Python

## Python Topics -

1. Basic syntax and data types
2. Control structures (if, else, loops)
3. Functions and modules
4. Object-oriented programming (classes and objects)
5. Exception handling
6. File handling (reading and writing files)
7. Regular expressions
8. Working with databases (SQL)
9. Web scraping
10. Networking (sockets and protocols)
11. Multithreading and multiprocessing
12. GUI programming (using frameworks like Tkinter, PyQt, or PySide)
13. Data visualisation (using libraries like Matplotlib or Seaborn)
14. Machine learning (using libraries like Scikit-learn, TensorFlow, or Keras)
15. Deep learning (using libraries like PyTorch or TensorFlow)
16. Natural Language Processing (using libraries like NLTK or spaCy)
17. Image Processing (using libraries like OpenCV or Pillow)
18. Scientific computing (using libraries like NumPy, SciPy, or Pandas)
19. Data analysis and data manipulation (using libraries like Pandas, NumPy, or SciPy)
20. Data Science tools (using libraries like Scikit-learn, Pandas, Matplotlib, Seaborn, etc.)
21. Web development (using frameworks like Flask or Django)
22. Building APIs (using frameworks like Flask or FastAPI)
23. Testing and debugging
24. Code optimization
25. Memory management
26. Concurrency and parallelism
27. Performance profiling
28. Debugging and profiling tools
29. Best practices and code style
30. Optimization and performance techniques
31. Security and cryptography

## 

## 

## Variable declaration in python

1. Use descriptive variable names that are easy to read and understand. For example, use ‘**num\_of\_students**’ instead of ‘**n**’, ‘**students**’ or ‘**s**’.

2. Avoid using reserved keywords as variable names. For example, don't use ‘**print**’ or ‘**list**’ as variable names.

3. Use lower case letters for variable names, with words separated by underscores. This is known as **snake\_case**. For example, ‘**my\_variable\_name**’ is better than ‘**MyVariableName**’ or ‘**myvariablename**’.

4. Use meaningful variable names that reflect the purpose of the variable. For example, ‘**user\_age**’ is better than ‘**age**’ or ‘**variable1**’.

5. Declare variables close to where they are first used. This helps to reduce confusion and makes it easier to understand the purpose of the variable.

6. Use constants for values that don't change. Constants are usually defined in all capital letters. For example, ‘**PI = 3.14159**’.

7. Don't reuse variable names for different purposes in the same block of code. This can lead to confusion and errors.

8. Avoid using global variables unless absolutely necessary. Global variables can make code difficult to understand and debug.

9. Use variable type annotations to indicate the type of a variable. For example, ‘**num\_students: int = 20**’. This can help with code clarity and error checking.

Remember that these are just general best practices, and there may be cases where they don't apply. Always use your best judgement when naming and declaring variables.

In Python, variables that start with a **single underscore (\_)** or **double underscore (\_\_)** have a special meaning. These variables are known as "**special variables**" or "**magic variables**". Here are some examples of special variables in Python:

1. **\_ (single underscore)**: The single underscore is used as a placeholder for variables that are not used in the code. For example, if you want to ignore a specific value returned by a function, you can use the single underscore as a variable name:

**\_ , second\_largest = sorted([5, 10, 2, 8, 3])**

Here, the single underscore is used as a placeholder for the smallest value in the list.

2. **\_\_ (double underscore)**: The double underscore is used for name mangling in Python classes. Name mangling is a technique that makes a variable or method private to a class. For example:

class MyClass:

def \_\_init\_\_(self):

self.\_\_my\_private\_variable = 42

def get\_my\_private\_variable(self):

return self.\_\_my\_private\_variable

In this example, the double underscore before the variable name makes it "private" to the class. It can still be accessed by calling the ‘**get\_my\_private\_variable**’ method.

3. **variable (double underscore on both sides)**: This type of variable is used for special methods or attributes in Python. For example, the ‘**\_\_init\_\_**’ method is called when an object is created from a class:

class MyClass:

def \_\_init\_\_(self):

self.my\_variable = 42

In this example, ‘**\_\_init\_\_**’ is a special method that is called when an object of the ‘**MyClass class**’ is created.

Keep in mind that while these special variables have specific meanings in Python, they are not enforced by the language. For example, you can still access a "private" variable in Python by using its name, even if it starts with a double underscore. However, it is considered best practice to follow the conventions of the language and use these special variables as intended.

