderman", "John Wick", "Loki", "batman", "Iron Man", "Loki", "Wonder Woman", "Thor", "Loki", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a.index("Loki"))
```

Output:

```
2
```

⭐ count() Method

Code:

```
a=("Spiderman", "John Wick", "Loki", "batman", "Iron Man", "Loki", "Wonder Woman", "Thor", "Loki", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool")
print(a.count("Loki"))
```

Output:

```
4
```

3.3 Python Sets

Set items are unordered, unchangeable, and do not allow duplicate values.

Note: Once a set is created, you cannot change its items, but you can remove items and add new items.

Code:

```
a={"Spiderman", "John Wick", "Loki", "batman", "Iron Man", "Loki", "Wonder Woman", "Thor", "Loki", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool"}
print(a)
print(type(a))
```

Output:

```
{'Iron Man', 'Deadpool', 'Captain America', 'Wonder Woman', 'John Wick', 'Loki', 'Spiderman', 'Superman', 'batman', 'Thor', 'Black Widow'}
<class 'set'>
```

Note: In sets, the values True and 1 are treated as duplicates, as are False and 0.

Code:

```
a={"Spiderman", "John Wick", "Loki", True, 1, 2, False, 0}
print(a)
```

Output:

```
{False, True, 2, 'Loki', 'Spiderman', 'John Wick'}
```

⭐Length of a Set

Code:

```
a={"Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool"}
print(len(a))
```

Output:

```
11
```

Code:

```
a={"Spiderman", "John Wick", "batman", "Iron Man", "Wonder Woman", "Thor", "Black Widow", "Captain America", "Loki", "Superman", "Deadpool"}
print('Spiderman' in a)
print('Spider' not in a)
```

Output:

```
True
True
```

3.4 Working with Set in Python

★ Add Set Items

Code:

```
a={"Spiderman", "John Wick"}
a.add("Loki")
print(a)
```

Output:

```
{'John Wick', 'Loki', 'Spiderman'}
```

Code:

```
a={"Spiderman", "John Wick"}
b={"Superman", "batman", "Thor"}
a.update(b)
print(a)
```

Output:

```
{'batman', 'Spiderman', 'Thor', 'John Wick', 'Superman'}
```

Note: update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

Code:

```
a={"Spiderman", "John Wick"}
b=["Superman", "batman", "Thor"]
a.update(b)
print(a)
```

Output:

```
{'batman', 'Spiderman', 'John Wick', 'Thor', 'Superman'}
```

★ Remove Set Items

Code:

```
a={"Spiderman", "John Wick", "Loki", "Deadpool"}
a.remove('Deadpool')
print(a)
```

Output:

```
{'Loki', 'Spiderman', 'John Wick'}
```

Note: If the item to remove does not exist, remove() will raise an error.

Code:

```
a={"Spiderman", "John Wick", "Loki", "Deadpool"}
a.discard('Deadpool')
print(a)
```

Output:

```
{'Spiderman', 'Loki', 'John Wick'}
```

Note: If the item to remove does not exist, discard() will NOT raise an error.

Code:

```
a={"Spiderman", "John Wick", "Loki", "Deadpool"}
b = a.pop()
print(b)
```

Output:

```
Deadpool
```

Note: Sets are unordered, so when using the pop() method, you do not know which item that gets removed.

Code:

```
a={"Spiderman", "John Wick", "Loki", "Deadpool"}
a.clear()
print(a)
```

Output:

```
set()
```

The del keyword will delete the set completely.

Code:

```
a={"Spiderman", "John Wick", "Loki", "Deadpool"}
del a
print(a)
```

Output:

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

★ Loop Sets

Code:

```
a={"Spiderman", "John Wick", "Loki", "Deadpool"}
for i in a:
    print(i)
```

Output:

```
Deadpool
John Wick
Loki
Spiderman
```

★ Join Sets

⭐ Union

Code:

```
a={"Spiderman", "John Wick"}
b={"Loki", "Deadpool"}
c=a.union(b)
print(c)
```

Output:

```
{'Loki', 'Spiderman', 'Deadpool', 'John Wick'}
```

Code:

```
a={"Spiderman", "John Wick"}
b={"Loki", "Deadpool"}
c=a|b
print(c)
```

Output:

```
{'John Wick', 'Spiderman', 'Loki', 'Deadpool'}
```

⭐ Join Multiple Sets

Code:

```
a={"Spiderman", "John Wick"}
b={"Loki", "Deadpool"}
c={"Krish"}
d={"Ra.one"}
e=a.union(b,c,d)
print(e)
```

Output:

```
{'Ra.one', 'Loki', 'Deadpool', 'Spiderman', 'John Wick', 'Krish'}
```

Code:

```
a={"Spiderman", "John Wick"}
b={"Loki", "Deadpool"}
c={"Krish"}
d={"Ra.one"}
e= a|b|c|d
print(e)
```

Output:

```
{'Krish', 'Deadpool', 'Loki', 'Ra.one', 'John Wick', 'Spiderman'}
```

⭐ Join a Set and a Tuple

Code:

```
a={"Spiderman", "John Wick"}
b=("Loki", "Deadpool")
c=a.union(b)
print(c)
```

Output:

```
{'Spiderman', 'Deadpool', 'John Wick', 'Loki'}
```

Note: The | operator only allows you to join sets with sets, and not with other data types.

⭐ Update

→ The update() method inserts all items from one set into another.

→ The update() changes the original set, and does not return a new set.

Code:

```
a={"Spiderman", "John Wick"}
b={"Loki", "Deadpool"}
a.update(b)
print(a)
```

Output:

```
{'John Wick', 'Loki', 'Deadpool', 'Spiderman'}
```

⋆ Intersection

→ The intersection() method will return a new set, that only contains the items that are present in both sets.

Code:

```
a={"Spiderman", "John Wick","Loki"}
b={"Loki", "Deadpool"}
c = a.intersection(b)
print(c)
```

Output:

```
{'Loki'}
```

Code:

```
a={"Spiderman", "John Wick","Loki"}
b={"Loki", "Deadpool"}
c = a&b
print(c)
```

Output:

```
{'Loki'}
```

Note: The & operator only allows you to join sets with sets, and not with other data types.

⋆ intersection with update

→ The intersection_update() method will also keep ONLY the duplicates, but it will change the original set instead of returning a new set.

Code:

```
a={"Spiderman", "John Wick","Loki"}
b={"Loki", "Deadpool"}
a.intersection_update(b)
print(a)
```

Output:

```
{'Loki'}
```

→ In Python, the values True and 1 are considered the same, as are False and 0.

Code:

```
a = {"Spiderman", "John Wick", "Loki", True, 0}
b = {"Loki", "Deadpool", False, 0, 1}
c = a.intersection(b)
print(c)
```

Output:

```
{False, 1, 'Loki'}
```

⋆ Difference

→ The difference() method creates a new set that includes only the elements from the first set that are not found in the second set.

Code:

```
a = {"Spiderman", "John Wick", "Loki", True, 0}
b = {"Loki", "Deadpool", False, 0, 1}
c = a.difference(b)
print(c)
```

Output:

```
{'John Wick', 'Spiderman'}
```

Code:

```
a = {"Spiderman", "John Wick", "Loki", True, 0}
b = {"Loki", "Deadpool", False, 0, 1}
c = a-b
print(c)
```

Output:

```
{'John Wick', 'Spiderman'}
```

Note: The - operator only allows you to join sets with sets, and not with other data types like you can with the difference() method.

⋆ difference with update

→ The difference_update() method modifies the original set by removing elements that are present in the second set, retaining only those items that are unique to the first set.

Code:

```
a = {"Spiderman", "John Wick", "Loki", True, 0}
b = {"Loki", "Deadpool", False, 0, 1}
a.difference_update(b)
print(a)
```

Output:

```
{'Spiderman', 'John Wick'}
```

⋆ Symmetric Differences

→ The symmetric_difference() method retains only the elements that are unique to each set, excluding those that are present in both.

Code:

```
a = {"Spiderman", "John Wick", "Loki", True, 0}
b = {"Loki", "Deadpool", False, 0, 1}
c=a.symmetric_difference(b)
print(c)
```

Output:

```
{'Deadpool', 'John Wick', 'Spiderman'}
```

Code:

```
a = {"Spiderman", "John Wick", "Loki", True, 0}
b = {"Loki", "Deadpool", False, 0, 1}
c=a^b
print(c)
```

Output:

```
{'Deadpool', 'John Wick', 'Spiderman'}
```

Note: The ^ operator is used exclusively to combine sets; it cannot be applied to other data types as the symmetric_difference() method can.

⋆ Symmetric difference with update

→ The symmetric_difference_update() method modifies the original set by retaining only the unique elements, excluding any duplicates, instead of creating a new set.

Code:

```
a = {"Spiderman", "John Wick", "Loki", True, 0}
b = {"Loki", "Deadpool", False, 0, 1}
a.symmetric_difference_update(b)
print(a)
```

Output:

```
{'Spiderman', 'John Wick', 'Deadpool'}
```

★ **Sets Methods**

Method	Shortcut	Description
add()		Adds an element to the set
clear()		Removes all the elements from the set
copy()		Returns a copy of the set
difference()	-	Returns a set containing the difference between two or more sets
difference_update()	-=	Removes the items in this set that are also included in another, specified set
discard()		Remove the specified item
intersection()	&	Returns a set, that is the intersection of two other sets
intersection_update()	&=	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()		Returns whether two sets have a intersection or not
issubset()	<=	Returns whether another set contains this set or not
	<	Returns whether all items in this set is present in other, specified set(s)
issuperset()	>=	Returns whether this set contains another set or not
	>	Returns whether all items in other, specified set(s) is present in this set
pop()		Removes an element from the set
remove()		Removes the specified element
symmetric_difference()	^	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	^=	Inserts the symmetric differences from this set and another
union()	|	Return a set containing the union of sets
update()	|=	Update the set with the union of this set and others

3.5 Python Dictionaries

- → Dictionaries are used to store data values in key-value pairs.
- → A dictionary is a collection that is ordered, changeable, and does not allow duplicates.
- → As of Python version 3.7, dictionaries maintain the order of insertion. In Python 3.6 and earlier, dictionaries are unordered.
- → Duplicates Not Allowed: Dictionaries cannot contain multiple items with the same key.

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'release_year': 2014, 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4}
```

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }
print(type(john_wick_info))
print(len(john_wick_info))
print(john_wick_info["portrayed_by"])
```

Output:

```
<class 'dict'>
4
Keanu Reeves
```

3.6 Working with Dictionaries in Python

★ Access Items

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }
print(john_wick_info["portrayed_by"])
```

Output:

```
Keanu Reeves
```

⋆ get()

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

a=john_wick_info.get("portrayed_by")
print(a)
```

Output:

```
Keanu Reeves
```

⋆ Get Keys

→ The keys() method will return a list of all the keys in the dictionary.

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

a=john_wick_info.keys()
print(a)
```

Output:

```
dict_keys(['title', 'release_year', 'portrayed_by', 'number_of_films'])
```

Code:

```
john_wick_info = {
    "title": "John Wick",
    "release_year": 2014,
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4
    }

a=john_wick_info.keys()
print(a)
john_wick_info["car"]="Mustang"
print(a)
```

Output:

```
dict_keys(['title', 'release_year', 'portrayed_by', 'number_of_films'])
dict_keys(['title', 'release_year', 'portrayed_by', 'number_of_films', 'car'])
```

⭐Get Values

Code:

```
john_wick_info = {
    "title": "John Wick",
    "release_year": 2014,
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4
    }

a=john_wick_info.values()
print(a)
```

Output:

```
dict_values(['John Wick', 2014, 'Keanu Reeves', 4])
```

Code:

```
john_wick_info = {
    "title": "John Wick",
    "release_year": 2014,
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4
    }

a=john_wick_info.values()
print(a)
john_wick_info["number_of_films"]=5
print(a)
```

Output:

```
dict_values(['John Wick', 2014, 'Keanu Reeves', 4])
dict_values(['John Wick', 2014, 'Keanu Reeves', 5])
```

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

a=john_wick_info.values()
print(a)
john_wick_info["car"]="Mustang"
print(a)
```

Output:

```
dict_values(['John Wick', 2014, 'Keanu Reeves', 4])
dict_values(['John Wick', 2014, 'Keanu Reeves', 4, 'Mustang'])
```

⋆ Get Items

→ The items() method will return each item in a dictionary, as tuples in a list.

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

a=john_wick_info.items()
print(a)
```

Output:

```
dict_items([('title', 'John Wick'), ('release_year', 2014), ('portrayed_by', 'Keanu Reeves'), ('number_of_films', 4)])
```

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

a=john_wick_info.items()
print(a)
john_wick_info["number_of_films"]=5
print(a)
```

Output:

```
dict_items([('title', 'John Wick'), ('release_year', 2014), ('portrayed_by', 'Keanu Reeves'), ('number_of_films', 4)])
dict_items([('title', 'John Wick'), ('release_year', 2014), ('portrayed_by', 'Keanu Reeves'), ('number_of_films', 5)])
```

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

a=john_wick_info.items()
print(a)
john_wick_info["car"]="Mustang"
print(a)
```

Output:

```
dict_items([('title', 'John Wick'), ('release_year', 2014), ('portrayed_by', 'Keanu Reeves'), ('number_of_films', 4)])
dict_items([('title', 'John Wick'), ('release_year', 2014), ('portrayed_by', 'Keanu Reeves'), ('number_of_films', 4), ('car', 'Mustang')])
```

⭐Check if Key Exists

Code:

```
john_wick_info= {
   "title": "John Wick",
   "release_year": 2014,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

if "car" in john_wick_info:
   print('Yes')
else:
   print('No')
```

Output:

```
No
```

★ Change Items

Code:

```
john_wick_info = {
    "title": "John Wick",
    "release_year": 2013,
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4
    }

john_wick_info["release_year"]=2014
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'release_year': 2014, 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4}
```

⭑Update Dictionary

Code:

```
john_wick_info = {
    "title": "John Wick",
    "release_year": 2013,
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4
    }

john_wick_info.update({"release_year":2014})
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'release_year': 2014, 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4}
```

★ Add Items

Code:

```
john_wick_info = {
    "title": "John Wick",
    "release_year": 2013,
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4
    }

john_wick_info["car"]="Mustang"
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'release_year': 2013, 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4, 'car': 'Mustang'}
```

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2013,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4
   }

john_wick_info.update({"car":"Mustang"})
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'release_year': 2013, 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4, 'car': 'Mustang'}
```

★ Remove Items

⭐ pop() method

Code:

```
john_wick_info = {
   "title": "John Wick",
   "release_year": 2013,
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang"
   }

john_wick_info.pop("number_of_films")
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'release_year': 2013, 'portrayed_by': 'Keanu Reeves', 'car': 'Mustang'}
```

⭐ popitem() method

→ The popitem() method removes the last inserted item. In versions prior to 3.7, it removes a random item instead.

Code:

```
john_wick_info = {
   "title": "John Wick",
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang",
   "release_year": 2013
   }

john_wick_info.popitem()
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4, 'car': 'Mustang'}
```

⋆ del

→ The del keyword removes the item associated with the specified key.

→ The del keyword can also be used to delete the entire dictionary.

Code:

```
john_wick_info = {
   "title": "John Wick",
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang",
   "release_year": 2013
   }

del john_wick_info["release_year"]
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4, 'car': 'Mustang'}
```

Code:

```
john_wick_info = {
   "title": "John Wick",
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang",
   "release_year": 2013
   }

del john_wick_info
print(john_wick_info) #this will cause an error because "john_wick_info" no longer exists.
```

Output:

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

⭐ clear() method

→ The clear() method empties the dictionary

Code:

```
john_wick_info = {
   "title": "John Wick",
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang",
   "release_year": 2013
   }

john_wick_info.clear()
print(john_wick_info)
```

Output:

```
{}
```

★ Loop Dictionaries

→ When looping through a dictionary, the return values are the keys. However, there are methods available to retrieve the values as well.

⭐ Print all key names in the dictionary

Code:

```
john_wick_info = {
   "title": "John Wick",
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang",
   "release_year": 2013
   }

for i in john_wick_info:
   print(i)
```

Output:

```
title
portrayed_by
number_of_films
car
release_year
```

⭐ Print all values in the dictionary

Code:

```
john_wick_info = {
    "title": "John Wick",
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4,
    "car": "Mustang",
    "release_year": 2013
    }

for i in john_wick_info:
    print(john_wick_info[i])
```

Output:

```
John Wick
Keanu Reeves
4
Mustang
2013
```

⭐ values() method

Code:

```
john_wick_info = {
    "title": "John Wick",
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4,
    "car": "Mustang",
    "release_year": 2013
    }

for i in john_wick_info.values():
    print(i)
```

Output:

```
John Wick
Keanu Reeves
4
Mustang
2013
```

⭐keys() method

Code:

```
john_wick_info = {
    "title": "John Wick",
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4,
    "car": "Mustang",
    "release_year": 2013
    }

for i in john_wick_info.keys():
    print(i)
```

Output:

```
title
portrayed_by
number_of_films
car
release_year
```

⭐items() method

Code:

```
john_wick_info = {
    "title": "John Wick",
    "portrayed_by": "Keanu Reeves",
    "number_of_films": 4,
    "car": "Mustang",
    "release_year": 2013
    }

for i,j in john_wick_info.items():
    print(i,'→',j)
```

Output:

```
title → John Wick
portrayed_by → Keanu Reeves
number_of_films → 4
car → Mustang
release_year → 2013
```

★ Copy Dictionaries

→ To copy a dictionary, you can't just use dict2 = dict1, because this will only create a reference to dict1, meaning that any changes made to dict1 will also affect dict2.

⭐ copy() method

One method to create an actual copy is to use the built-in copy() method of the dictionary.

Code:

```
john_wick_info = {
   "title": "John Wick",
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang",
   "release_year": 2013
   }

a=john_wick_info.copy()
print(a)
```

Output:

```
{'title': 'John Wick', 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4, 'car': 'Mustang', 'release_year': 2013}
```

⭐ dict() method

Code:

```
john_wick_info = {
   "title": "John Wick",
   "portrayed_by": "Keanu Reeves",
   "number_of_films": 4,
   "car": "Mustang",
   "release_year": 2013
   }

a=dict(john_wick_info)
print(a)
```

Output:

```
{'title': 'John Wick', 'portrayed_by': 'Keanu Reeves', 'number_of_films': 4, 'car': 'Mustang', 'release_year': 2013}
```

Note: dict(): Used to create a new dictionary from scratch or to copy an existing dictionary. copy(): Specifically used to create a shallow copy of an existing dictionary.

Code:

```
a = dict(a=1, b=2, c=3)
print(a)
```

Output:

```
{'a': 1, 'b': 2, 'c': 3}
```

★ Nested Dictionaries

→ Nested dictionaries are dictionaries that contain other dictionaries as their values. This allows for more complex data structures, enabling you to organize data hierarchically.

Inline Declaration

Code:

```
students = {
   'student1': {
      'name': 'John Wick',
      'age': 20,
      'dob': '15/05/2005',
      'enrollment_no': 'EN12345',
      'mobile_number': '1234567890'
   },
   'student2': {
      'name': 'Spiderman',
      'age': 12,
      'dob': '21/12/2012',
      'enrollment_no': 'EN12346',
      'mobile_number': '0987654321'
   },
   'student3': {
      'name': 'Batman',
      'age': 13,
      'dob': '11/11/2011',
      'enrollment_no': 'EN12347',
      'mobile_number': '5551234567'
   }
}
print(students)
```

Output:

```
{'student1': {'name': 'John Wick', 'age': 20, 'dob': '15/05/2005', 'enrollment_no': 'EN12345', 'mobile_number': '1234567890'}, 'student2': {'name': 'Spiderman', 'age': 12, 'dob': '21/12/2012', 'enrollment_no': 'EN12346', 'mobile_number': '0987654321'}, 'student3': {'name': 'Batman', 'age': 13, 'dob': '11/11/2011', 'enrollment_no': 'EN12347', 'mobile_number': '5551234567'}}
```

Declare Sub-Dictionaries First: This approach involves defining each dictionary individually and then combining them into a nested structure.

Code:

```
student1 = {
   'name': 'John Wick',
   'age': 20,
   'dob': '15/05/2005',
   'enrollment_no': 'EN12345',
   'mobile_number': '1234567890'
}

student2 = {
   'name': 'Spiderman',
   'age': 12,
   'dob': '21/12/2012',
   'enrollment_no': 'EN12346',
   'mobile_number': '0987654321'
}

student3 = {
   'name': 'Batman',
   'age': 13,
   'dob': '11/11/2011',
   'enrollment_no': 'EN12347',
   'mobile_number': '5551234567'
}

students = {
   'student1': student1,
   'student2': student2,
   'student3': student3
}

print(students)
```

Output:

```
{'student1': {'name': 'John Wick', 'age': 20, 'dob': '15/05/2005', 'enrollment_no': 'EN12345', 'mobile_number': '1234567890'}, 'student2': {'name': 'Spiderman', 'age': 12, 'dob': '21/12/2012', 'enrollment_no': 'EN12346', 'mobile_number': '0987654321'}, 'student3': {'name': 'Batman', 'age': 13, 'dob': '11/11/2011', 'enrollment_no': 'EN12347', 'mobile_number': '5551234567'}}
```

Access Items in Nested Dictionaries

Code:

```
students = {
   'student1': {
      'name': 'John Wick',
      'age': 20,
      'dob': '15/05/2005',
      'enrollment_no': 'EN12345',
      'mobile_number': '1234567890'
   },
   'student2': {
      'name': 'Spiderman',
      'age': 12,
      'dob': '21/12/2012',
      'enrollment_no': 'EN12346',
      'mobile_number': '0987654321'
   },
   'student3': {
      'name': 'Batman',
      'age': 13,
      'dob': '11/11/2011',
      'enrollment_no': 'EN12347',
      'mobile_number': '5551234567'
   }
}
print(students['student1']['name'])
```

Output:

```
John Wick
```

Loop Through Nested Dictionaries

Code:

```
students = {
   'student1': {
      'name': 'John Wick',
      'age': 20,
      'dob': '15/05/2005',
      'enrollment_no': 'EN12345',
      'mobile_number': '1234567890'
   },
   'student2': {
      'name': 'Spiderman',
      'age': 12,
      'dob': '21/12/2012',
      'enrollment_no': 'EN12346',
      'mobile_number': '0987654321'
   },
   'student3': {
      'name': 'Batman',
      'age': 13,
      'dob': '11/11/2011',
      'enrollment_no': 'EN12347',
      'mobile_number': '5551234567'
   }
}
for i, dic in students.items():
   print(i)

   for j in dic:
      print(j + ':', dic[j])
```

Output:

```
student1
name: John Wick
age: 20
dob: 15/05/2005
enrollment_no: EN12345
mobile_number: 1234567890
student2
name: Spiderman
age: 12
dob: 21/12/2012
enrollment_no: EN12346
mobile_number: 0987654321
student3
name: Batman
age: 13
dob: 11/11/2011
enrollment_no: EN12347
mobile_number: 5551234567
```

Generalized Example of Nesting

Code:

```
john_wick_info = {
    "title": "John Wick",
    "release_year": 2014,
    "director": "Chad Stahelski",
    "main_character": {
        "name": "John Wick",
        "portrayed_by": "Keanu Reeves",
        "background": "Retired hitman seeking vengeance for the death of his dog."
    },
    "key_elements": [
        "Action",
        "Vengeance",
        "Assassins",
        "Loyalty"
    ],
    "franchise": {
        "number_of_films": 4,
        "films": [
            "John Wick (2014)",
            "John Wick: Chapter 2 (2017)",
            "John Wick: Chapter 3 – Parabellum (2019)",
            "John Wick: Chapter 4 (2023)"
        ]
    },
    "notable_characters": {
        "Iosef Tarasov": "Antagonist in the first film",
        "Viggo Tarasov": "Iosef's father and crime lord",
        "Winston": "Owner of the Continental Hotel",
        "Sofia": "John's ally, portrayed by Halle Berry"
    }
}
print(john_wick_info)
```

Output:

```
{'title': 'John Wick', 'release_year': 2014, 'director': 'Chad Stahelski', 'main_character': {'name': 'John Wick', 'portrayed_by': 'Keanu Reeves', 'background': 'Retired hitman seeking vengeance for the death of his dog.'}, 'key_elements': ['Action', 'Vengeance', 'Assassins', 'Loyalty'], 'franchise': {'number_of_films': 4, 'films': ['John Wick (2014)', 'John Wick: Chapter 2 (2017)', 'John Wick: Chapter 3 – Parabellum (2019)', 'John Wick: Chapter 4 (2023)']}, 'notable_characters': {'Iosef Tarasov': 'Antagonist in the first film', 'Viggo Tarasov': "Iosef's father and crime lord", 'Winston': 'Owner of the Continental Hotel', 'Sofia': "John's ally, portrayed by Halle Berry"}}
```

★ Merging two dictionaries

Code:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}

merged_dict = dict1.copy()
merged_dict.update(dict2)
print(merged_dict)
```

Output:

```
{'a': 1, 'b': 3, 'c': 4}
```

⭐ merge using | operator

Code:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}

merged_dict = dict1 | dict2
print(merged_dict)
```

Output:

```
{'a': 1, 'b': 3, 'c': 4}
```

★ Dictionary Methods

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

3.7 Comparative Analysis of Python's Data Structures: Lists, Tuples, Sets, and Dictionaries

List	Tuple	Set	Dictionary
List is a non-homogeneous data structure that stores the elements in single row and multiple rows and columns	Tuple is also a non-homogeneous data structure that stores single row and multiple rows and columns	Set data structure is also non-homogeneous data structure but stores in single row	Dictionary is also a non-homogeneous data structure which stores key value pairs
List can be represented by [ ]	Tuple can be represented by ( )	Set can be represented by { }	Dictionary can be represented by { }
List allows duplicate elements	Tuple allows duplicate elements	Set will not allow duplicate elements	Set will not allow duplicate elements and dictionary doesn't allow duplicate keys.
List can use nested among all	Tuple can use nested among all	Set can use nested among all	Dictionary can use nested among all
Example: [1, 2, 3, 4, 5]	Example: (1, 2, 3, 4, 5)	Example: {1, 2, 3, 4, 5}	Example: {1, 2, 3, 4, 5}
List can be created using list() function	Tuple can be created using tuple() function.	Set can be created using set() function	Dictionary can be created using dict() function.
List is mutable i.e we can make any changes in list.	Tuple is immutable i.e we can't make any changes in tuple	Set is mutable i.e we can make any changes in set. But elements are not duplicated.	Dictionary is mutable. But Keys are not duplicated.
List is ordered	Tuple is ordered	Set is unordered	Dictionary is ordered (Python 3.7 and above)
Creating an empty list a=[]	Creating an empty Tuple a=()	If you create an empty set using a = {}, you'll actually create an empty dictionary instead of an empty set. To create an empty set, you should use a = set().	Creating an empty dictionary a={}

4.1 Conditional Statements

Conditional statements allow a program to make decisions and execute certain parts of code based on whether a condition is true or false. In Python, the most common conditional statements are if, elif, and else. They are used to control the flow of execution depending on certain conditions.

1. if Statements

Example:

```
a = 10
b = 20
if b > a:
  print("b is greater than a")
```

Output:

```
b is greater than a
```

2. if-else Statements

Example:

```
a = 20
b = 10
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Output:

```
b is not greater than a
```

3. elif Statements

Example:

```
number = int(input('Enter a number: '))

# Check if the number is negative, zero, or positive
if number < 0:
    print("The number is negative.")
elif number == 0:
    print("The number is zero.")
else:
    print("The number is positive.")
```

Output:

```
Enter a number: 10
The number is positive.
```

4. Nested if Statements

Example:

```
number = 100

# Check if the number is positive
if number > 0:
    print("The number is positive.")
    # Nested if to check if the number is greater than 10
    if number > 10:
        print("The number is greater than 10.")
    else:
        print("The number is 10 or less.")
else:
    print("The number is not positive.")
```

Output:

```
The number is positive.

The number is greater than 10.
```

4.2 Loop Statements

Loop statements are used to repeatedly execute a block of code as long as a certain condition is met. In Python, the primary loop constructs are for and while loops. They are essential for performing repetitive tasks efficiently.

→ **For Loop:**

1. Looping Through a List

Example:

```
branch = ["CSE AIML", "CSE", "IT"]
for x in branch:
  print(x)
```

Output:

```
CSE AIML
CSE
IT
```

2. Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example:

```
for x in 'CSE AIML':
  print(x)
```

Output:

```
C
S
E

A
I
M
L
```

3. The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

i. *The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.*

Example:

```
for x in range(11):
  print(x)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

ii. *The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(1, 10), which means values from 1 to 11 (but not including 11):*

Example:

```
for x in range(1,11):
  print(x)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

iii. *The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(1, 11,2):*

Example:

```
for x in range(1,11,2):
  print(x)
```

Output:

```
1
3
5
7
9
```

iv. *The range() function defaults to incrementing the sequence by 1. To specify a decrement, you can use a negative step value as the third parameter, for example: range(10, 0, -2).*

This will generate a sequence starting at 10 and ending before 0, with a decrement of 2.

Example:

```
for x in range(10,0,-2):
  print(x)
```

Output:

```
10
8
6
4
2
```

4. Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example:
Print all numbers from 1 to 10, and print a message when the loop has ended.

```
for x in range(1,11):
  print(x)
else:
  print("The numbers from 1 to 10 were printed.")
```

Output:

```
1
2
3
4
5
6
7
8
9
10
The numbers from 1 to 10 were printed.
```

Note: The else block will NOT be executed if the loop is stopped by a break statement.

→ **While Loop:**

With the while loop we can execute a set of statements as long as a condition is true.

1. Basics of while loop:

Example:
Print i as long as i is less than 11:

```
i = 1
while i < 11:
  print(i)
  i += 1
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Note: Remember to increment i, or else the loop will continue forever to the ∞ stage.

2. else in While Loop

With the else statement we can run a block of code once when the condition no longer is true.

Example:
Print a message once the condition is false.

```
i = 1
while i < 5:
    print(i)
    i += 1
else:
    print("The condition is now false. The loop has finished.")
```

Output:

```
1
2
3
4
The condition is now false. The loop has finished.
```

4.3 Jump Statements

Jump statements are used to alter the normal flow of control in a program. In Python, break, continue, and pass are common jump statements. break exits a loop, continue skips the rest of the code inside the loop for the current iteration and moves to the next iteration, and pass is a null operation that serves as a placeholder.

Types of Jump Statements:

1. Break
2. Continue
3. Pass

1. Break Statement

A break statement is used to break or stop a flow control. This is generally used in a loop. A break statement is used in a loop in such a way, that when a particular condition becomes true, the break statement is executed and we come out of the loop immediately at that moment.

Syntax:

```
Loop{
   Condition:
      break
   }
```

Example:

```
for i in range(1, 5):
   if(i == 3):
      break
   print(i)
```

Output:

```
1
2
```

2. Continue Statement

The continue statement is somewhat similar to the break statement but there is one significant difference between them. A break statement terminates the entire loop but the continue statement skips only the current iteration and continues with the rest steps. So, if we use the continue statement in a loop, it will only skip one iteration when the associated condition is true and then continue the rest of the loop unlike the break statement.

Syntax:

```
while Condition:
    ...
    Condition:
        continue
```

Example:

```
for i in range(1, 5):
    if(i == 2):
        continue
    print(i)
```

Output:

```
1
3
4
```

3. Pass Statement

The pass statement is an interesting statement, as when it gets executed nothing happens. It plays the role of a placeholder. This means that, if we want to write a piece of code over there in the near future but we cannot keep that space empty, then we can use the pass statement. In simple words, it is used to avoid getting an error from keeping an empty space.

Example:

```
for i in range(10):
    pass  # TODO: Add logic here later
```

4.4 List of logical programs in Python

Note: You can use your own logic to solve the logical programming problems, but make sure that the output matches the provided solutions exactly. Test your code to verify that it produces the correct results.

[1] Fibonacci Series

→ Fibonacci series is a special type of series in which the next term is the sum of the previous two terms. For example, if 0 and 1 are the two previous terms in a series, then the next term will be 1(0+1).

Code:

```
# Initialization
number1, number2 = 0, 1
count = 5  # Number of Fibonacci numbers to generate

# Printing the first two numbers
print(number1, number2, end=' ')

# Generating and printing the rest of the Fibonacci sequence
for i in range(2, count):
    number3 = number1 + number2
    print(number3, end=' ')
    number1, number2 = number2, number3
```

Output:

```
0 1 1 2 3
```

[2] Armstrong Number

→ An Armstrong Number is a special positive number whose sum of cubes of its digit is equal to that number. The number 153 is an Armstrong number because if we perform the sum of cubes of each digit, it will result in the same number.

Example:

```
= 153
= (1)^3 + (5)^3 + (3)^3
= 1+125+27
= 153
```

Code:

```
num = int(input("Enter a number: "))

sum = 0

# find the sum of the cube of each digit
temp = num
while temp > 0:
   digit = temp % 10
   sum += digit ** 3
   temp //= 10
if num == sum:
   print(num,"is an Armstrong number")
else:
   print(num,"is not an Armstrong number")
```

Output:

```
Enter a number: 153
153 is an Armstrong number
```

[3] Perfect number

→ The Perfect Number is also a special type of positive number. When the number is equal to the sum of its positive divisors excluding the number, it is called a Perfect Number. For example, 28 is the perfect number because when we sum the divisors of 28, it will result in the same number. The divisors of 28 are 1, 2, 4, 7, and 14. So,

Example:

```
28 = 1+2+4+7
28 = 28
```

Code:

```
Input_Number = 28
Sum = 0
for i in range(1, Input_Number):
   if(Input_Number % i == 0):
      Sum = Sum + i
if (Sum == Input_Number):
   print('Number is a Perfect Number.')
else:
   print('Number is not a Perfect Number.')
```

Output:

```
Number is a Perfect Number.
```

[4] Prime number

→ Prime number is also a special type of number. When the number is divided greater than 1 and divided by 1 or itself is referred to as the Prime number. 0 and 1 are not counted as prime numbers. All the even numbers can be divided by 2, so 2 is the only even prime number.

Example:

```
43
```

Code:

```
num = 43
# Negative numbers, 0 and 1 are not primes
if num > 1:

   # Iterate from 2 to n // 2
   for i in range(2, (num//2)+1):

      # If num is divisible by any number between
      # 2 and n / 2, it is not prime
      if (num % i) == 0:
         print(num, "is not a prime number")
         break
   else:
      print(num, "is a prime number")
else:
   print(num, "is not a prime number")
```

Output:

```
43 is a prime number
```

Code 2:

```
# Get user input for the range
start = int(input("Enter the start of the range: "))
end = int(input("Enter the end of the range: "))

# Iterate through each number in the range
for num in range(start, end + 1):
   if num > 1:
      # Assume the number is prime
      for i in range(2, (num // 2) + 1):
         if num % i == 0:
            break
      else:
         print(num, "is a prime number")
```

Output2:

```
Enter the start of the range: 1
Enter the end of the range: 43
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
23 is a prime number
29 is a prime number
31 is a prime number
37 is a prime number
41 is a prime number
43 is a prime number
```

[5] Factorial of a number

→ The product of all positive integers which are less than or equal to the given positive number is referred to as the factorial of that given integer number.

For example, the factorial of the number 8 can be calculated by multiplying all the integers which are less than or equal to the 8 like this:

```
= 8
= 8*7*6*5*4*3*2*1
= 40320
```

Code:

```
# To take input from the user
num = int(input("Enter a number: "))

factorial = 1

# check if the number is negative, positive or zero
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1,num + 1):
       factorial = factorial*i
   print("The factorial of",num,"is",factorial)
```

Output:

```
Enter a number: 8
The factorial of 8 is 40320
```

[6] Reverse a string

Code:

```
s='Milan Bhadaliya'
rstr = ''
for i in s:
    rstr = i + rstr

print(rstr)
```

Output:

```
ayiladahB naliM
```

[7] Reverse a number

Code:

```
number = int(input("Enter the integer number: "))
revn=0
while (number > 0):
    remainder = number % 10
    revn = (revn * 10) + remainder
    number = number // 10
print("The reverse number is : ",revn)
```

Output:

```
Enter the integer number: 12345
The reverse number is :  54321
```

[8] Palindrome number

→ A palindrome number is a special number. After reversing a number, if it is the same as the original number referred to as a palindrome number. There can be a series of palindrome numbers. For example, 121, 232, 34543, 171, and 343 etc.

Code:

```
number = int(input("Enter the integer number: "))
tempnum=number
revn=0
while (number > 0):
    remainder = number % 10
    revn = (revn * 10) + remainder
    number = number // 10

if tempnum==revn:
    print(revn, 'is Palindrome number')
else:
    print(revn, 'is not Palindrome number')
```

Output:

```
Enter the integer number: 121
121 is Palindrome number
```

[9] Palindrome string

→ Just like a number, a string can also be a palindrome. The logic of checking a string is different from the number.

Code:

```
s='mom'
rstr = ''
for i in s:
    rstr = i + rstr
if s==rstr:
    print(rstr, 'is Palindrome string')
else:
    print(rstr, 'is not Palindrome string')
```

Output:

```
mom is Palindrome string
```

[10] Find duplicate characters in a string

Code:

```
s = input("Enter a string: ")
count_dict = {}

for char in s:
    if char != ' ':
        if char in count_dict:
            count_dict[char] += 1
        else:
            count_dict[char] = 1

print("Occurrence of characters in the given string:")
for char, count in count_dict.items():
    print(f"{s} contains '{char}' {count} times")
```

Output:

```
Enter a string: Milan Bhadaliya
Occurrence of characters in the given string:
Milan Bhadaliya contains 'M' 1 times
Milan Bhadaliya contains 'i' 2 times
Milan Bhadaliya contains 'l' 2 times
Milan Bhadaliya contains 'a' 4 times
Milan Bhadaliya contains 'n' 1 times
Milan Bhadaliya contains 'B' 1 times
Milan Bhadaliya contains 'h' 1 times
Milan Bhadaliya contains 'd' 1 times
Milan Bhadaliya contains 'y' 1 times
```

[11] Number is a Harshad Number

→ Harshad Numbers can be divided by the sum of its digits. They are also called Niven Numbers. For instance, 18 is a Harshad Number as it can be divided by 9, the sum of its digits (8+1=9). In this article, we will discuss various approaches to determine whether the given number is a Harshad Number in Python.

Example:

```
Input: 18
Output: 18 is a Harshad Number
Input: 15
Output: 15 is not a Harshad Number
```

Code:

```
st = input('Enter the number')
sum = 0
length = len(st)
for i in st:
   sum = sum + int(i)

if (int(st) % sum == 0):
   print('Number is Harshad number')
else:
   print('Number is not Harshad number')
```

Output:

```
Enter the number18
Number is Harshad number
```

[12] Number is Happy Number

→ A number is called happy if it leads to 1 after a sequence of steps wherein each step number is replaced by the sum of squares of its digit that is if we start with a Happy Number and keep replacing it with digits square sum, we reach 1. In this article, we will check if the given number is a Happy Number

Example:

19 is Happy Number,
$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

Code:

```
n=int(input('Enter the number'))
temp=n
st = set()
flag=1
while (flag==1):
    squareSum = 0
    while (n != 0):
        squareSum += (n % 10) * (n % 10)
        n = n // 10
    n=squareSum
    print(n)
    if (n == 1):
        flag=1
        break
    if n in st:
        flag=0
    st.add(n)

if(flag==1):
    print(temp,' is Happy Number')
else:
    print(temp,' is not Happy Number')
```

Output:

```
Enter the number19
82
68
100
1
19 is Happy Number
```

[13] Number is Ramanujan Numbers

→ A Ramanujan number is a number that can be expressed as the sum of two cubes in two different ways. Formally, a number N is a Ramanujan number if there exist positive integers a, b, c and d such that: $N = a^3 + b^3 = c^3 + d^3$ where (a, b) and (c, d) are distinct pairs of integers.

Also Known As:
Hardy-Ramanujan Number or Taxicab Number

Example:

The smallest Ramanujan number is 1729:
1729 = 123 + 13
1729 = 103 + 93

The name "Hardy-Ramanujan number" honors both G.H. Hardy, who first introduced the number to the public, and Srinivasa Ramanujan, who identified its interesting property.

Code:

```
# Define the number to check
number = int(input("Enter the number to check: "))

# Initialize a flag to determine if the number is a Ramanujan number
is_ramanujan = False

# Loop through possible values of a, b, c, and d
for a in range(1, int(number**(1/3)) + 1):
    for b in range(a, int(number**(1/3)) + 1):
        for c in range(1, int(number**(1/3)) + 1):
            for d in range(c, int(number**(1/3)) + 1):
                if a!=c and b!=d:
                    if a**3 + b**3 == c**3 + d**3 and a**3 + b**3 == number:
                        print(f"{number} is a Ramanujan number: {a}^3 + {b}^3 = {c}^3 + {d}^3")
                        is_ramanujan = True
                        break
            if is_ramanujan:
                break
        if is_ramanujan:
            break
    if is_ramanujan:
        break

if not is_ramanujan:
    print(f"{number} is not a Ramanujan number")
```

Output:

```
Enter the number to check: 1729
1729 is a Ramanujan number: 1^3 + 12^3 = 9^3 + 10^3
```

[14] Write a program to check if the two strings are anagrams or not.

→ An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, using all the original letters exactly once. For example, the words "listen" and "silent" are anagrams because they contain the same letters in different orders.

Code:

```
str1 = "listen"
str2 = "silent"
if len(str1) != len(str2):
   print(f"{str1} and {str2} are not anagrams")

anagram1 = sorted(str1.lower())
anagram2 = sorted(str2.lower())

if anagram1 == anagram2:
   print("They are anagrams")
else:
   print("They are not anagrams")
```

Output:

```
They are anagrams
```

ASCII-Table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	@	96	60	`
1	1	*[START OF HEADING]*	33	21	!	65	41	**A**	97	61	**a**
2	2	*[START OF TEXT]*	34	22	"	66	42	**B**	98	62	**b**
3	3	*[END OF TEXT]*	35	23	#	67	43	**C**	99	63	**c**
4	4	*[END OF TRANSMISSION]*	36	24	$	68	44	**D**	100	64	**d**
5	5	*[ENQUIRY]*	37	25	%	69	45	**E**	101	65	**e**
6	6	*[ACKNOWLEDGE]*	38	26	&	70	46	**F**	102	66	**f**
7	7	*[BELL]*	39	27	'	71	47	**G**	103	67	**g**
8	8	*[BACKSPACE]*	40	28	(	72	48	**H**	104	68	**h**
9	9	*[HORIZONTAL TAB]*	41	29	)	73	49	**I**	105	69	**i**
10	A	*[LINE FEED]*	42	2A	*	74	4A	**J**	106	6A	**j**
11	B	*[VERTICAL TAB]*	43	2B	+	75	4B	**K**	107	6B	**k**
12	C	*[FORM FEED]*	44	2C	,	76	4C	**L**	108	6C	**l**
13	D	*[CARRIAGE RETURN]*	45	2D	-	77	4D	**M**	109	6D	**m**
14	E	*[SHIFT OUT]*	46	2E	.	78	4E	**N**	110	6E	**n**
15	F	*[SHIFT IN]*	47	2F	/	79	4F	**O**	111	6F	**o**
16	10	*[DATA LINK ESCAPE]*	48	30	**0**	80	50	**P**	112	70	**p**
17	11	*[DEVICE CONTROL 1]*	49	31	**1**	81	51	**Q**	113	71	**q**
18	12	*[DEVICE CONTROL 2]*	50	32	**2**	82	52	**R**	114	72	**r**
19	13	*[DEVICE CONTROL 3]*	51	33	**3**	83	53	**S**	115	73	**s**
20	14	*[DEVICE CONTROL 4]*	52	34	**4**	84	54	**T**	116	74	**t**
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	**5**	85	55	**U**	117	75	**u**
22	16	*[SYNCHRONOUS IDLE]*	54	36	**6**	86	56	**V**	118	76	**v**
23	17	*[END OF TRANS. BLOCK]*	55	37	**7**	87	57	**W**	119	77	**w**
24	18	*[CANCEL]*	56	38	**8**	88	58	**X**	120	78	**x**
25	19	*[END OF MEDIUM]*	57	39	**9**	89	59	**Y**	121	79	**y**
26	1A	*[SUBSTITUTE]*	58	3A	:	90	5A	**Z**	122	7A	**z**
27	1B	*[ESCAPE]*	59	3B	;	91	5B	[	123	7B	{
28	1C	*[FILE SEPARATOR]*	60	3C	<	92	5C	\	124	7C	|
29	1D	*[GROUP SEPARATOR]*	61	3D	=	93	5D	]	125	7D	}
30	1E	*[RECORD SEPARATOR]*	62	3E	>	94	5E	^	126	7E	~
31	1F	*[UNIT SEPARATOR]*	63	3F	?	95	5F	_	127	7F	*[DEL]*

ASCII to Character:

```
print(chr(65))
```

Character to ASCII:

```
print(ord('A'))
```

Print A to Z using ASCII value:

```
for i in range (65,91):
    print(chr(i))
```

Extra Example for Pattern:

# Number of rows 	
n = 5
 	
# Loop through each row 	
for i in range(1, n + 1):
  # Print stars for each column in the row
  print('*' * i)
	*
	**
	***
	****
	*****

# Number of rows and columns 	
n = 5
 	
# Loop through each row 	
for i in range(n):
  # Print stars for each column in the row
  print('*' * n)
	*****
	*****
	*****
	*****
	*****

# Number of rows 	
n = 5
 	
# Loop through each row 	
for i in range(1, n + 1):
  # Print numbers for each column in the row
  print(' '.join(str(j) for j in range(1, i + 1)))
	1
	1 2
	1 2 3
	1 2 3 4
	1 2 3 4 5

# Number of rows for the top half 	
n = 5
 	
# Top half of the diamond 	
for i in range(1, n + 1):
  print(' ' * (n - i) + '*' * (2 * i - 1))
 	
# Bottom half of the diamond 	
for i in range(n - 1, 0, -1):
  print(' ' * (n - i) + '*' * (2 * i - 1))
	

# Number of rows 	
n = 5
 	
# Loop through each row 	
for i in range(1, n + 1):
  # Print spaces for the left side
  print(' ' * (n - i) + '*' * (2 * i - 1))
	

5.1 Types of Files

In Python, there are several types of files you can work with, each serving different purposes. Here are the main types:

1. **Text Files:** These contain plain text and are typically encoded in formats like UTF-8. You can read and write to them.
2. **Binary Files:** These contain data in a format that is not human-readable (e.g., images, audio). You handle them similarly to text files but use binary modes.
3. **CSV Files:** Comma-separated values files are often used for storing tabular data. Python's 'csv' module provides functions to read from and write to CSV files easily.
4. **JSON Files:** JavaScript Object Notation files are used for data interchange. You can use the 'json' module to read and write JSON data.
5. **Excel Files:** Python can work with Excel files (.xlsx) using libraries like 'pandas' or 'openpyxl'.
6. **Pickle Files:** These files store Python objects in a binary format. Use the 'pickle' module to serialize and deserialize Python objects.
7. **SQLite Database Files:** SQLite databases are files that store structured data. Use the 'sqlite3' module to interact with these databases.
8. **Log Files:** Typically text files used to store logs from applications. You can write logs using the built-in `logging` module.

Each file type has its own use cases, so the choice depends on what data you're handling and how you want to store it!

→ The key function for working with files in Python is the open() function.
→ The open() function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:

[1] "r" - Read - Default value. Opens a file for reading, error if the file does not exist
[2] "a" - Append - Opens a file for appending, creates the file if it does not exist
[3] "w" - Write - Opens a file for writing, creates the file if it does not exist
[4] "x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

[1] "t" - Text - Default value. Text mode
[2] "b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("\path")
```

The code above is the same as:

```
f = open("\path", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

5.2 File Methods to Read Data, Write Data, Creating File

demo.txt

With great power comes great responsibility.

★ File Opening

⋆Opening a File in the Same Directory as a Python Script

Code:

```
f = open("demo.txt", "r")
print(f.read())
```

Output:

```
With great power comes great responsibility.
```

⋆Open a file on a different location

Code:

```
path='C:\\Users\\Desktop\\New folder\\demo.txt'
f = open(path, "r")
print(f.read())
```

Output:

```
With great power comes great responsibility.
```

⋆Read only some Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return.

→ Return the 6 first characters of the file

Code:

```
f = open("demo.txt", "r")
print(f.read(6))
```

Output:

```
With g
```

demo.txt

With great power comes great responsibility.

Why so Serious?

⋆Read Lines

You can return one line by using the readline() method.

→ Read one line of the file.

Code:

```
f = open("demo.txt", "r")
print(f.readline())
```

Output:

```
With great power comes great responsibility.
```

→ By calling readline() two times, you can read the two first lines.

Code:

```
f = open("demo.txt", "r")
print(f.readline())
print(f.readline())
```

Output:

```
With great power comes great responsibility.
Why so Serious?
```

⋆By looping through the lines of the file, you can read the whole file, line by line.

Code:

```
f = open("demo.txt", "r")
for x in f:
  print(x)
```

Output:

```
With great power comes great responsibility.
Why so Serious?
```

★ Close File

→ It is a good practice to always close the file when you are done with it.

Code:

```
f = open("demo.txt", "r")
print(f.readline())
f.close()
```

Output:

```
With great power comes great responsibility.
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

★ File Write

To write to an existing file, you must add a parameter to the open() function.

[1] "a" - Append - will append to the end of the file
[2] "w" - Write - will overwrite any existing content

→ Open the file demo.txt and append content to the file.

Code:

```
f = open("demo.txt", "a")
f.write("Why so Serious?")
f.close()

#open and read the file after the appending.
f = open("demo.txt", "r")
print(f.read())
```

Output:

```
With great power comes great responsibility.
Why so Serious?
```

→ Open the file demo.txt and overwrite the content.

Code:

```
f = open("demo.txt", "w")
f.write("Spider-Man\nSuperman\nBatman\nAquaman\nCaptain America\nThor\nIron Man
\nBlack Widow\nDeadpool ")
f.close()

#open and read the file after the overwriting.
f = open("demo.txt", "r")
print(f.read())
```

Output:

```
Spider-Man
Superman
Batman
Aquaman
Captain America
Thor
Iron Man
Black Widow
Deadpool
```

Note: the "w" method will overwrite the entire file.

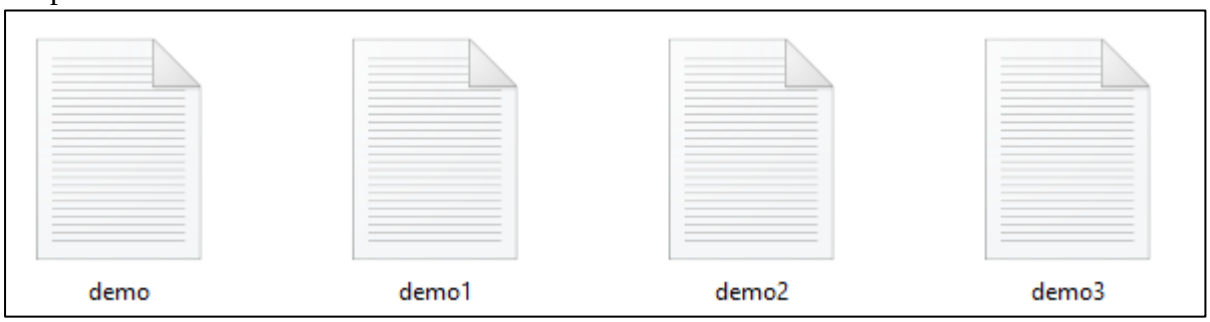
Create a New File

[1] "x" - Create - will create a file, returns an error if the file exist
[2] "a" - Append - will create a file if the specified file does not exist
[3] "w" - Write - will create a file if the specified file does not exist

Code:

```
f = open("demo1.txt", "x")
f = open("demo2.txt", "w")
f = open("demo3.txt", "a")
```

Output:



★ Delete a File

To delete a file, you must import the OS module, and run its os.remove() function.

Code:

```
import os
os.remove("demo3.txt")
```

⋆ Check if File exist

Code:

```
import os
if os.path.exists("demo3.txt"):
  os.remove("demo3.txt")
else:
  print("The file does not exist")
```

Output:

```
The file does not exist
```

★ Delete Folder

Code:

```
import os
os.rmdir("demofolder")
```

Note: You can only remove empty folders.

5.3 Reading and Writing Binary Files

★ Key Differences between Text and Binary Modes

⋆Text Mode ('t')

→ Default Mode: When you open a file without specifying a mode, it defaults to text mode.
→ Encoding: Text mode reads and writes strings. When you read from a file, it automatically decodes the bytes to a string using a specified encoding (default is usually UTF-8).
→ Line Endings: In text mode, Python translates line endings. For example, on Windows, it converts \r\n (carriage return + line feed) to \n (line feed).
→ Use Cases: Suitable for files containing human-readable text, such as .txt, .csv, etc.

⋆Binary Mode ('b')

→ Raw Bytes: Binary mode reads and writes raw bytes. It does not perform any encoding or decoding.
→ No Translation: No translation of line endings occurs; you get exactly what is in the file.
→ Use Cases: Ideal for non-text files like images, audio, or binary data files (e.g., .png, .jpg, .exe).

Code:

```
file = open('demo.txt','rb')
print(file.read())
file.close()
```

Output:

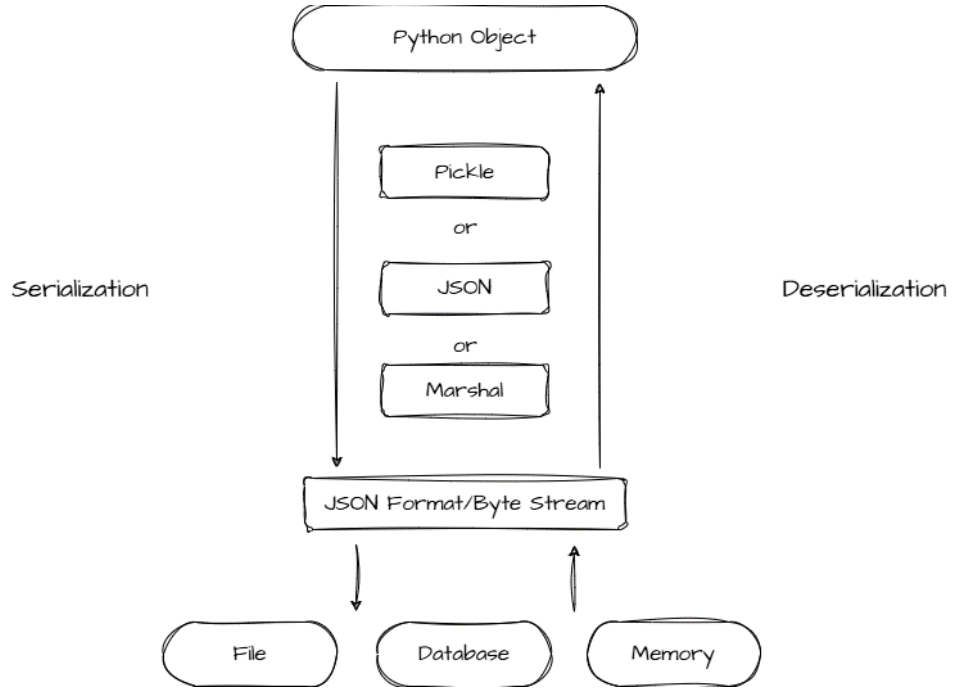
```
b'With great power comes great responsibility.\r\nWhy so Serious?'
```

5.4 The Pickle Module

The pickle module in Python is used for serializing and deserializing Python objects, allowing you to save complex data structures to a file or transfer them over a network. Serialization, the process of converting an object into a byte stream (0s and 1s), is also called pickling, flattening, or marshalling. We can convert the byte stream (generated through pickling) back into Python objects through a process called unpickling. In simpler terms, deserialization is the reverse process.

For this purpose, Python provides the following three modules:

1. Pickle Module
2. JSON Module
3. Marshal Module



Pickle In real-world scenarios

→ The use of pickling and unpickling is widespread, as they allow us to easily transfer data from one server or system to another, and then store it in a file or database.

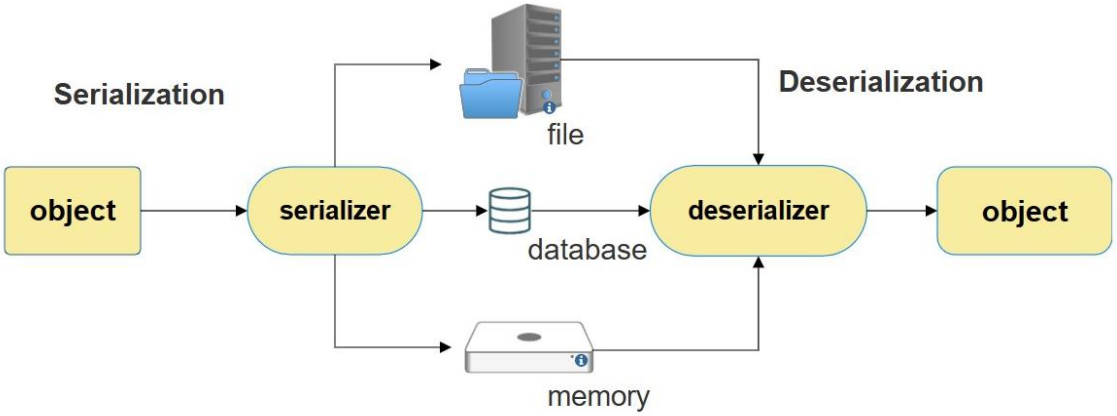
Note: Be cautious with pickle when loading data from untrusted sources, as it can execute arbitrary code during deserialization. Use safer alternatives (like JSON) for untrusted data.