4. Many Values to Multiple Variables

**x, y, z = "Orange", "Banana", "Cherry"**

5. One Value to Multiple Variables

**x = y = z = "Orange"**

6. Unpack a Collection

**fruits = ["apple", "banana", "cherry"]**

**x, y, z = fruits**

7. You can also use the ‘**+**’ operator to output multiple variables:

x = "Python "

y = "is "

z = "awesome"

**print(x + y + z)**

Following will give error as string and int are combined

x = 5

y = "John"

**print(x + y)**

## Variable Scoping -

In Python 3, variable scoping refers to the rules that determine where in your code a particular variable can be accessed and modified. Python 3 has two main types of variable scopes: **global** and **local**.

Global variables are defined outside of any function or class definition, and they can be accessed from anywhere in your code. Here's an example:

**x = 10**

**def print\_x():**

**print(x)**

**print\_x() # Output: 10**

In this example, '**x**' is a global variable because it is defined outside of any function. The function **print\_x()** can access **x** and print its value, because **x** is defined in a global scope.

Local variables are defined within a function or class definition, and they can only be accessed from within that function or class. Here's an example:

**def add\_numbers(a, b):**

**c = a + b**

**print(c)**

**add\_numbers(2, 3) # Output: 5**

In this example, **a**, **b**, and **c** are all local variables because they are defined within the **add\_numbers()** function. They cannot be accessed from outside the function.

It's important to note that if you try to modify a global variable from within a function, you need to use the global keyword to indicate that you're referring to the global variable and not creating a new local variable with the same name. Here's an example:

**x = 10**

**def modify\_x():**

**global x**

**x = 20**

**modify\_x()**

**print(x) # Output: 20**

In this example, **modify\_x()** modifies the global variable x by using the global keyword to indicate that it's referring to the global variable and not creating a new local variable with the same name.

In summary, variable scoping in Python 3 determines where in your code a particular variable can be accessed and modified. Global variables can be accessed from anywhere in your code, while local variables can only be accessed from within the function or class in which they are defined. If you need to modify a global variable from within a function, you need to use the global keyword to indicate that you're referring to the global variable and not creating a new local variable with the same name.

## Difference between Data Types and Data Structures -

Data types and data structures are both fundamental concepts in computer programming, but they are distinct from each other.

In Python, a data type is a classification of data that determines the type of operations that can be performed on it. Python has several built-in data types such as integers, floats, strings, and booleans. Each data type has its own set of operations that can be performed on it. For example, arithmetic operations can be performed on integer and float data types, but not on string data types.

On the other hand, a data structure is a way of organising and storing data in a computer so that it can be accessed and used efficiently. Python provides several built-in data structures such as lists, tuples, dictionaries, and sets. Each data structure has its own set of methods and operations that can be performed on it. For example, the append() method can be used to add an element to the end of a list, and the keys() method can be used to retrieve all the keys in a dictionary.

In summary, while data types determine the type of data that can be stored and the operations that can be performed on them, data structures are a way of organising and storing data to make it more efficient to access and use.

## Data Types -

| 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 | None |

1. **Integer**: Integer data type represents whole numbers (positive, negative, or zero) without a fractional part. For example: '**x = 5**'. This data type is used to store whole numbers. Use cases for integers include counting items, tracking inventory, or representing a numerical value in a calculation.

2. **Float**: Float data type represents numbers with a decimal point. For example: '**y = 3.14**'. Floats can also be scientific numbers with an "e" to indicate the power of 10.Folat takes more memory in space than int.This data type is used to store decimal numbers. Use cases for floats include representing measurements, calculating percentages, or representing a value with a fractional component

3. **Complex**: Complex numbers are written with a "**j**" as the imaginary part **‘x = 3+5j’**

4. **Boolean**: Boolean data type represents the truth values "**True**" and "**False**". For example: '**a = True**'. This data type is used to represent true or false values. Use cases for booleans include conditional statements, decision-making, or checking whether a condition is true or false.

5. **String**: String data type represents a sequence of characters. Strings are enclosed in single or double quotes. For example: '**s = "Hello, World!”.** This data type is used to store text. Use cases for strings include representing names, addresses, or other pieces of text data.

6. **List**: List data type represents an ordered collection of items. Lists are enclosed in square brackets and each item is separated by a comma. For example: '**l = [1, 2, 3, 4, 5]**'. This data type is used to store a collection of items. Use cases for lists include storing a series of values, representing a set of items, or building a queue

7. **Tuple**: Tuple data type is similar to lists, but it is immutable (cannot be modified). Tuples are enclosed in parentheses and each item is separated by a comma. For example: '**t = (1, 2, 3, 4, 5)**'. This data type is similar to a list, but it is immutable, meaning that its values cannot be changed once it is created. Use cases for tuples include representing a fixed set of data, building a dictionary, or returning multiple values from a function.

8. **Set**: Set data type represents an unordered collection of unique items. Sets are enclosed in curly braces and each item is separated by a comma. For example: '**s = {1, 2, 3, 4, 5}**'. This data type is used to store an unordered collection of unique elements. Use cases for sets include eliminating duplicates from a list, performing set operations such as union, intersection, and difference, or checking for membership of an element in a collection.

9. **Dictionary**: Dictionary data type represents a collection of key-value pairs. Dictionaries are enclosed in curly braces and each key-value pair is separated by a colon. For example: '**d = {"name": "John", "age": 30}**'. This data type is used to store key-value pairs. Use cases for dictionaries include storing metadata, building a lookup table, or representing complex data structures.

10. **NoneType**: NoneType data type represents a variable that has no value assigned to it. For example: '**n = None**'

These are the basic Python data types that you will use in most programming tasks. There are also advanced data types such as sets, frozen sets, and named tuples, but these are less commonly used. It is often used to indicate that a variable or function argument does not have a value, or that a function call does not return anything meaningful.

**Type Conversion** -

You can convert from one type to another with the **int()**, **float()**, and **complex()** methods:

Example

x = 1 # int

y = 2.8 # float

z = 1j # complex

a = float(x) # Convert from int to float

b = int(y) # Convert from float to int

c = complex(x) # Convert from int to complex

**Random Number** -

Python does not have a **random()** function to make a random number, but Python has a built-in module called random that can be used to make random numbers

Example -

**import random**

**print(random.randrange(1, 10))**

**Math** - The Python math library is a built-in module in Python that provides a wide range of mathematical functions. These functions include mathematical constants such as **pi**, **e**, and **infinity**, as well as a range of mathematical operations like **trigonometric**, **logarithmic**, and **exponential** functions. Here are a few examples of using the math library in Python:

1. Finding the square root of a number:

**import math**

**number = 25**

**sqrt\_number = math.sqrt(number)**

**print(sqrt\_number)**

2. Finding the value of pi:

**import math**

**pi\_value = math.pi**

**print(pi\_value)**

https://www.programiz.com/python-programming/modules/math

**Python Casting** **/ Type conversion** -

Type conversion is the process of converting an object of one data type to another data type, while casting specifically refers to changing the type of a variable from one type to another using constructor functions.

There may be times when you want to specify a type of a variable. This can be done with casting. Python is an object-oriented language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

* **int()** - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
* **float()** - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
* **str()** - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Integers:

x = int(1) # x will be 1

y = int(2.8) # y will be 2

z = int("3") # z will be 3

Floats:

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

Strings:

x = str("s1") # x will be 's1'

y = str(2) # y will be '2'

z = str(3.0) # z will be '3.0'

# converting string to integer

number\_str = '5'

number\_int = int(number\_str)

print(number\_int)

# converting integer to float

number\_int = 5

number\_float = float(number\_int)

print(number\_float)

# converting integer to string

number\_int = 5

number\_str = str(number\_int)

print(number\_str)

# converting boolean to integer

value\_bool = True

value\_int = int(value\_bool)

print(value\_int)

The output of this code would be:

5

5.0

5

1

It is important to note that not all types can be converted to all other types. For example, you cannot convert a string that does not represent a valid number to an integer or float. Additionally, some conversions may result in loss of precision or rounding errors, especially when converting between floating-point and integer types. Therefore, it is important to carefully consider the data types involved and the potential impact of type conversion on your program's behaviour.

## Specialized Data Types in Python -

Here are all the specialised data types in Python 3 with examples:

1. **Lists**: A list is a collection of items that can be of any data type. Lists are mutable, which means you can add, remove, or modify elements in the list.

Example:

**fruits = ['apple', 'banana', 'cherry']**

2. **Tuples**: A tuple is similar to a list, but it is immutable, which means you cannot modify its contents once it is created. Tuples are commonly used to store related data that should not be changed.

Example:

**person = ('John', 25, 'Male')**

3. **Dictionaries**: A dictionary is a collection of key-value pairs. The keys are unique and used to access the corresponding values. Dictionaries are commonly used to store and retrieve data in a structured way.

Example:

**person = {'name': 'John', 'age': 25, 'gender': 'Male'}**

4. **Sets**: A set is an unordered collection of unique elements. Sets are useful when you need to perform operations such as intersection, union, and difference on collections of data.