Process in Simple Language

→ The pickle module is used to turn Python objects (like lists or dictionaries) into a format that can be saved on your computer, often as a file. This process is called "pickling."

→ When you pickle an object, it creates a special stream of data that holds all the information needed to recreate that object later.

→ Later, when you want to use the saved object again, you can "unpickle" it, which means turning that stream of data back into the original Python object. This way, you can store complex data structures and load them whenever you need them, making it easy to transfer data between programs or save it for later use.



This module provides the following four methods:

1. dump(): This method is used to serialize to an open file object
2. dumps(): This method is used for serializing to a string
3. load(): This method deserializes from an open-like object.
4. loads(): This does deserialization from a string.

Pickle a dictionary / Dump Functions

→ the dump function is used to write a pickled version of the object into a file.

Code:

```
import pickle

characters = {
   'Spider-Man': {
      'real_name': 'Peter Parker',
      'universe': 'Marvel',
      'powers': ['wall-crawling', 'super strength', 'spider-sense']
   },
   'John Wick': {
      'real_name': 'John Wick',
      'universe': 'None (original character)',
      'skills': ['expert marksman', 'hand-to-hand combat', 'strategic thinking']
   },
   'Batman': {
      'real_name': 'Bruce Wayne',
      'universe': 'DC',
      'powers': ['high intelligence', 'martial arts', 'gadgetry']
   }
}

with open('data.txt', 'wb') as file:
   pickle.dump(characters, file)
```

Output:

```
€□•V□      }”(Œ
Spider-Man”}”(Œ real_name”Œ
Peter Parker”Œ□universe”Œ□Marvel”Œ□powers”]”(Œ
wall-crawling”Œ□super strength”Œ
spider-sense”euŒ        John Wick”}”(h□h
h□Œ□None (original character)”Œ□skills”]”(Œ□expert marksman”Œ□hand-to-hand combat”
Œ□strategic thinking”euŒ□Batman”}”(h□Œ
Bruce Wayne”h□Œ□DC”h ]”(Œ□high intelligence”Œ
martial arts”Œ□gadgetry”euu.
```

Unpickle a dictionary / Load Function

→ This function is used to read the information in picked form from a file and reconstruct it into the original form.

Code:

```
import pickle
with open('data.txt', 'rb') as file:
   loaded_data = pickle.load(file)
print(loaded_data)
```

Output:

```
{'Spider-Man': {'real_name': 'Peter Parker', 'universe': 'Marvel', 'powers': ['wall-crawling', 'super strength', 'spider-sense']}, 'John Wick': {'real_name': 'John Wick', 'universe': 'None (original character)', 'skills': ['expert marksman', 'hand-to-hand combat', 'strategic thinking']}, 'Batman': {'real_name': 'Bruce Wayne', 'universe': 'DC', 'powers': ['high intelligence', 'martial arts', 'gadgetry']}}
```

Dumps Function and Loads Function

Python Dumps Function

→ This function returns the pickled representation of the given data in bytes object form.
→ This syntax is similar to the dump() function, except we do not have a file argument here.

Python Loads Function

→ This function is used to read the pickled representation from an object and returns the reconstructed version.
→ This is similar to the dumps() function except that here we pass the bytes object rather than a normal object.

Code:

```
import pickle

data = {'name': 'Jonathan Wick','aka': 'John Wick', 'Nickname': 'Baba Yaga'}
bytes_data = pickle.dumps(data)
print("Serialized data (bytes):", bytes_data)

loaded_data = pickle.loads(bytes_data)
print("Loaded data:", loaded_data)
```

Output:

```
Serialized data (bytes):
b'\x80\x04\x95E\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x04name\x94\x8c\rJonathan
Wick\x94\x8c\x03aka\x94\x8c\tJohn Wick\x94\x8c\x08Nickname\x94\x8c\tBaba
Yaga\x94u.'

Loaded data: {'name': 'Jonathan Wick', 'aka': 'John Wick', 'Nickname': 'Baba Yaga'}
```

Advantages of Using Pickle in Python

→ Recursive objects (objects containing references to themselves): Pickle keeps track of the objects it has already serialized, so later references to the same object won't be serialized again. (The marshal module breaks for this.)
→ Object sharing (references to the same object in different places): This is similar to self-referencing objects. Pickle stores the object once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.
→ User-defined classes and their instances: Marshal does not support these at all, but Pickle can save and restore class instances transparently. The class definition must be importable and live in the same module as when the object was stored.

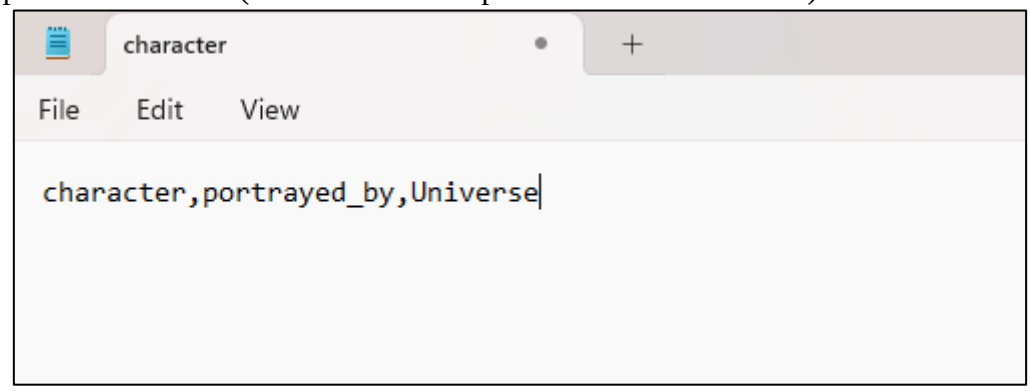
Disadvantages of Using Pickle in Python

→ Python Version Dependency: Data of pickle is so sensitive to the version of Python that produced. Pickled object created with one version of Python that might not be unpickled with a various version.
→ Non-Readble: The format of pickle is binary and not easily readable or editable by humans. The contracts that are in JSON or XML format can be easily modified.
→ Large data inefficiency: large datasets can slow down the pickling and unpickling. Serialization might be more appropriate for such use-cases.

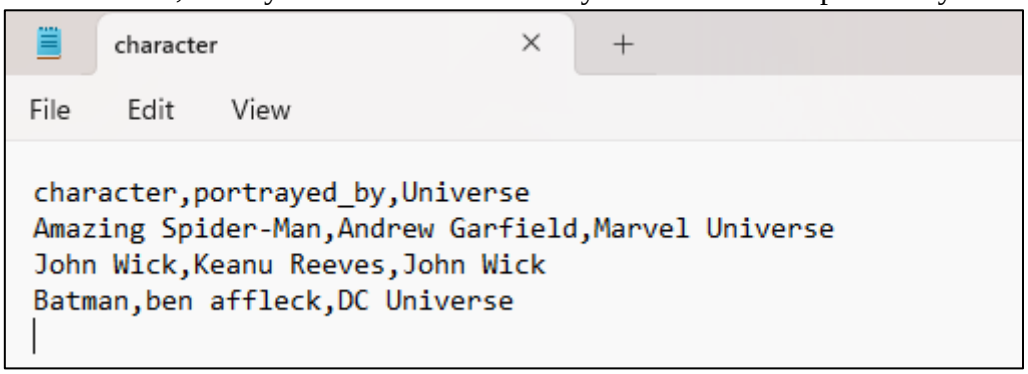
5.5 Reading and Writing CSV Files

→ Step-by-Step guide on how to create a CSV file using Notepad

1. Open Notepad
2. Type the header row (Use commas to separate each column name)



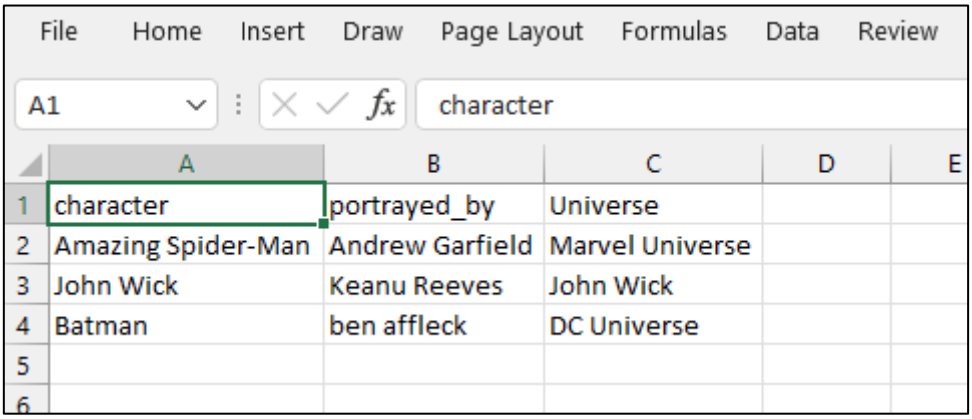
3. On the next lines, write your data. Each data entry should also be separated by commas



4. Save the File (Enter your desired filename followed by .csv)

(character.csv file)

5. Open with Excel



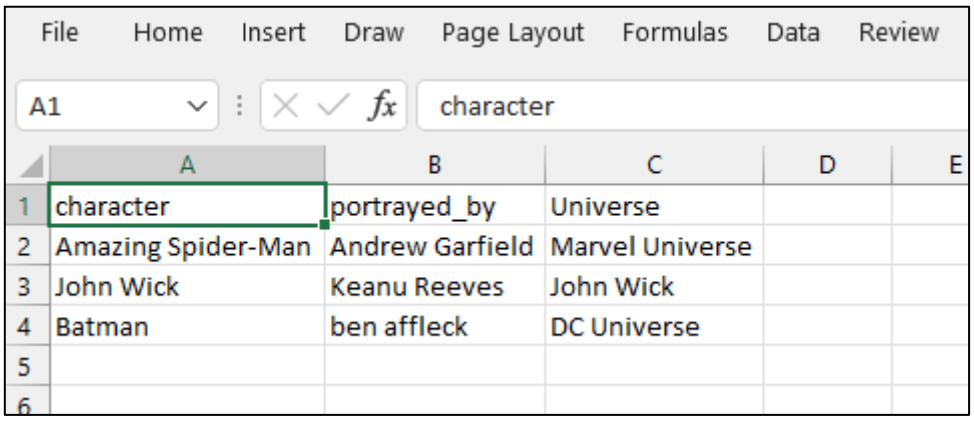
Reading and writing CSV (Comma-Separated Values) files is a common task in data processing.

Writing CSV Files

Code:

```
import csv

data = [
    ['character', 'portrayed_by', 'Universe'],
    ['Amazing Spider-Man', 'Andrew Garfield', 'Marvel Universe'],
    ['John Wick', 'Keanu Reeves', 'John Wick'],
    ['Batman', 'ben affleck', 'DC Universe']
]

with open('character.csv', mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

Output:



Reading CSV Files

Code:

```
import csv

with open('character.csv', mode='r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Output:

```
['character', 'portrayed_by', 'Universe']
['Amazing Spider-Man', 'Andrew Garfield', 'Marvel Universe']
['John Wick', 'Keanu Reeves', 'John Wick']
['Batman', 'ben affleck', 'DC Universe']
```

Note: If you're dealing with larger datasets or require additional features, it's advisable to use the pandas library.

NOTE: When working on the CSV file program, please pay attention to the following guidelines based on your coursework:

→ **If your question is from CO-4:** Use the built-in csv module in Python for handling CSV files. This module provides functions to read from and write to CSV files easily.

→ **If your question is from CO-3:** Use the pandas module for working with CSV files. Pandas offers powerful data manipulation capabilities, making it ideal for more complex data analysis tasks.

5.6 List of logical programs in Python

[1] Write a Python program to:

1. Retrieve and display the current working directory.
2. List all contents of the current directory.
3. Prompt the user to specify a subdirectory they wish to open.
4. Check if the specified subdirectory exists.
5. If it exists, display the contents of the subdirectory.
6. If it does not exist, inform the user that the directory does not exist.

Code:

```
import os

# Retrieve and print the current working directory
current_directory = os.getcwd()
print("Current Working Directory:", current_directory)

# List and print all contents of the current directory
contents = os.listdir(current_directory)
print("\nContents of Current Directory:", contents)

sub_dir = input("\nWhich directory would you like to open?\n")
subdirectory = os.path.join(current_directory, sub_dir)

# Check if the subdirectory exists and list its contents
if os.path.exists(subdirectory) and os.path.isdir(subdirectory):
    contents = os.listdir(subdirectory)
    print("Contents of",sub_dir, "Directory:")
    for item in contents:
        print(item)
else:
    print(sub_dir, "directory does not exist.")
```

Output:

Input: Notes	Current Working Directory: C:\Users\Milan\Desktop\Python Contents of Current Directory: ['character.csv', 'demo.py', 'Notes'] Which directory would you like to open? Notes Contents of Notes Directory: Unit-1
Input: note	Current Working Directory: C:\Users\Milan\Desktop\Python Contents of Current Directory: ['character.csv', 'demo.py', 'Notes'] Which directory would you like to open? note note directory does not exist.

[2] Create a Python program that reads a text file and updates it by converting all characters to uppercase.

Code:

```
with open("demo.txt", 'r+') as file:
    content = file.read()
    print("Original content:\n", content)
    capitalized_content = content.upper()
    file.seek(0)
    file.write(capitalized_content)
    file.truncate()
    file.seek(0)
    content = file.read()

print("\nCapitalized content:\n", content)
```

Output:

```
Original content:
With great power comes great responsibility.
Why so Serious?

Capitalized content:
WITH GREAT POWER COMES GREAT RESPONSIBILITY.
WHY SO SERIOUS?
```

[3] Create a Python program to transfer the contents from one text file to another. / Write a Python program to copy the contents from one text file to another.

Code:

```
with open("demo.txt", 'r') as source_file:
    content = source_file.read()

with open("demo1.txt", 'r+') as destination_file:
    destination_file.write(content)
    destination_file.truncate()
    destination_file.seek(0)
    content = destination_file.read()
print("Copied content\n",content)
```

Output:

```
Copied content
With great power comes great responsibility.
Why so Serious?
```

[4] Create a Python program to read from two specified files, file1 and file2, line by line, and write the combined content into a third file, file3.

demo1.txt

With great power
comes great
responsibility.
Why so Serious?

demo2.txt

Spiderman
Batman

demo3.txt

With great power
comes great
responsibility.
Why so Serious?
Spiderman
Batman

Code:

```
with open('demo3.txt', 'w') as file3:
    with open('demo1.txt', 'r') as file1:
        for line in file1:
            file3.write(line)

    with open('demo2.txt', 'r') as file2:
        file3.write('\n')
        for line in file2:
            file3.write(line)

with open('demo3.txt', 'r') as file3:
    content = file3.read()
print(content)
```

Output:

```
With great power comes great responsibility.
Why so Serious?
Spiderman
Batman
```

[5] Create a Python program to read a text file and perform the following tasks:

1. **Count and print the number of words.**
2. **Count and print the number of statements.**

Code:

```
word_count = 0
statement_count = 0

with open('demo3.txt', 'r') as file:
    content = file.read()

    words = content.split()
    word_count = len(words)

    statement_count = content.count('\n') + 1
    # Count statements (considering sentences ending with '.', '!', or '?')
    # statement_count = content.count('.') + content.count('!') + content.count('?')

print(f"Number of words: {word_count}")
print(f"Number of statements: {statement_count}")
```

Output:

```
Number of words: 11
Number of statements: 4
```

[6] Create a Python program to extract the name and date of birth (in the format mm-dd-yyyy) of students from a specified file named student.txt. (Note that we assume the name and date of birth are in a fixed format.)

student.txt

My Name is Spider Man and my DOB is 15-01-2015. I swing through the city, fighting crime and helping those in need. With great power comes great responsibility, and I take that to heart. Whether it's saving the day or facing my fears, I always rise to the occasion. My friends call me Peter, and I cherish the moments I get to spend with them. From web-slinging adventures to battling villains, every day is a new challenge. I believe in justice, friendship, and the importance of staying true to myself. After all, being a hero isn't just about the powers—it's about the heart!

Code:

```
with open('student.txt', 'r') as file:
    content = file.read()
    words = content.split()

for i in range(len(words)):
    if words[i] == 'is':
        if words[i-1] == 'Name':
            print('Name: ',words[i+1],words[i+2])
            break

for i in range(len(words)):
    if words[i] == 'is':
        if words[i-1] == 'DOB':
            print('DOB: ',words[i+1])
            break
```

Output:

```
Name:  Spider Man
DOB:  15-01-2015.
```

[7] Calculate Monthly Light Bill

creating a Python program to calculate the monthly light bill for users based on their electricity consumption. The billing structure is as follows:

1. The first 50 units are charged at ₹3 per unit.
2. Units from 51 to 200 are charged at ₹4 per unit.
3. Any units above 200 are charged at ₹5 per unit.
4. Additionally, there is a fixed charge of ₹120 added to the total bill.

Requirements:

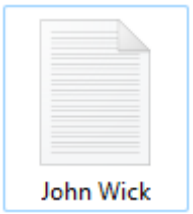
1. The program should prompt the user for their name and the number of light bill units consumed.
2. It should calculate the total bill based on the specified pricing structure.
3. The program should create a file named after the user (e.g., "John.txt") and store the following information:
   → User's name
   → Units consumed
   → Total bill amount

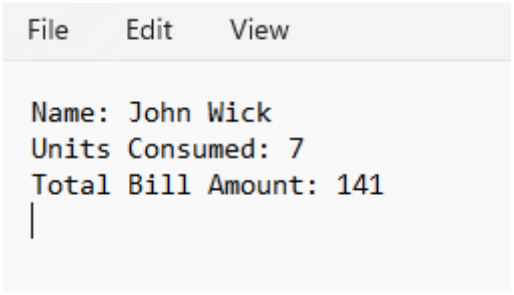
Code:

```
name = input("Enter your name: ")
units_consumed = int(input("Enter the number of light bill units: "))

bill_amount = 0
fixed_charge = 120

if units_consumed <= 50:
    bill_amount = units_consumed * 3
elif units_consumed <= 200:
    bill_amount = (50 * 3) + ((units_consumed - 50) * 4)
else:
    bill_amount = (50 * 3) + (150 * 4) + ((units_consumed - 200) * 5)

# Add the fixed charge
total_bill = bill_amount + fixed_charge

# Create a filename based on the user's name
filename = f"{name}.txt"

with open(filename, 'w') as file:
    file.write(f"Name: {name}\n")
    file.write(f"Units Consumed: {units_consumed}\n")
    file.write(f"Total Bill Amount: {total_bill}\n")
```

Output:

Enter your name: John Wick
Enter the number of light bill units: 007





★ What are the differences between the file modes w+, r+, and a+ in programming, and how do they affect file operations like reading and writing?

w+ (Write and Read	r+ (Read and Write)	a+ (Append and Read)
→ Opens a file for both reading and writing. → If the file already exists, it is truncated to zero length (i.e., all its content is deleted). → If the file does not exist, it is created.	→ Opens a file for both reading and writing. → The file must already exist; if it does not, the operation will fail. → The file's content is not truncated, so you can read from it and write to it without losing existing data (unless you explicitly overwrite).	→ Opens a file for both reading and writing. → If the file exists, all writes will be appended to the end of the file without truncating it. → If the file does not exist, it is created. → Allows reading from any part of the file, but writes will always occur at the end.

6.1 Introduction to Libraries

→ Normally, a library is a collection of books or is a room or place where many books are stored to be used later. Similarly, in the programming world, a library is a collection of precompiled codes that can be used later on in a program for some specific well-defined operations. Other than pre-compiled codes, a library may contain documentation, configuration data, message templates, classes, and values, etc.

→ A Python library is a collection of related modules. It contains bundles of code that can be used repeatedly in different programs. It makes Python Programming simpler and convenient for the programmer. As we don't need to write the same code again and again for different programs. Python libraries play a very vital role in fields of Machine Learning, Data Science, Data Visualization, etc.

Working of Python Library

→ As is stated above, a Python library is simply a collection of codes or modules of codes that we can use in a program for specific operations. We use libraries so that we don't need to write the code again in our program that is already available. But how it works. Actually, in the MS Windows environment, the library files have a DLL extension (Dynamic Load Libraries). When we link a library with our program and run that program, the linker automatically searches for that library. It extracts the functionalities of that library and interprets the program accordingly. That's how we use the methods of a library in our program. We will see further, how we bring in the libraries in our Python programs.

Python standard library

→ The Python Standard Library contains the exact syntax, semantics, and tokens of Python. It contains built-in modules that provide access to basic system functionality like I/O and some other core modules. Most of the Python Libraries are written in the C programming language. The Python standard library consists of more than 200 core modules. All these work together to make Python a high-level programming language. Python Standard Library plays a very important role. Without it, the programmers can't have access to the functionalities of Python. But other than this, there are several other libraries in Python that make a programmer's life easier. Let's have a look at some of the commonly used libraries:

[1] **TensorFlow:** This library was developed by Google in collaboration with the Brain Team. It is an open-source library used for high-level computations. It is also used in machine learning and deep learning algorithms. It contains a large number of tensor operations. Researchers also use this Python library to solve complex computations in Mathematics and Physics.

[2] **Matplotlib:** This library is responsible for plotting numerical data. And that's why it is used in data analysis. It is also an open-source library and plots high-defined figures like pie charts, histograms, scatterplots, graphs, etc.

[3] **Pandas:** Pandas are an important library for data scientists. It is an open-source machine learning library that provides flexible high-level data structures and a variety of analysis tools. It eases data analysis, data manipulation, and cleaning of data. Pandas support operations like Sorting, Re-indexing, Iteration, Concatenation, Conversion of data, Visualizations, Aggregations, etc.

[4] **Numpy:** The name "Numpy" stands for "Numerical Python". It is the commonly used library. It is a popular machine learning library that supports large matrices and multi-dimensional data. It consists of in-built mathematical functions for easy computations. Even libraries like TensorFlow use Numpy internally to perform several operations on tensors. Array Interface is one of the key features of this library.

[5] **SciPy:** The name "SciPy" stands for "Scientific Python". It is an open-source library used for high-level scientific computations. This library is built over an extension of Numpy. It works with Numpy to handle complex computations. While Numpy allows sorting and indexing of array data, the numerical data code is stored in SciPy. It is also widely used by application developers and engineers.

[6] **Scrapy:** It is an open-source library that is used for extracting data from websites. It provides very fast web crawling and high-level screen scraping. It can also be used for data mining and automated testing of data.

[7] **Scikit-learn:** It is a famous Python library to work with complex data. Scikit-learn is an open-source library that supports machine learning. It supports variously supervised and unsupervised algorithms like linear regression, classification, clustering, etc. This library works in association with Numpy and SciPy.

[8] **PyGame:** This library provides an easy interface to the Standard Directmedia Library (SDL) platform-independent graphics, audio, and input libraries. It is used for developing video games using computer graphics and audio libraries along with Python programming language.

[9] PyTorch: PyTorch is the largest machine learning library that optimizes tensor computations. It has rich APIs to perform tensor computations with strong GPU acceleration. It also helps to solve application issues related to neural networks.

[10] PyBrain: The name "PyBrain" stands for Python Based Reinforcement Learning, Artificial Intelligence, and Neural Networks library. It is an open-source library built for beginners in the field of Machine Learning. It provides fast and easy-to-use algorithms for machine learning tasks. It is so flexible and easily understandable and that's why is really helpful for developers that are new in research fields.

There are many more libraries in Python. We can use a suitable library for our purposes. Hence, Python libraries play a very crucial role and are very helpful to the developers.

You can find libraries in Python using the Python Package Index (PyPI) at https://pypi.org/. You can also use the pip tool to search for and install libraries directly from the command line.

Use of Libraries in Python Program

Example:

```
import math
a = 9
print(math.sqrt(a))
```

Output:

```
3.0
```

Importing specific items from a library module

→ As in the above code, we imported a complete library to use one of its methods. But we could have just imported "sqrt" from the math library. Python allows us to import specific items from a library.

Example:

```
from math import sqrt, sin
a = 9
print(math.sqrt(a))
print(math.sin(a))
```

Output:

```
3.0
0.4121184852417566
```

6.2 What is PIP?

→ PIP stands for "Pip Installs Packages." It is a package management system used for installing and managing software packages written in Python. PIP allows users to easily install libraries and dependencies from the Python Package Index (PyPI) and other sources.

Note: If you have Python version 3.4 or later, PIP is included by default.

To update pip, use the command below:

```
Python.exe -m pip install --upgrade pip
```

Navigate your command line to the location of Python's script directory, and type the following:

[1] For Installation:

```
pip install package_name
```

[2] For Uninstallation:

```
pip uninstall package_name
```

[3] For Upgrading Packages:

```
pip install --upgrade package-name
```

[4] List Packages

Use the list command to list all the packages installed on your system.

```
pip list
```

6.3 NumPy: Creating, accessing, manipulating array, performing various operations on array.

→ NumPy is a Python library used for working with arrays.

→ It also has functions for working in domain of linear algebra, fourier transform, and matrices.

→ NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

→ NumPy stands for Numerical Python.

Why is NumPy Faster than Lists?

→ NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

→ This behavior is called locality of reference in computer science.

→ This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

NumPy arrays provide several advantages over nested Python lists, making them a preferred choice for numerical computations and data analysis

1. Performance
   → Speed: NumPy arrays are implemented in C, allowing for faster operations compared to Python lists, especially for large datasets.
   → Memory Efficiency: NumPy arrays consume less memory due to their fixed-type nature, which reduces overhead.

2. Functionality
   → Vectorized Operations: NumPy supports element-wise operations, enabling concise and efficient computations without the need for explicit loops.
   → Broadcasting: Allows operations on arrays of different shapes, simplifying code and enhancing performance.

3. Data Types
   → Homogeneous Data: All elements in a NumPy array must be of the same type, which can enhance performance and memory usage.
   → Support for Multiple Data Types: NumPy can handle various numerical types (integers, floats, complex numbers) and even structured types.

4. Convenient Methods
   → Rich Functionality: NumPy provides a vast array of built-in functions for mathematical operations, statistical analysis, and linear algebra.
   → Slicing and Indexing: NumPy offers advanced indexing and slicing capabilities that are more powerful than those available in Python lists.

5. Interoperability
   → Integration with Libraries: Many scientific and machine learning libraries (e.g., SciPy, Pandas, TensorFlow) are built on top of NumPy, making it a core part of the scientific Python ecosystem.

6. Multidimensional Support
   → N-Dimensional Arrays: NumPy easily handles multi-dimensional arrays (matrices, tensors), allowing for complex data manipulation and representation.

7. Convenience for Data Analysis
   → Ease of Use: NumPy's syntax and functions facilitate mathematical operations and data manipulation, making it easier for users to work with large datasets.

8. Advanced Features
   → Linear Algebra Support: NumPy has built-in functions for matrix operations, which are essential for scientific computing.
   → Random Number Generation: Efficient tools for generating random numbers and distributions.

Installation of NumPy:

```
pip install numpy
```

```
C:\Users\Milan \Python\Python311\Scripts>pip install numpy
Collecting numpy
  Downloading numpy-2.1.2-cp311-cp311-win_amd64.whl.metadata (59 kB)
Downloading numpy-2.1.2-cp311-cp311-win_amd64.whl (12.9 MB)
   ---------------------------------------- 12.9/12.9 MB 5.6 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-2.1.2
```

The source code for NumPy is located at this github repository **https://github.com/numpy/numpy**

Example:	import numpy a = numpy.array([1, 2, 3, 4, 5]) print(a)	import numpy as np a = np.array([1, 2, 3, 4, 5]) print(a) print(type(a))
Output:	[1 2 3 4 5]	[1 2 3 4 5] <class 'numpy.ndarray'>

NumPy as np

→ NumPy is usually imported under the np alias.

→ **alias:** In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

```
import numpy as np
```

Now the NumPy package can be referred to as np instead of numpy.

Checking NumPy Version

Example:

```
import numpy as np
print(np.__version__)
```

Output:

```
2.1.2
```

0-D Arrays

→ 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example:

```
import numpy as np
a = np.array(7)
print(a)
```

Output:

```
7
```

1-D Arrays

→ An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

Example:

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
print(a)
```

Output:

```
[1 2 3 4 5]
```

2-D Arrays

→ An array that has 1-D arrays as its elements is called a 2-D array.

Example:

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
```

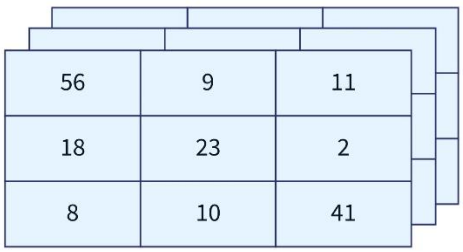
Output:

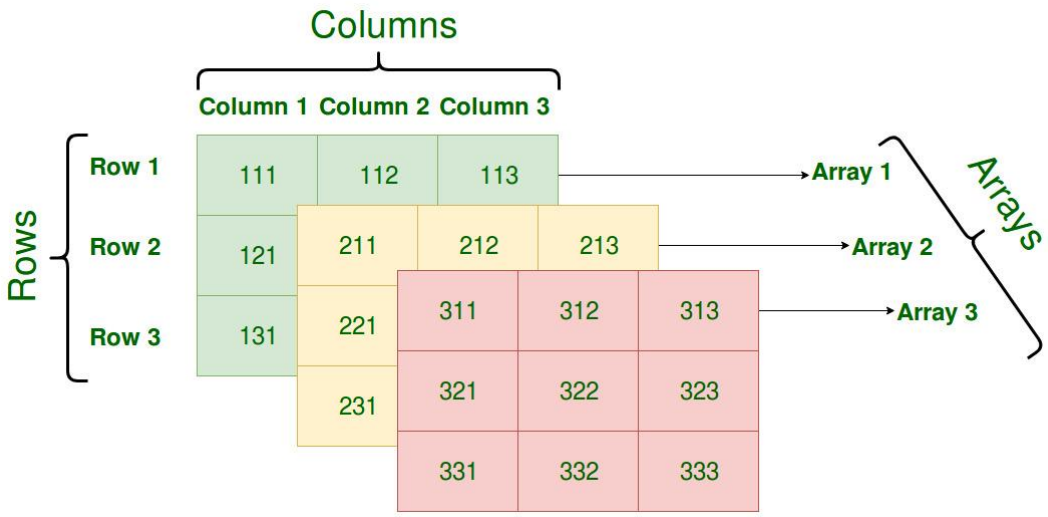
```
[[1 2 3]
 [4 5 6]]
```

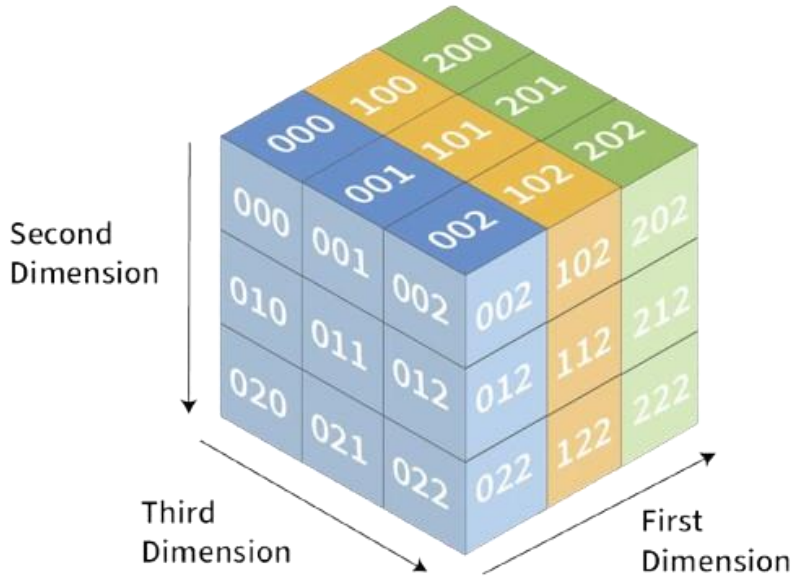
2-D Array

	Column 0	Column 1	Column 2
Row 0	a[0][0]	a[0][1]	a[0][2]
Row 1	a[1][0]	a[1][1]	a[1][2]
Row 2	a[2][0]	a[2][1]	a[2][2]
Row 3	a[3][0]	a[3][1]	a[3][2]
Row 4	a[4][0]	a[4][1]	a[4][2]

3-D Array







3-D arrays

→ An array that has 2-D arrays (matrices) as its elements is called 3-D array.

Example:

```
import numpy as np
a = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(a)
```

Output:

```
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
```

Check how many dimensions the arrays have

Example:

```
import numpy as np
a = np.array(7)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Output:

```
0
1
2
3
```

Higher Dimensional Arrays

→ An array can have any number of dimensions.

→ When the array is created, you can define the number of dimensions by using the ndmin argument.

Example:

```
import numpy as np
a = np.array([1, 2, 3, 4], ndmin=7)
print(a)
print('number of dimensions :', a.ndim)
```

Output:

```
[[[[[[[1 2 3 4]]]]]]]
number of dimensions : 7
```

Access Array Elements

→ Array indexing is the same as accessing an array element.

→ You can access an array element by referring to its index number.

→ The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example:

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a[0])
```

Output:

```
1
```

Write a code snippet to compute the sum of all elements in a NumPy array.

Example:

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
sum=0
for i in a:
    sum+=i
print(sum)
```

Output:

```
15
```

Write a code snippet to compute the sum of all elements in a NumPy array.(Using built-in sum function)

Example:

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
total_sum = np.sum(a)  #NumPy's built-in sum function
print(total_sum)
```

Output:

```
15
```

Access 2-D Arrays

Example:

```
import numpy as np
a = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(a[0, 1])
print(a[0][1])
```

Output:

```
2
2
```

Access 3-D Arrays

Example:

```
import numpy as np
a = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(a[0, 1, 2])
print(a[0] [1] [2])
```

Output:

```
6
6
```

Negative Indexing

Example:

```
import numpy as np
a = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('a[1, -1]: ',a[1, -1])
print('a[-2, -1]: ',a[-2, -1])
```

Output:

```
a[1, -1]: 10
a[-2, -1]: 5
```

Slicing arrays

1. Basic Syntax: [start:end]
2. Extended Syntax: [start:end:step]
   → If we don't pass start its considered 0
   → If we don't pass end its considered length of array in that dimension
   → If we don't pass step its considered 1

Example:

```
import numpy as np
a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(a[1:3])
print(a[2:])
print(a[:3])
print(a[-3:-1])
print(a[1:8:2])
print(a[::2])
```

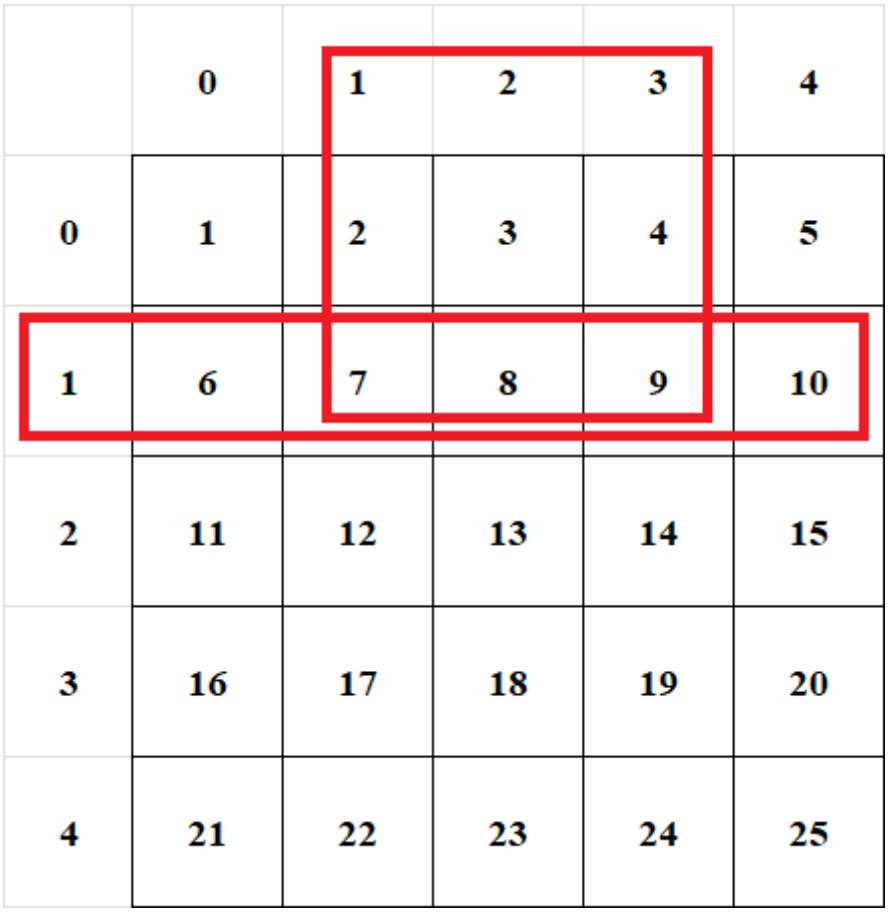
Output:

```
[2 3]
[ 3  4  5  6  7  8  9 10]
[1 2 3]
[8 9]
[2 4 6 8]
[1 3 5 7 9]
```

Slicing 2-D Arrays

Example:

```
import numpy as np
a = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(a[1, 1:4])
```