Example:

**fruits = {'apple', 'banana', 'cherry'}**

5. **Strings**: A string is a sequence of characters. Strings are immutable in Python, which means you cannot modify them once they are created. Strings are used to represent text data.

Example:

**greeting = 'Hello, World!'**

6. **Bytes**: A bytes object is a sequence of bytes. Bytes are used to represent binary data, such as image or audio files.

Example:

**data = b'\x00\xFF\x0A'**

7. **Byte Arrays**: A bytearray object is similar to a bytes object, but it is mutable, which means you can modify its contents.

Example:

**data = bytearray(b'\x00\xFF\x0A')**

To find out data type in python user **type(variable)** function

## Operators in Python -

1. Arithmetic Operators:

Addition (+): Adds two operands.

Example: 2 + 3 returns 5.

Subtraction (-): Subtracts the second operand from the first operand.

Example: 3 - 2 returns 1.

Multiplication (\*): Multiplies two operands.

Example: 2 \* 3 returns 6.

Division (/): Divides the first operand by the second operand.

Example: 6 / 3 returns 2.

Modulo (%): Returns the remainder after division of the first operand by the second operand.

Example: 7 % 3 returns 1.

Floor Division (//): Returns the quotient after division of the first operand by the second operand, rounding down to the nearest integer.

Example: 7 // 3 returns 2.

Exponentiation (\*\*): Raises the first operand to the power of the second operand.

Example: 2 \*\* 3 returns 8.

Operator Precedence - Operator precedence is the order in which operations are evaluated in an expression. In programming languages, mathematical expressions often involve multiple operators, and operator precedence determines the order in which these operators are evaluated.

2. Assignment Operators:

Simple Assignment (=): Assigns the value of the right-hand side to the left-hand side.

Example: x = 5 assigns the value 5 to the variable x.

Add and Assignment (+=): Adds the value of the right-hand side to the value of the left-hand side, and assigns the result to the left-hand side.

Example: x += 3 is equivalent to x = x + 3.

Subtract and Assignment (-=): Subtracts the value of the right-hand side from the value of the left-hand side, and assigns the result to the left-hand side.

Example: x -= 3 is equivalent to x = x - 3.

Multiply and Assignment (\*=): Multiplies the value of the right-hand side with the value of the left-hand side, and assigns the result to the left-hand side.

Example: x \*= 3 is equivalent to x = x \* 3.

Divide and Assignment (/=): Divides the value of the left-hand side by the value of the right-hand side, and assigns the result to the left-hand side.

Example: x /= 3 is equivalent to x = x / 3.

Modulo and Assignment (%=): Calculates the modulo of the left-hand side with the right-hand side, and assigns the result to the left-hand side.

Example: x %= 3 is equivalent to x = x % 3.

3. Comparison Operators:

Equal to (==): Returns True if the values of the two operands are equal.

Example: 2 == 2 returns True.

Not equal to (!=): Returns True if the values of the two operands are not equal.

Example: 2 != 3 returns True.

Greater than (>): Returns True if the value of the left-hand side is greater than the value of the right-hand side.

Example: 3 > 2 returns True.

Less than (<): Returns True if the value of the left-hand side is less than the value of the right-hand side.

Example: 2 < 3 returns True.

Greater than or equal to (>=): Returns True if the value of the left-hand side is greater than or equal to the value of the right-hand side.

Example: 3 >= 2 returns True.

Less than or equal to (<=): Returns True if the value of the left-hand side is less than or equal to the value of the right-hand side.

Example: 2 <= 3 returns True.

4. Logical Operators:

Logical AND (and):

Returns True if both the left-hand side and the right-hand side are True. Otherwise, it returns False. For example, if the left-hand side is True and the right-hand side is False, the expression returns False. However, if both the left-hand side and the right-hand side are True, the expression returns True.

Logical OR (or):

Returns True if either the left-hand side or the right-hand side is True. If both the left-hand side and the right-hand side are False, the expression returns False.

For example, if the left-hand side is False and the right-hand side is True, the expression returns True. However, if both the left-hand side and the right-hand side are False, the expression returns False.

Logical NOT (not):

Returns the opposite of the Boolean value of the operand. If the operand is True, it returns False. If the operand is False, it returns True. For example, if the operand is True, the expression returns False. However, if the operand is False, the expression returns True.

## Expression vs Statement -

In Python, an expression is a piece of code that produces a value, whereas a statement is a complete unit of execution that performs an action. An expression can be part of a statement or stand alone, but a statement cannot be part of an expression.

For example, the following line of code is an expression:

3 + 4

This expression evaluates to the value 7. It can be used as part of a larger statement, such as an assignment statement:

x = 3 + 4

Here, the expression 3 + 4 is evaluated first, producing the value 7, which is then assigned to the variable x.

On the other hand, the following line of code is a statement:

print("Hello, World!")

This statement performs an action, which is to print the message "Hello, World!" to the console. It does not produce a value that can be used in another expression.

In general, expressions are used to compute values, whereas statements are used to perform actions. It is important to distinguish between expressions and statements in Python, as they have different syntax and rules for use.

## Python Strings -

**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')

**Multiline Strings**

You can assign a multiline string to a variable by using three quotes ‘’’ or ”””

a = """Lorem ipsum dolor sit amet,

consectetur adipiscing elit,

sed do eiusmod tempor incididunt

ut labore et dolore magna aliqua."""

print(a)

**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.

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

a = "Hello, World!"

print(a[1])

String concatenation - String concatenation in Python is the process of combining two or more strings together to create a single, longer string. In Python, string concatenation is performed using the **+** operator.

For example, consider the following code:

first\_name = "John"

last\_name = "Doe"

full\_name = first\_name + " " + last\_name

print(full\_name)

In this example, we have two string variables: first\_name and last\_name. We can concatenate these two strings to create a full name by using the **+** operator to combine them along with a space character. The result of this operation is assigned to a new variable **full\_name**, which we then print to the console.

The output of this code will be:

John Doe

String concatenation can also be used to combine string literals and variables:

greeting = "Hello, "

name = "John"

message = greeting + name + "!"

print(message)

In this example, we have a string variable **greeting** and a string variable **name**. We can concatenate these two strings along with an exclamation mark to create a personalised message. The result of this operation is assigned to a new variable message, which we then print to the console.

The output of this code will be:

Hello, John!

String concatenation is a useful technique for combining strings in Python and is used frequently in text processing, data manipulation, and many other applications.

print(‘Hello’ + 5)

It is important to note that not all types can be converted to all other types. For example, you cannot convert a string that does not represent a valid number to an integer or float. Additionally, some conversions may result in loss of precision or rounding errors, especially when converting between floating-point and integer types. Therefore, it is important to carefully consider the data types involved and the potential impact of type conversion on your program's behaviour.

**Escape sequence** -

In Python, an escape sequence is a combination of characters that represents a special character or sequence of characters. Escape sequences are commonly used to represent characters that cannot be directly entered into a string or to add special formatting to output.

Here are some of the most commonly used escape sequences in Python:

**\n** - newline character, inserts a line break.

**\t** - tab character, inserts a horizontal tab.

**\'** - single quote character, allows a single quote to be included in a string that is delimited by single quotes.

**\"** - double quote character, allows a double quote to be included in a string that is delimited by double quotes.

**\\** - backslash character, allows a backslash to be included in a string.

Here are some examples of using escape sequences in Python:

# using newline and tab characters

print("First line\nSecond line")

print("Column 1\tColumn 2\tColumn 3")

# using single and double quotes

print('He said, "Hello World!"')

print(“She's a Python programmer.")

# using backslash

print("C:\\Users\\John\\Documents")

The output of this code would be:

First line

Second line

Column 1 Column 2 Column 3

He said, "Hello World!"

She's a Python programmer.

C:\Users\John\Documents

It is important to use escape sequences correctly to avoid syntax errors and unexpected behaviour in your program.

**Formatted Strings** -

In Python, a formatted string is a string that contains one or more replacement fields, which are enclosed in curly braces {}. These replacement fields can contain expressions that are evaluated at runtime and inserted into the string in place of the field.

Formatted strings are useful when you need to dynamically generate output that includes variables, values, or other dynamic content. They make it easy to create output that is easy to read and understand, and can be customised to meet your specific needs.

Here is an example of using formatted strings in Python:

name = "Alice"

age = 25

occupation = "Software Engineer"

print(f"My name is {name}, and I am {age} years old. I work as a {occupation}.")

In this example, the **f** prefix before the string indicates that it is a formatted string. The replacement fields **{name}**, **{age}**, and **{occupation}** are enclosed in curly braces and contain expressions that are evaluated at runtime. The resulting output would be:

My name is Alice, and I am 25 years old. I work as a Software Engineer.