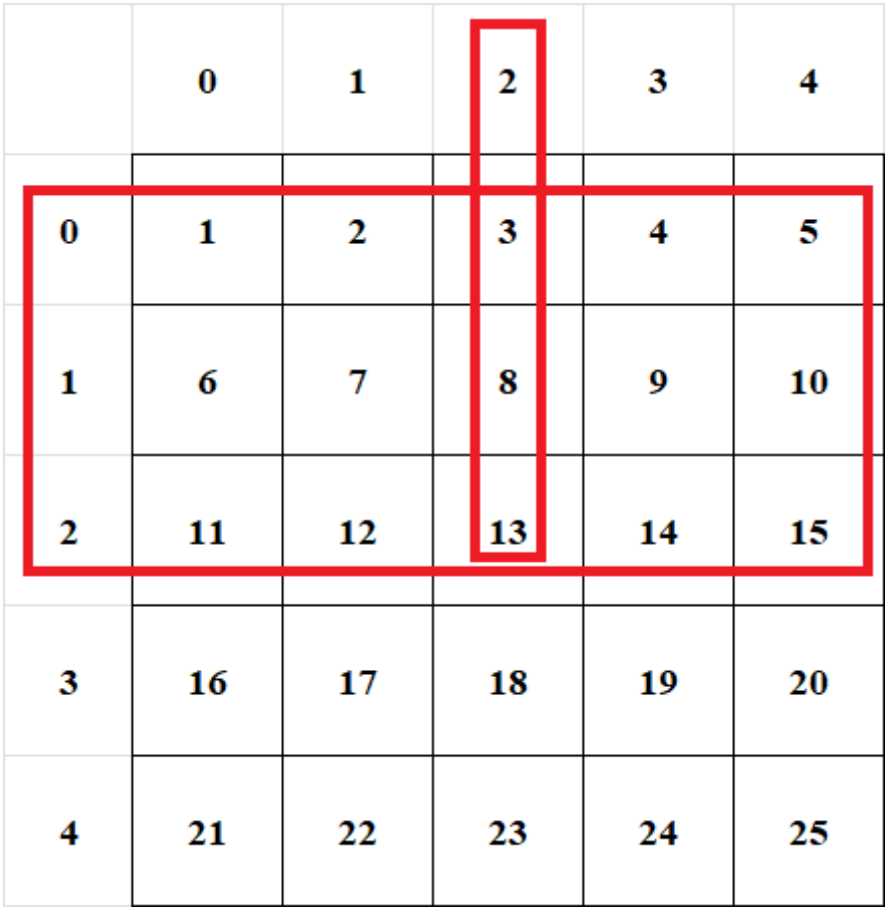
	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15
3	16	17	18	19	20
4	21	22	23	24	25

Output:

```
[7 8 9]
```

Example:

```
import numpy as np
a = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15], [16, 17, 18, 19, 20], [21, 22, 23, 24, 25]])
print(a[0:3, 2])
```



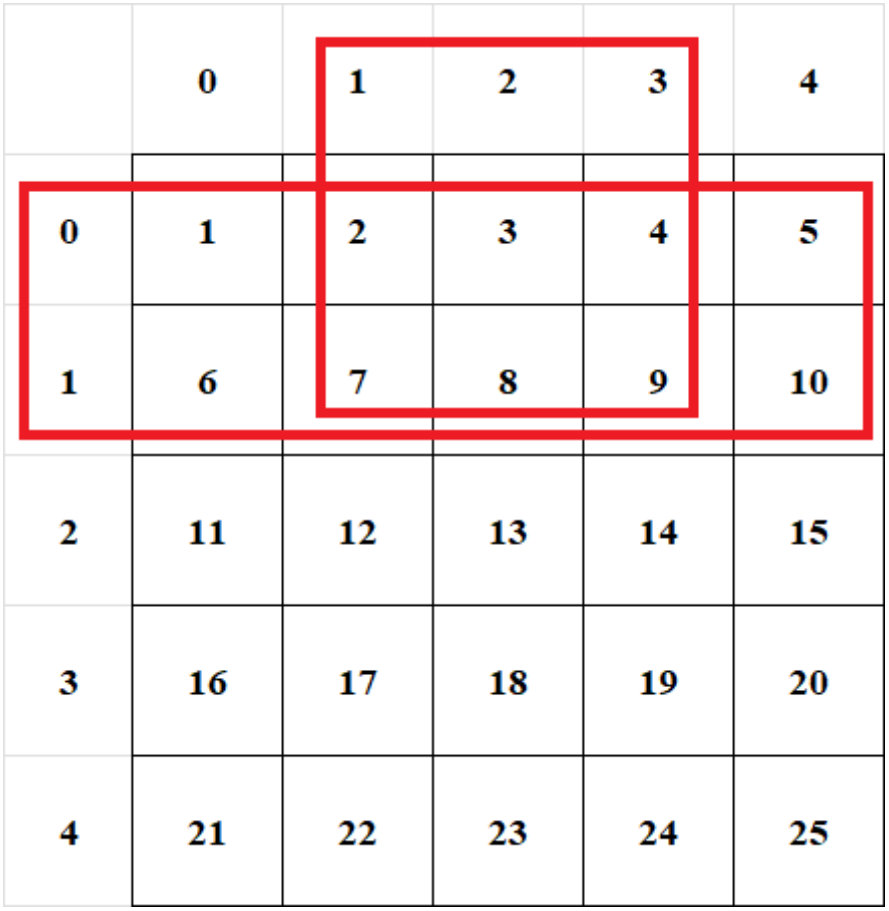
	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15
3	16	17	18	19	20
4	21	22	23	24	25

Output:

```
[3 8 13]
```

Example:

```
import numpy as np
a = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15], [16, 17, 18, 19, 20], [21, 22, 23, 24, 25]])
print(a[0:2, 1:4])
```



Output:

```
[[2 3 4]
 [7 8 9]]
```

[1] Write a Python program that uses NumPy to perform the following tasks:

1 Create a NumPy array from the given nested list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]].
2 Calculate and print the sum of each row in the array.

Code:

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
row_sums = np.sum(a, axis=1)
print(row_sums)
```

Output:

```
[6 15 24]
```

[2] Outline the steps to calculate the memory size occupied by a NumPy array. Given the array a = [1, 2, 3], write a Python program that calculates and prints the memory size it occupies. Make sure to include the necessary imports and any relevant calculations.

Code 1:

```
import numpy as np

a = np.array([1, 2, 3],dtype=np.int64)
#a = np.array([1, 2, 3],dtype='int64') You can also write it like this.

data_type = a.dtype
print(f"The data type of the array is: {data_type}")
memory_size = a.nbytes
print(f"Memory size occupied by the array: {memory_size} bytes")

a = np.array([1, 2, 3],dtype=np.int32)
#a = np.array([1, 2, 3],dtype='int32') You can also write it like this.

data_type = a.dtype
print(f"The data type of the array is: {data_type}")
memory_size = a.nbytes
print(f"Memory size occupied by the array: {memory_size} bytes")
```

Output:

```
The data type of the array is: int64
Memory size occupied by the array: 24 bytes
The data type of the array is: int32
Memory size occupied by the array: 12 bytes
```

Code 2:

```
import numpy as np

a = np.array([1, 2, 3],dtype=np.int64)

dtype_size = a.dtype.itemsize
num_elements = a.size
memory_size = dtype_size * num_elements

print(f"Size of each element (dtype size): {dtype_size} bytes")
print(f"Number of elements: {num_elements}")
print(f"Total memory size occupied by the array: {memory_size} bytes")
```

Output:

```
Size of each element (dtype size): 8 bytes
Number of elements: 3
Total memory size occupied by the array: 24 bytes
```

6.4 Pandas: Functions to create, access and manipulate sequence and data frame from file

→ Pandas is a powerful Python library designed for data manipulation and analysis. It provides a wide range of functions for cleaning, exploring, and analyzing datasets, making it an essential tool for data scientists.

→ The name "Pandas" is derived from "Panel Data" and "Python Data Analysis." It was created by Wes McKinney in 2008. With Pandas, users can effectively analyze large datasets and draw conclusions based on statistical theories.

→ One of the key features of Pandas is its ability to clean messy datasets, transforming them into readable and relevant formats. Relevant data is crucial in data science, and Pandas helps users derive insights from their data. For example, it can answer questions such as:

- Is there a correlation between two or more columns?
- What is the average value of a column?
- What are the maximum and minimum values?

→ Additionally, Pandas offers functions to remove irrelevant rows or those containing erroneous values, such as empty or NULL entries, a process known as data cleaning.

Installation of Pandas:

```
pip install pandas
```

```
Collecting pandas
  Downloading pandas-2.2.3-cp311-cp311-win_amd64.whl.metadata (19 kB)
Requirement already satisfied: numpy>=1.23.2 in c:\users\milan \python\python311\lib\site-packages (from pandas) (2. 1.2)
Collecting python-dateutil>=2.8.2 (from pandas)
  Downloading python_dateutil-2.9.0.post0-py2.py3-none-any.whl.metadata (8.4 kB)
Collecting pytz>=2020.1 (from pandas)
  Downloading pytz-2024.2-py2.py3-none-any.whl.metadata (22 kB)
Collecting tzdata>=2022.7 (from pandas)
  Downloading tzdata-2024.2-py2.py3-none-any.whl.metadata (1.4 kB)
Collecting six>=1.5 (from python-dateutil>=2.8.2->pandas)
  Downloading six-1.16.0-py2.py3-none-any.whl.metadata (1.8 kB)
Downloading pandas-2.2.3-cp311-cp311-win_amd64.whl (11.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.6/11.6 MB 11.0 MB/s eta 0:00:00
Downloading python_dateutil-2.9.0.post0-py2.py3-none-any.whl (229 kB)
Downloading pytz-2024.2-py2.py3-none-any.whl (508 kB)
Downloading tzdata-2024.2-py2.py3-none-any.whl (346 kB)
Downloading six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: pytz, tzdata, six, python-dateutil, pandas
Successfully installed pandas-2.2.3 python-dateutil-2.9.0.post0 pytz-2024.2 six-1.16.0 tzdata-2024.2
```

The source code for Pandas is located at this github repository **https://github.com/pandas-dev/pandas**

Code:

```
import pandas as pd

dataset = {
  'characters' : ["John Wick","Amazing Spider-Man","Batman"],
  'portrayed_by' : ["Keanu Reeves","Andrew Garfield","Ben Affleck"]
}

a = pd.DataFrame(dataset)

print(a)
print(type(a))
```

Output:

```
      characters          portrayed_by
0     John Wick           Keanu Reeves
1     Amazing Spider-Man  Andrew Garfield
2     Batman              Ben Affleck
<class 'pandas.core.frame.DataFrame'>
```

Pandas as pd

→ pandas is usually imported under the pd alias.

→ **alias:** In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

```
import pandas as pd
```

Now the Pandas package can be referred to as pd instead of pandas.

Checking pandas Version

Example:

```
import pandas as pd
print(pd.__version__)
```

Output:

```
2.2.3
```

★ Pandas DataFrames

Pandas DataFrames are a powerful data structure provided by the Pandas library in Python, designed for data manipulation and analysis. They are essentially two-dimensional, size-mutable, and potentially heterogeneous tabular data structures with labeled axes (rows and columns).

Key Features:

→ Labeled Axes: Each row and column can be labeled, making it easy to access data.
→ Heterogeneous Data: Different columns can hold different types of data (e.g., integers, floats, strings).
→ Size-Mutable: You can easily add or remove columns and rows.
→ Alignment: DataFrames automatically align data based on labels, simplifying operations between different datasets.
→ Powerful Data Manipulation: Built-in functions for filtering, grouping, merging, and reshaping data.

A pandas DataFrame can be created using various inputs like:

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

Create an Empty DataFrame

Code:

```
import pandas as pd
a = pd.DataFrame()
print(a)
```

Output:

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

Code:

```
import pandas as pd
data = [1,2,3,4,5]
a = pd.DataFrame(data)
print(a)
```

Output:

```
   0
0  1
1  2
2  3
3  4
4  5
```

Code:

```
import pandas as pd

dataset = {
  'characters' : ["John Wick","Amazing Spider-Man","Batman"],
  'portrayed_by' : ["Keanu Reeves","Andrew Garfield","Ben Affleck"]
}

a = pd.DataFrame(dataset)
print(a)
```

Output:

	characters	portrayed_by
0	John Wick	Keanu Reeves
1	Amazing Spider-Man	Andrew Garfield
2	Batman	Ben Affleck

Locate Row

→ Pandas use the loc attribute to return one or more specified row(s)

Code:

```
import pandas as pd

dataset = {
  'characters' : ["John Wick","Amazing Spider-Man","Batman"],
  'portrayed_by' : ["Keanu Reeves","Andrew Garfield","Ben Affleck"]
}

a = pd.DataFrame(dataset)
print(a.loc[0])
print(a.loc[1])
```

Output:

```
characters              John Wick
portrayed_by            Keanu Reeves
Name: 0, dtype: object

characters              Amazing Spider-Man
portrayed_by            Andrew Garfield
Name: 1, dtype: object
```

Code:

```
import pandas as pd

dataset = {
  'characters' : ["John Wick","Amazing Spider-Man","Batman"],
  'portrayed_by' : ["Keanu Reeves","Andrew Garfield","Ben Affleck"]
}

a = pd.DataFrame(dataset)
print(a.loc[[0,1]])
```

Output:

	characters	portrayed_by
0	John Wick	Keanu Reeves
1	Amazing Spider-Man	Andrew Garfield

Named Indexes

→ With the index argument, you can name your own indexes.

Code:

```
import pandas as pd

dataset = {
  'characters' : ["John Wick","Amazing Spider-Man","Batman"],
  'portrayed_by' : ["Keanu Reeves","Andrew Garfield","Ben Affleck"]
}


a = pd.DataFrame(dataset, index = ["char1", "char2", "char3"])
print(a)
```

Output:

	characters	portrayed_by
char1	John Wick	Keanu Reeves
char2	Amazing Spider-Man	Andrew Garfield
char3	Batman	Ben Affleck

Locate Named Indexes

→ Use the named index in the loc attribute to return the specified row(s).

Code:

```
import pandas as pd

dataset = {
  'characters' : ["John Wick","Amazing Spider-Man","Batman"],
  'portrayed_by' : ["Keanu Reeves","Andrew Garfield","Ben Affleck"]
}


a = pd.DataFrame(dataset, index = ["char1", "char2", "char3"])
print(a.loc['char1'])
```

Output:

```
characters      John Wick
portrayed_by    Keanu Reeves
Name: char1, dtype: object
```

Column Selection

Code:

```
import pandas as pd

dataset = {
  'characters' : ["John Wick","Amazing Spider-Man","Batman"],
  'portrayed_by' : ["Keanu Reeves","Andrew Garfield","Ben Affleck"]
}

a = pd.DataFrame(dataset)
print(a[['characters']])
```

Output:

```
        characters
0       John Wick
1       Amazing Spider-Man
2       Batman
```

Viewing Data:

```
print(a.head()) # first five rows
print(a.head(2)) # first two rows
print(a.tail()) # last five rows
print(a.tail(2)) # last two rows
```

Selecting Columns:

```
names = a['Name']  # Select a single column
data = a[['Name', 'DOB']]  # Select multiple columns
```

Adding a New Column:

```
a['Salary'] = [5000, 6000, 7000]
```

Filtering Rows:

```
filtered_a = a[a['salary'] > 5000]
print(filtered_a)
```

Other common functions

```
print(a.iloc[0]) #iloc stands for "integer location".
print(a.shape) #Return a dimensionality of the DataFrame
print(a.describe(include='all')) #describe() method gives us summary statistics for numerical columns in our DataFrame.
print(a.info()) #info() method allows us to learn the shape of object types of our data
```

6.5 Matplotlib: Line Graphs, Scatter Graph, Pie Charts, Bar Charts, Figures and Subplot

Matplotlib is a low-level graph plotting library in Python that functions as a visualization tool. It was developed by John D. Hunter and is open-source, allowing for free usage. While primarily written in Python, some components are implemented in C, Objective-C, and JavaScript for better platform compatibility.

Installation of Matplotlib:

```
pip install matplotlib
```

The source code for Matplotlib is located at this github repository https://github.com/matplotlib/matplotlib

Checking Matplotlib Version

Code:

```
import matplotlib
print(matplotlib.__version__)
```

Output:

```
3.9.2
```

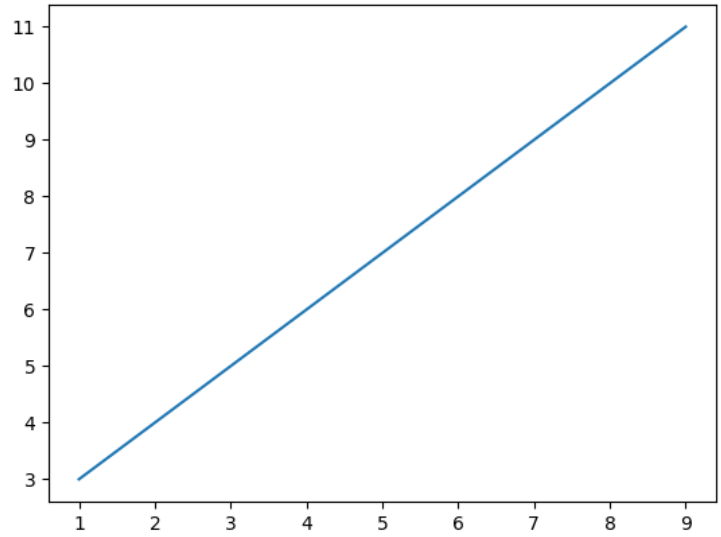
Matplotlib Plotting

- → The plot() function is used to display points (markers) on a graph. By default, it connects the points with lines.
- → To specify the points, you provide two arrays: one for the x-axis values and another for the y-axis values.
- → For example, to draw a line from the point (1, 3) to (9, 11), you would pass the arrays [1, 9] and [3, 11] to the plot() function.

Code:

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 9])
ypoints = np.array([3, 11])

plt.plot(xpoints, ypoints)
plt.show()
```

Output:



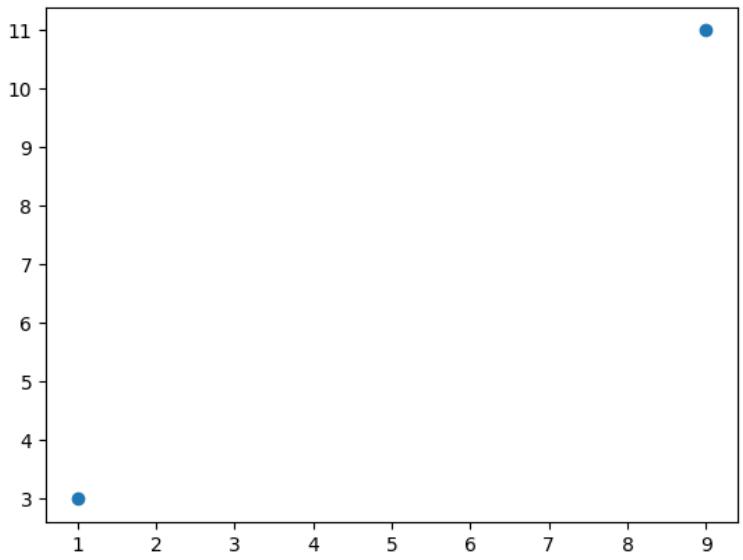
Plotting Without Line

To plot only the markers, you can use shortcut string notation parameter 'o', which means 'rings'.