You can also use formatted strings to specify formatting options for the values that are inserted into the string. For example, you can specify the number of decimal places to display for a floating-point value, or the width of a field to ensure that values are aligned properly. Here is an example:

pi = 3.14159265358979323846

print(f"The value of pi is approximately {pi:.2f}")

In this example, the **:.2f** format specifier specifies that the floating-point value should be rounded to two decimal places. The resulting output would be:

The value of pi is approximately 3.14

Formatted strings are a powerful feature in Python that make it easy to create output that is dynamic and easy to read. By using them effectively, you can create output that meets your specific needs and makes your code easier to maintain and understand.

In Python, **.format()** is a method that is used to format strings. It provides an alternative way to create formatted strings that is similar to formatted string literals, but can be used in a wider range of situations.

The **.format()** method works by replacing placeholders in a string with values that are passed as arguments. The placeholders are defined using curly braces **{}** with optional format specifiers inside. The arguments are passed to the method as positional or keyword arguments.

Here is an example of using .format() in Python:

name = "Bob"

age = 35

occupation = "Data Scientist"

message = "My name is {}, and I am {} years old. I work as a {}.".format(name, age, occupation)

print(message)

In this example, the string "My name is {}, and I am {} years old. I work as a {}." contains three placeholders {} that are replaced with the values of the name, age, and occupation variables using the .format() method. The resulting output would be:

My name is Bob, and I am 35 years old. I work as a Data Scientist.

You can also specify format options for the placeholders by adding a colon : followed by the format specifier inside the curly braces {}. Here is an example:

pi = 3.14159265358979323846

message = "The value of pi is approximately {:.2f}.".format(pi)

print(message)

In this example, the placeholder {:.2f} specifies that the value of pi should be formatted as a floating-point number with two decimal places. The resulting output would be:

The value of pi is approximately 3.14.

The .format() method is a flexible way to format strings in Python that provides a wide range of options for controlling the output. By using it effectively, you can create output that is easy to read and understand, and meets your specific needs.

String indexes -

In Python, a string is a sequence of characters, and each character in the string has a unique position or index. String indexing is the process of accessing individual characters in a string by their position or index.

In Python, string indexing starts from 0, which means that the first character in a string has an index of 0, the second character has an index of 1, and so on. You can access the individual characters in a string using square brackets [ ] and the index of the character that you want to access.

Here is an example of using string indexing in Python:

message = "Hello, World!"

print(message[0]) # Output: "H"

print(message[4]) # Output: "o"

print(message[-1]) # Output: "!"

In this example, the string "Hello, World!" is assigned to the variable message, and we use string indexing to access the first character "H", the fifth character "o", and the last character "!" using their respective indices 0, 4, and -1 (which refers to the last character in the string).

You can also use slicing to extract a portion of a string. Slicing allows you to specify a range of indices and extract the characters between those indices. The syntax for slicing is

**string[ start : end : step ]**

where start is the index of the first character to include, end is the index of the last character to include (exclusive), and step is the step size (default is 1).

Here is an example of using string slicing in Python:

message = "Hello, World!"

print(message[0:5]) # Output: "Hello"

print(message[7:]) # Output: "World!"

print(message[::2]) # Output: "Hlo ol!"

In this example, we use string slicing to extract the substring "Hello" using the range 0:5, the substring "World!" using the range 7: (which includes all characters from index 7 to the end of the string), and the substring "Hlo ol!" using a step size of 2.

String indexing and slicing are powerful features in Python that make it easy to work with strings and extract the information that you need. By using them effectively, you can manipulate and transform strings to meet your specific needs.

**Immutability** -

Strings in python are immutable

In Python, strings are immutable, which means that once a string is created, you cannot change its contents. Any attempt to modify a string will result in the creation of a new string object.

Here's an example to demonstrate this:

greeting = "Hello, World!"

greeting[7] = "F"

print(greeting)

When you try to run this code, you will get an error message that says "TypeError: 'str' object does not support item assignment". This is because the string greeting is immutable, and you cannot change its contents by assigning a new value to a specific character.

To modify a string, you need to create a new string that includes the changes you want to make. For example:

greeting = "Hello, World!"

new\_greeting = greeting[:7] + "F" + greeting[8:]

print(new\_greeting)

In this example, we create a new string new\_greeting by concatenating three substrings. The first substring is the characters from the start of the original string up to, but not including, the character we want to replace ("Hello, "). The second substring is the new character we want to insert ("F"). The third substring is the characters from the original string starting from the character after the one we replaced ("World!").

This creates a new string that is identical to the original string, except for the replaced character "W" with "F". The original string greeting remains unchanged, because strings are immutable in Python.

**String methods in python -**

| **Sr. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1 | capitalize() | Converts the first character to uppercase |
| 2 | casefold() | Converts string into lower case |
| 3 | center() | Returns a centered string |
| 4 | count() | Returns the number of times a specified value occurs in a string |
| 5 | encode() | Returns an encoded version of the string |
| 6 | endswith() | Returns true if the string ends with the specified value |
| 7 | expandtabs() | Sets the tab size of the string |
| 8 | find() | Searches the string for a specified value and returns the position of where it was found |
| 9 | format() | Formats specified values in a string |
| 10 | format\_map() | Formats specified values in a string |
| 11 | index() | Searches the string for a specified value and returns the position of where it was found |
| 12 | isalnum() | Returns True if all characters in the string are alphanumeric |
| 13 | isalpha() | Returns True if all characters in the string are in the alphabet |
| 14 | isdecimal() | Returns True if all characters in the string are decimals |
| 15 | isdigit() | Returns True if all characters in the string are digits |
| 16 | isidentifier() | Returns True if the string is an identifier |
| 17 | islower() | Returns True if all characters in the string are lower case |
| 18 | isnumeric() | Returns True if all characters in the string are numeric |
| 19 | isprintable() | Returns True if all characters in the string are printable |
| 20 | isspace() | Returns True if all characters in the string are whitespaces |
| 21 | istitle() | Returns True if the string follows the rules of a title |
| 22 | isupper() | Returns True if all characters in the string are upper case |
| 23 | join() | Joins the elements of an iterable to the end of the string |
| 24 | ljust() | Returns a left justified version of the string |
| 25 | lower() | Converts a string into lower case |
| 26 | lstrip() | Returns a left trim version of the string |
| 27 | maketrans() | Returns a translation table to be used in translations |
| 28 | partition() | Returns a tuple where the string is parted into three parts |
| 29 | replace() | Returns a string where a specified value is replaced with a specified value |
| 30 | rfind() | Searches the string for a specified value and returns the last position of where it was found |
| 31 | rindex() | Searches the string for a specified value and returns the last position of where it was found |
| 32 | rjust() | Returns a right justified version of the string |
| 33 | rpartition() | Returns a tuple where the string is parted into three parts |
| 34 | rsplit() | Splits the string at the specified separator, and returns a list |
| 35 | rstrip() | Returns a right trim version of the string |
| 36 | split() | Splits the string at the specified separator, and returns a list |
| 37 | splitlines() | Splits the string at line breaks and returns a list |
| 38 | startswith() | Returns true if the string starts with the specified value |
| 39 | strip() | Returns a trimmed version of the string |
| 40 | swapcase() | Swaps cases, lower case becomes upper case and vice versa |
| 41 | title() | Converts the first character of each word to upper case |
| 42 | translate() | Returns a translated string |
| 43 | upper() | Converts a string into upper case |
| 44 | zfill() | Fills the string with a specified number of 0 values at the beginning |

**Built in functions and methods** -

Python provides a wide range of built-in functions and methods that you can use in your code without having to define them yourself. Here are some commonly used built-in functions and methods in Python:

**print()**: This built-in function is used to print text or variables to the console. For example: print("Hello, World!").

**len()**: This built-in function is used to get the length of a string or list. For example: len("Hello, World!").

**type()**: This built-in function is used to get the data type of a variable. For example: type("Hello, World!").

**int()**, **float()**, **str()**: These built-in functions are used to convert variables to different data types. For example: int("42") converts the string "42" to an integer.

**Booleans -**

In Python, a boolean is a data type that has two possible values: True and False. Booleans are used to represent truth values or logical values.

Here are some important things to note about booleans in Python:

True and False are the only two boolean values in Python.

Booleans are often used in conditional statements to determine the flow of a program based on whether a condition is true or false.

Booleans can be combined using logical operators such as and, or, and not to create more complex boolean expressions.

Boolean expressions can be evaluated to True or False using comparison operators such as

== (equal to),

!= (not equal to),

< (less than),

> (greater than),

<= (less than or equal to)

>= (greater than or equal to).

Here are some examples of using booleans in Python:

# Assigning boolean values to variables

x = True

y = False

# Using boolean values in conditional statements

if x:

print("x is true")

if not y:

print("y is not true")

# Combining booleans with logical operators

if x and not y:

print("x is true and y is not true")

# Evaluating boolean expressions with comparison operators

if 10 > 5:

print("10 is greater than 5")

if "hello" == "world":

print("This won't be printed")

Output:

x is true

y is not true

x is true and y is not true

10 is greater than 5

**Commenting in python -**

In Python, you can add comments to your code to explain what it does or to provide additional information for others who may read it. Comments are ignored by the Python interpreter and do not affect the execution of the program.