Code:

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 9])
ypoints = np.array([3, 11])

plt.plot(xpoints, ypoints, 'o')
plt.show()
```

Output:



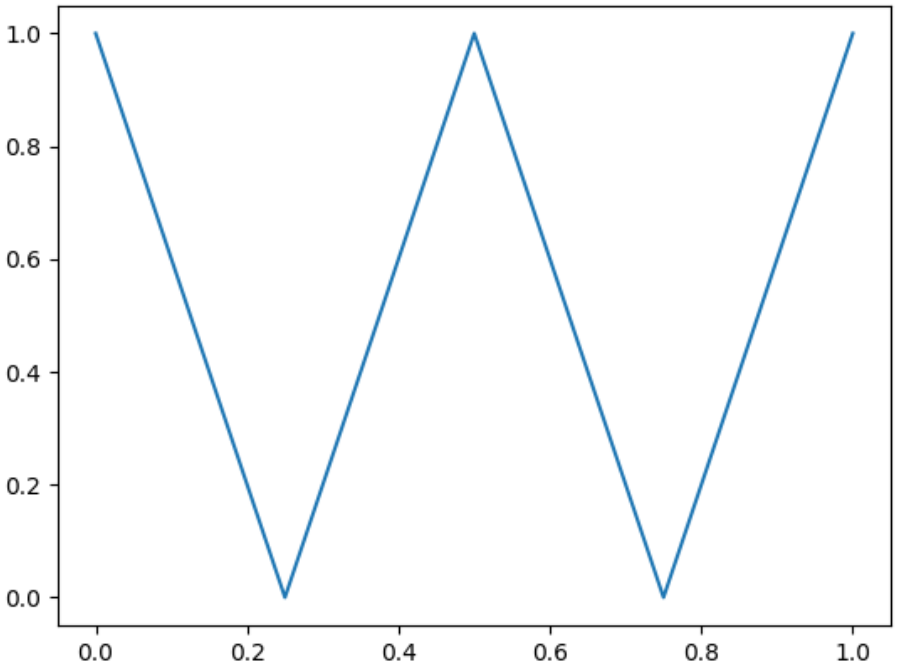
Multiple Points

You can plot as many points as you want, but ensure that both axes have the same number of points.

Code:

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 0.25, 0.5, 0.75, 1])
ypoints = np.array([1, 0, 1, 0, 1])

plt.plot(xpoints, ypoints)
plt.show()
```

Output:



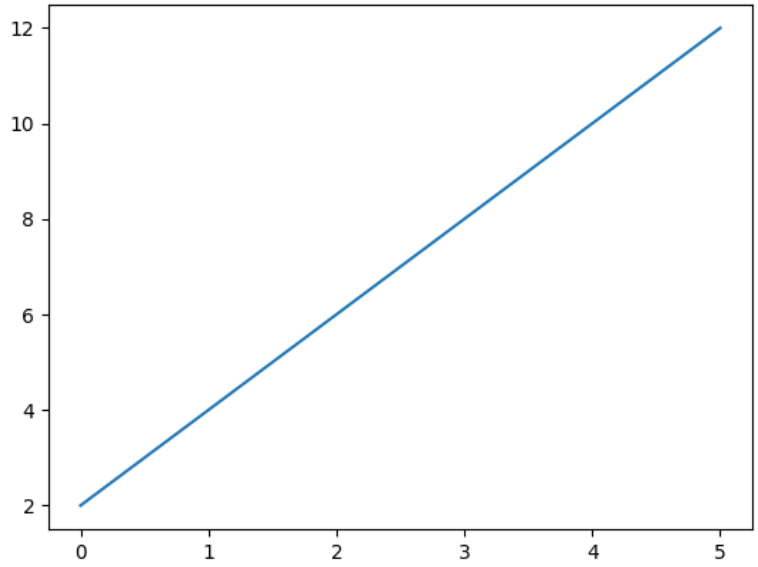
Default X-Points

If we don't specify the points on the x-axis, they will automatically default to 0, 1, 2, 3, and so on, based on the number of y-points.

Code:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([2, 4, 6, 8, 10, 12])

plt.plot(ypoints)
plt.show()
```

Output:



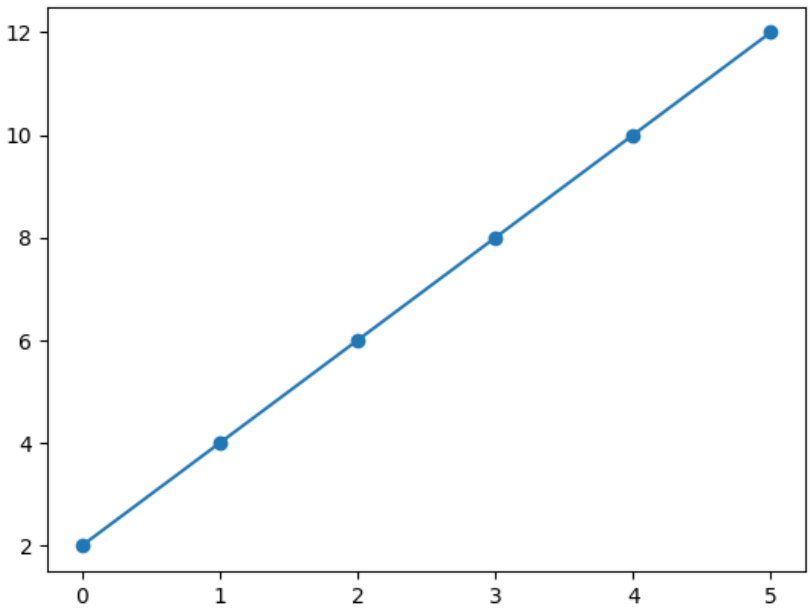
Matplotlib Markers

You can use the keyword argument marker to highlight each point with a designated marker.

Code:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([2, 4, 6, 8, 10, 12])

plt.plot(ypoints, marker = 'o')
plt.show()
```

Output:



Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

Format Strings (fmt)

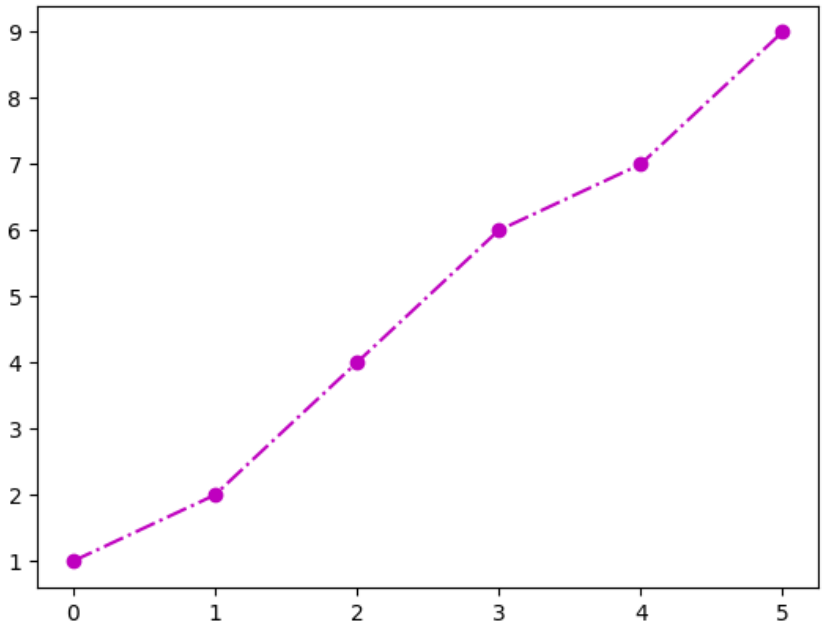
You can also use a shortcut string notation parameter to define the marker. This parameter, known as fmt, follows this syntax:

marker|line|color

Code:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([1, 2, 4, 6, 7, 9])
plt.plot(ypoints, 'o-.m')
plt.show()
```

Output:



Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Marker Size

```
plt.plot(ypoints, marker = 'o', ms = 25)
```

Marker Color

```
plt.plot(ypoints, marker = 'o', ms = 25, mec = 'y')
plt.plot(ypoints, marker = 'o', ms = 25, mfc = 'm')
plt.plot(ypoints, marker = 'o', ms = 25, mec = 'y', mfc = 'm')
```

You can also use hexadecimal color values:

```
plt.plot(ypoints, marker='o', ms=20, mec='#FF5733', mfc='#FF5733')
```

Alternatively, you can choose from the 140 supported color names:

```
plt.plot(ypoints, marker='o', ms=20, mec='skyblue', mfc='skyblue')
```

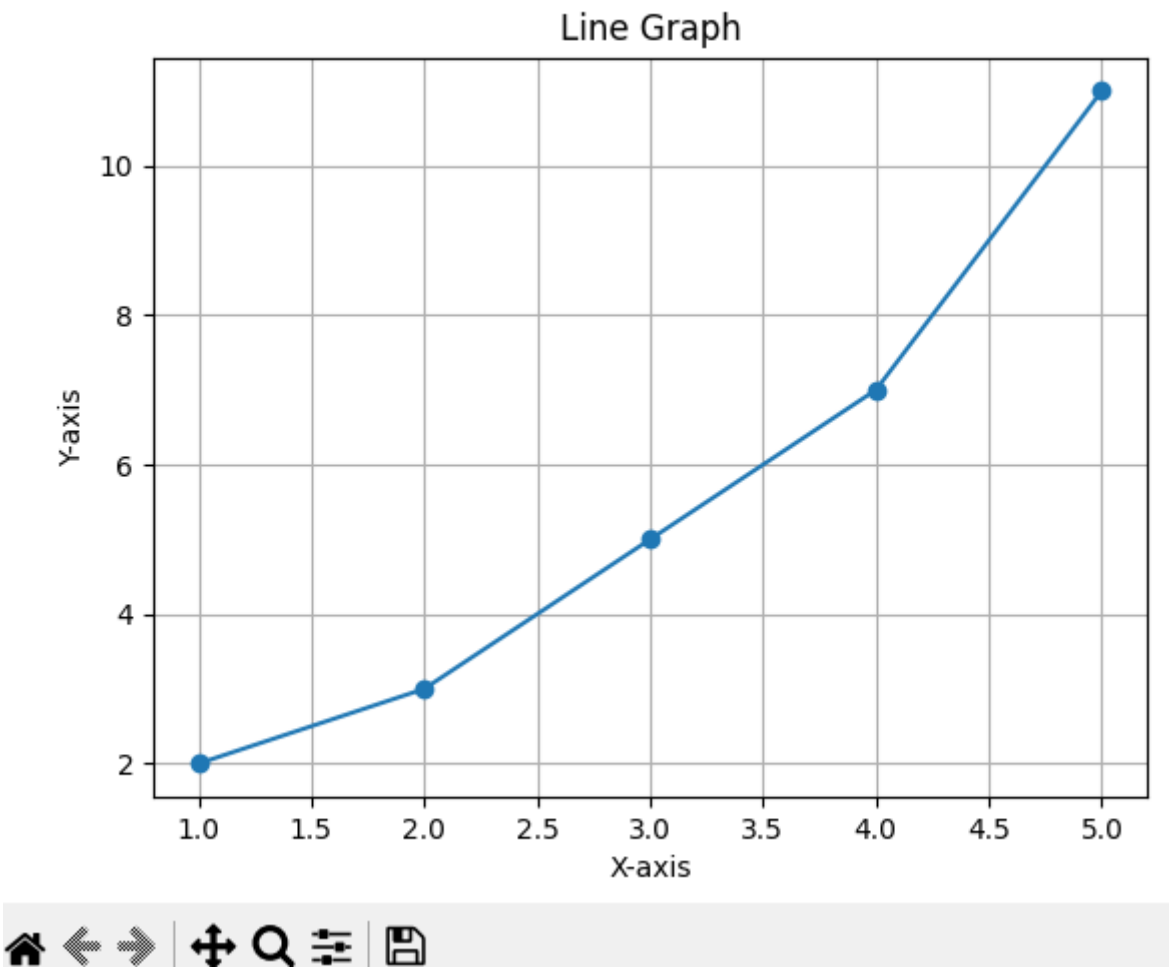
1 Line Graphs

A line graph is useful for showing trends over time.

Code:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

plt.plot(x, y, marker='o')
plt.title('Line Graph')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid()
plt.show()
```

Output:



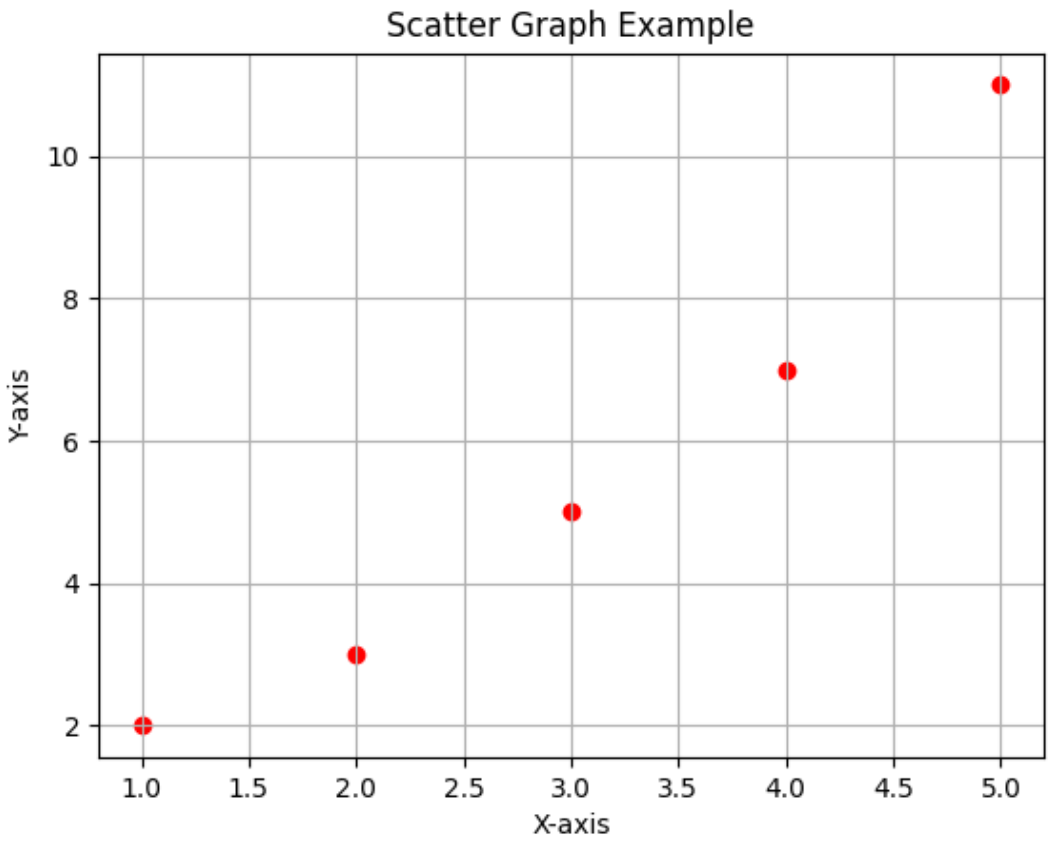
2 Scatter Graph

Scatter plots display values for two variables for a set of data.

Code:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

plt.scatter(x, y, color='red')
plt.title('Scatter Graph Example')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid()
plt.show()
```

Output:



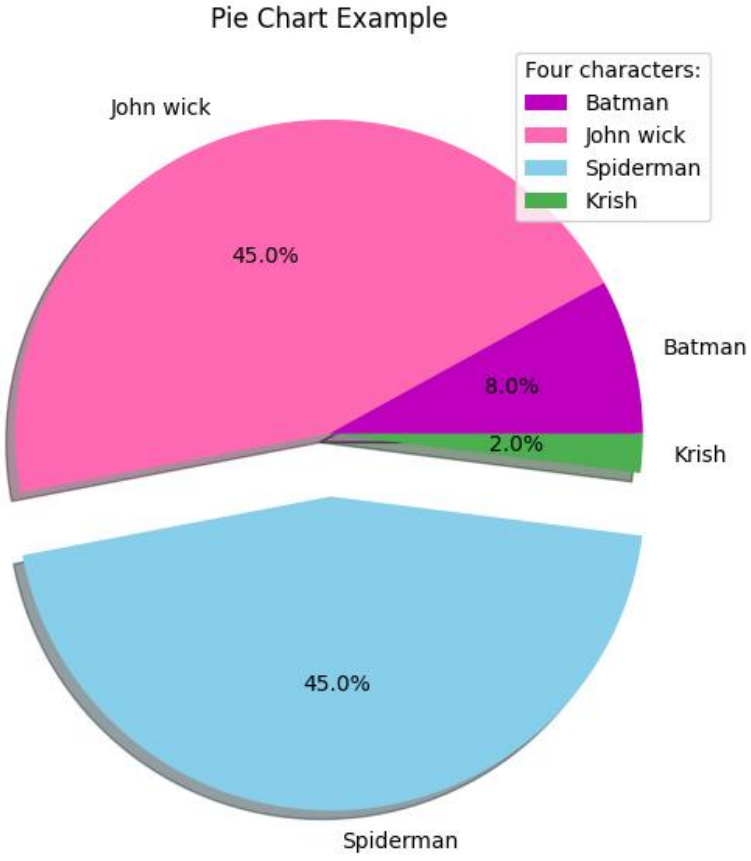
3 Pie Charts

Pie charts show the proportion of parts to a whole.

Code:

```
import matplotlib.pyplot as plt

labels = ['Batman', 'John wick', 'Spiderman', 'Krish']
sizes = [8, 45, 45, 2]
explode = [0, 0, 0.2, 0]
colors = ["m", "hotpink", "skyblue", "#4CAF50"]

plt.pie(sizes, labels=labels,autopct='%1.1f%%', explode = explode, shadow = True,
colors=colors)
plt.title('Pie Chart Example')
plt.legend(title = "Four characters:")
plt.show()
```

Output:



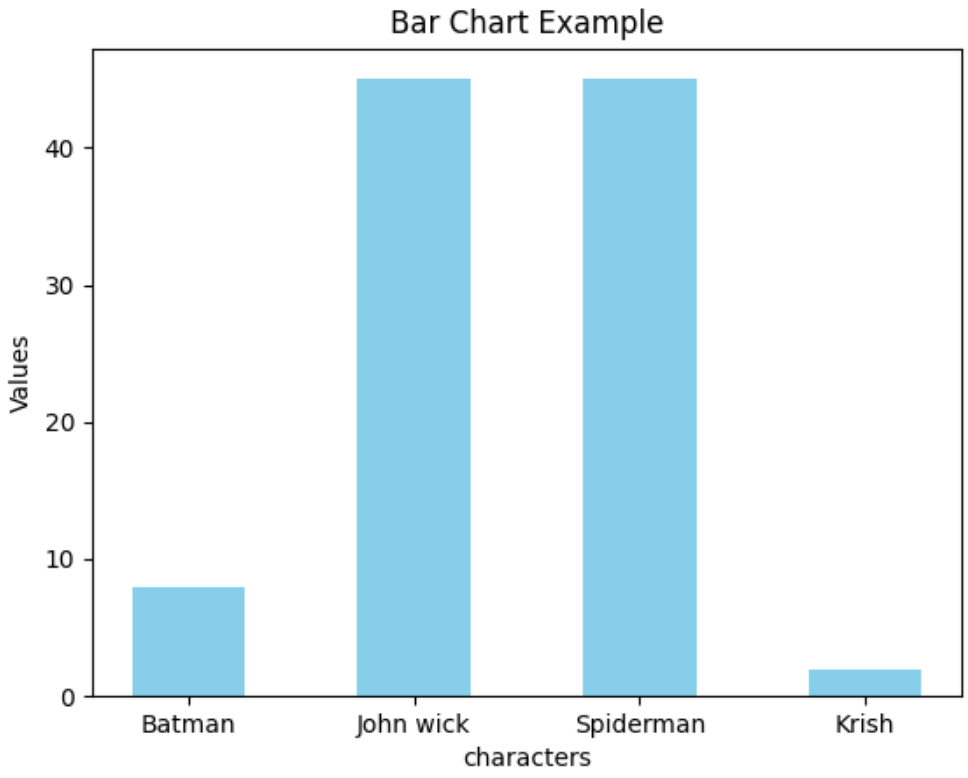
4 Bar Charts

Bar charts are great for comparing quantities across categories.

Code:

```
import matplotlib.pyplot as plt

characters = ['Batman', 'John wick', 'Spiderman', 'Krish']
values = [8, 45, 45, 2]

plt.bar(characters, values, color = "skyblue", width = 0.5)
# Horizontal Bars #plt.barh(characters, values, height = 0.5)
plt.title('Bar Chart Example')
plt.xlabel('characters')
plt.ylabel('Values')
plt.show()
```

Output:



5 Figures and Subplots

You can create multiple plots in a single figure using subplots.

Code:

```
import matplotlib.pyplot as plt
# Data for the plots
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

characters = ['Batman', 'John Wick', 'Spiderman', 'Krish']
values = [8, 45, 45, 2]
explode = [0, 0, 0.2, 0]
colors = ["m", "hotpink", "skyblue", "#4CAF50"]

# Create a figure and a 2x2 grid of subplots
fig, axs = plt.subplots(2, 2, figsize=(10, 8))

# Line Graph
axs[0, 0].plot(x, y, marker='o')
axs[0, 0].set_title('Line Graph')
axs[0, 0].set_xlabel('X-axis')
axs[0, 0].set_ylabel('Y-axis')
axs[0, 0].grid()

# Scatter Plot
axs[0, 1].scatter(x, y, color='red')
axs[0, 1].set_title('Scatter Graph Example')
axs[0, 1].set_xlabel('X-axis')
axs[0, 1].set_ylabel('Y-axis')
axs[0, 1].grid()

# Bar Chart
axs[1, 0].bar(characters, values, color="skyblue", width=0.5)
axs[1, 0].set_title('Bar Chart Example')
axs[1, 0].set_xlabel('Characters')
axs[1, 0].set_ylabel('Values')

# Pie Chart
axs[1, 1].pie(values, labels=characters, autopct='%1.1f%%', explode=explode, shadow=True,
colors=colors)
axs[1, 1].set_title('Pie Chart Example')
axs[1, 1].legend(title="Four Characters:")

# Adjust layout
plt.tight_layout()
plt.show()
```

Output:

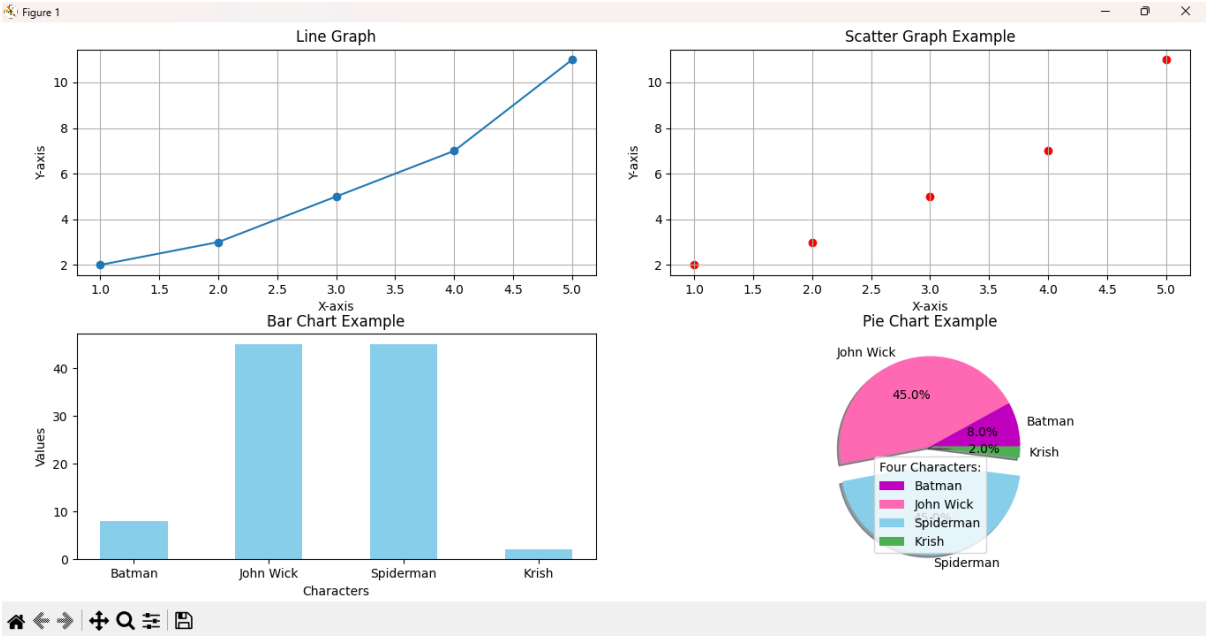


Figure 1
Line Graph
Y-axis
X-axis
Scatter Graph Example
Y-axis
X-axis
Bar Chart Example
Values
Batman
John Wick
Spiderman
Krish
Characters
Pie Chart Example
John Wick
45.0%
Batman
8.0%
Krish
2.0%
Four Characters:
Batman
John Wick
Spiderman
Krish
Spiderman

6.6 Scikitlearn: Linear regression, K-Nearest neighbour classifier, Logistic regression, Decision Tree, Random Forest classifier Clustering and anomaly detection

Scikit-learn is a popular open-source machine learning library for Python that provides simple and efficient tools for data mining and data analysis. It is built on top of NumPy, SciPy, and Matplotlib, making it well-suited for scientific computing.

Common Use Cases:

- → Predictive modeling
- → Customer segmentation
- → Image recognition
- → Natural language processing
- → Time series analysis

Installation of scikit-learn

```
pip install scikit-learn
```

Checking scikit-learn Version

Code:

```
import sklearn
print(sklearn.__version__)
```

Output:

```
1.5.2
```

Regression

The term regression is used when you try to find the relationship between variables.

Linear Regression

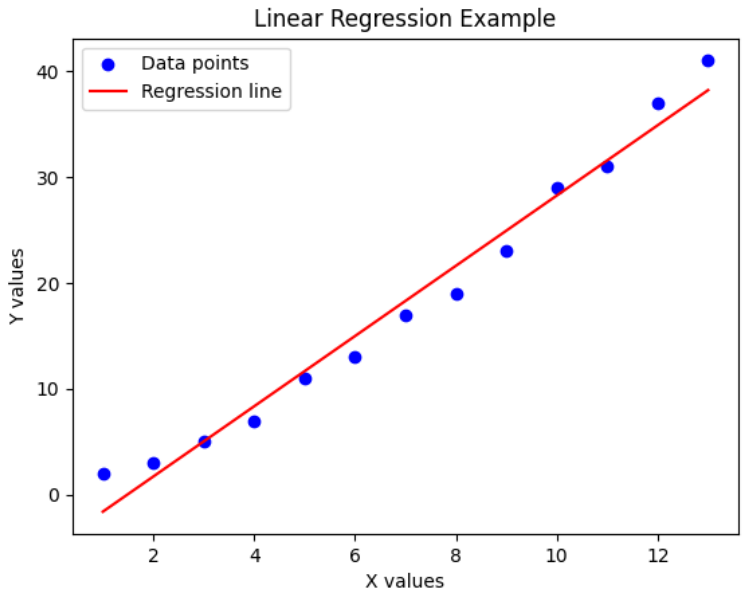
→ Linear regression uses the relationship between the data-points to draw a straight line through all them.

→ This line can be used to predict future values.

Code:

```
import matplotlib.pyplot as plt
from scipy import stats

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
y = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y, color='blue', label='Data points')
plt.plot(x, mymodel, color='red', label='Regression line')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('Linear Regression Example')
plt.legend()
plt.show()
```

Output:



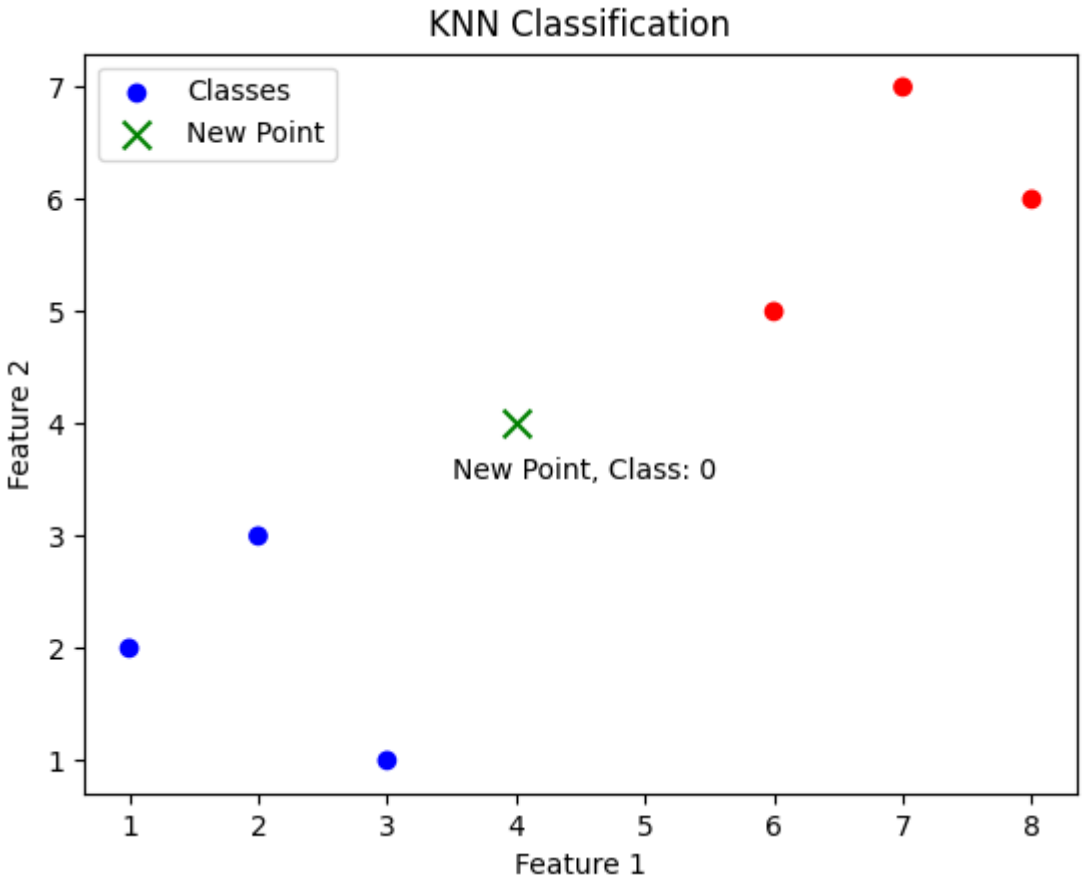
K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a straightforward supervised machine learning algorithm suitable for both classification and regression tasks, as well as for imputation of missing values. The core principle of KNN is that the observations nearest to a specific data point are the most "similar" within the dataset. Consequently, we can classify new points based on the values of their closest existing neighbors. By selecting the parameter K, users can determine how many nearby observations to consider in the algorithm.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

# Sample data
data = np.array([[1, 2], [2, 3], [3, 1], [6, 5], [7, 7], [8, 6]])
classes = np.array([0, 0, 0, 1, 1, 1])  # Two classes: 0 and 1

# Create the KNN classifier
knn = KNeighborsClassifier(n_neighbors=3)

# Fit the model
knn.fit(data, classes)

# New point to classify
new_point = np.array([[4, 4]])
prediction = knn.predict(new_point)

# Prepare data for plotting
x, y = data[:, 0], data[:, 1]
new_x, new_y = new_point[0]

# Plot the data points and the new point
plt.scatter(x, y, c=classes, cmap='bwr', label='Classes')
plt.scatter(new_x, new_y, c='green', marker='x', s=100, label='New Point')
plt.text(new_x - 0.5, new_y - 0.5, f"New Point, Class: {prediction[0]}", fontsize=10)
plt.title("KNN Classification")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

Output:


KNN Classification
Classes
New Point
New Point, Class: 0
Feature 1
Feature 2
1
2
3
4
5
6
7
8

Logistic regression

Logistic regression is designed to address classification problems by predicting categorical outcomes, in contrast to linear regression, which predicts continuous values.

In its simplest form, logistic regression deals with two possible outcomes, known as binomial classification. An example of this would be determining whether an email is spam or not. In scenarios where there are more than two categories to classify, the method is referred to as multinomial logistic regression. A common example of this is predicting the type of fruit—such as apple, orange, or banana—based on various features like color and size.

Code:

```
import numpy
from sklearn import linear_model

# New input data
X = numpy.array([1.0, 2.5, 3.2, 4.1, 5.0, 1.8, 2.9, 3.6, 4.3, 5.5, 0.5, 3.0]).reshape(-1, 1)
y = numpy.array([0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0])

logr = linear_model.LogisticRegression()
logr.fit(X, y)

def logit2prob(logr, X):
    log_odds = logr.coef_ * X + logr.intercept_
    odds = numpy.exp(log_odds)
    probability = odds / (1 + odds)
    return probability

print(logit2prob(logr, X))
```

Output:

```
[[0.021369  ]
 [0.16963374]
 [0.36708002]
 [0.6892957 ]
 [0.89458185]
 [0.06712677]
 [0.27052477]
 [0.51287199]
 [0.74931466]
 [0.94703732]
 [0.01025641]
 [0.30092523]]
```

Decision Tree

A Decision Tree is a popular and intuitive machine learning model used for both classification and regression tasks.

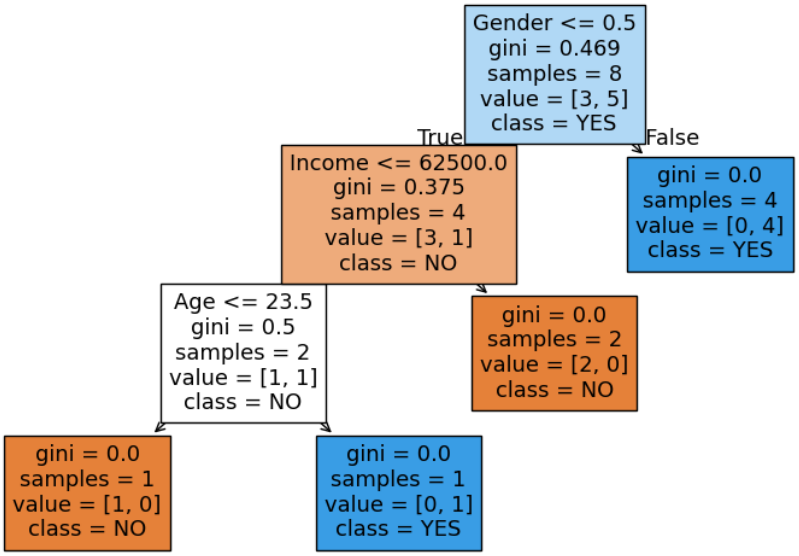
Code:

```
import pandas as pd
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

# Load data from CSV
df = pd.read_csv("customers.csv")

# Map categorical variables to numeric
df['Gender'] = df['Gender'].map({'Male': 0, 'Female': 1})
df['MaritalStatus'] = df['MaritalStatus'].map({'Single': 0, 'Married': 1, 'Divorced': 2})
df['Buy'] = df['Buy'].map({'YES': 1, 'NO': 0})

features = ['Age', 'Income', 'Gender', 'MaritalStatus']

X = df[features]
y = df['Buy']

# Create and fit the Decision Tree model
dtree = DecisionTreeClassifier(random_state=0)
dtree.fit(X, y)

# Plot the Decision Tree
plt.figure(figsize=(10, 6))
tree.plot_tree(dtree, feature_names=features, class_names=['NO', 'YES'], filled=True)

# Save the plot to a file
plt.savefig("decision_tree_plot.png")
plt.close()  # Close the plot to avoid display if running in a script
```

customers.csv

```
Age,Income,Gender,MaritalStatus,Buy
25,50000,Male,Single,YES
30,60000,Female,Married,YES
22,40000,Male,Single,NO
35,70000,Female,Married,YES
40,80000,Male,Divorced,NO
29,65000,Female,Single,YES
45,75000,Male,Married,NO
50,90000,Female,Divorced,YES
```

Output:



6.7 What are the reasons Python is considered the top choice for data visualization?

Python is a favored option for data visualization due to several compelling factors:

1 Rich Ecosystem of Libraries

→ Python boasts a variety of powerful libraries designed for data visualization, including:
→ Matplotlib: A foundational library for creating static, animated, and interactive plots.
→ Seaborn: Built on Matplotlib, it provides a higher-level interface and is excellent for statistical graphics.
→ Plotly: Enables the creation of interactive plots that can be easily shared.
→ Bokeh: Focuses on interactive and web-ready visualizations.
→ Altair: Offers a declarative statistical visualization framework, making it easy to generate complex visualizations.

2 Ease of Use

→ Python's syntax is clear and intuitive, which lowers the learning curve for beginners. The libraries often have simple functions that make it easy to generate complex visualizations with minimal code.

3 Integration with Data Science Stack

→ Python integrates seamlessly with data analysis libraries like Pandas and NumPy. This allows for efficient data manipulation and transformation before visualization, facilitating a smooth workflow from data gathering to analysis to visualization.

4 Support for Large Datasets

→ Libraries like Bokeh and Plotly can handle large datasets and provide tools for zooming, panning, and interactive exploration, making them suitable for big data applications.

5 Cross-Platform Compatibility

→ Python runs on various operating systems (Windows, macOS, Linux) and can be used in different environments, including Jupyter notebooks, web applications, and integrated development environments (IDEs).

6 Community Support

→ Python has a large, active community that contributes to a wealth of resources, tutorials, and documentation. This support makes it easier for users to find solutions to problems and learn best practices.

7 Flexibility and Customization

→ Python libraries allow for extensive customization of visualizations. Users can modify almost every aspect of their charts, from colors and labels to interactive features, enabling the creation of tailored visualizations that fit specific needs.

8 Reproducibility

→ Using Python scripts for visualization supports reproducibility in research and analysis. Users can share their code, making it easy to replicate and validate results.

9 Integration with Machine Learning

→ As data science increasingly incorporates machine learning, Python's visualization libraries can be used to visualize model performance, feature importance, and other insights from machine learning workflows.