There are two ways to add comments to your Python code:

Single-line comments: Single-line comments begin with the **#** character and continue until the end of the line. Anything after the # character is ignored by the interpreter.

Example:

# This is a single-line comment

print("Hello, World!") # This is another single-line comment

Multi-line comments: Multi-line comments are enclosed in triple quotes (''' or """) and can span multiple lines. They are often used for docstrings (documentation strings) to provide information about a function, module, or class.

Example:

'''

This is a multi-line comment.

It can span multiple lines and is often used for docstrings.

'''

def my\_function():

"""

This is a docstring.

It provides information about the function and its parameters.

"""

Pass

Comments are a helpful way to document your code and make it more understandable to others. It is good practice to include comments in your code to explain what it does, how it works, and any limitations or assumptions it may have.

## **Lists -**

In Python, a list is a collection of values that are ordered and changeable. Lists are one of the most versatile and commonly used data structures in Python, and they can be used to store any combination of data types, including other lists.

Here are some important things to note about lists in Python:

Lists are created using square brackets [ ] and commas , to separate values. For example, **my\_list = [1, 2, 3]** creates a list with three values.

Lists are ordered, which means that the position of each value in the list is important. You can access individual values in a list using their index, which starts at 0 for the first value in the list. For example, my\_list[0] would return the first value in the list.

Lists are mutable, which means that you can add, remove, or modify values in a list after it has been created. For example, my\_list.append(4) would add the value 4 to the end of the list.

Lists can contain any combination of data types, including other lists. This means that you can create nested lists, where one or more values in the list are also lists.

Here are some examples of using lists in Python:

# Creating a list

my\_list = [1, 2, 3, "hello", True]

# Accessing values in a list

print(my\_list[0]) # 1

print(my\_list[3]) # "hello"

# Modifying a list

my\_list[1] = "world"

print(my\_list) # [1, "world", 3, "hello", True]

# Adding values to a list

my\_list.append(4)

print(my\_list) # [1, "world", 3, "hello", True, 4]

# Removing values from a list

my\_list.remove("hello")

print(my\_list) # [1, "world", 3, True, 4]

# Nested lists

nested\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(nested\_list[1][2]) # 6

Output:

1

hello

[1, 'world', 3, 'hello', True]

[1, 'world', 3, 'hello', True, 4]

[1, 'world', 3, True, 4]

6

Lists are an incredibly useful data structure in Python, and they are used extensively in many different types of programs. Understanding how to create, modify, and access values in lists is an important skill for any Python programmer.

**List slicing -**

In Python, list slicing is a way to extract a portion of a list by specifying a range of indices. The syntax for list slicing is as follows:

my\_list[start:stop:step]

start is the index of the first element to include in the slice (inclusive).

stop is the index of the last element to include in the slice (exclusive).

step is the increment between elements in the slice (default is 1).

Here are some examples of using list slicing in Python:

# Creating a list

my\_list = [0, 1, 2, 3, 4, 5]

# Slicing a list

print(my\_list[1:4]) # [1, 2, 3]

print(my\_list[:3]) # [0, 1, 2]

print(my\_list[3:]) # [3, 4, 5]

print(my\_list[::2]) # [0, 2, 4]

print(my\_list[1::2]) # [1, 3, 5]

In the first example, we slice my\_list from index 1 to 4 (exclusive), which returns the values [1, 2, 3]. In the second example, we slice my\_list from the beginning up to index 3 (exclusive), which returns the values [0, 1, 2]. In the third example, we slice my\_list from index 3 to the end, which returns the values [3, 4, 5]. In the fourth example, we slice my\_list with a step of 2, which returns every second value, starting from the beginning, so the output is [0, 2, 4]. In the fifth example, we slice my\_list with a step of 2, but starting from index 1, so the output is [1, 3, 5].

List slicing is a powerful technique that allows you to work with a subset of a list, and it can be used in a wide variety of programming situations. By understanding how to use list slicing in Python, you can make your code more concise and efficient.

Matrix -

In Python, a multi-dimensional list is a list of lists. Each element of the outer list is itself a list, which can contain other lists, and so on. Multi-dimensional lists are useful for representing matrices, tables, or any other data structure that has multiple dimensions.

Here's an example of a two-dimensional list in Python:

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

In this example, the matrix is a 3x3 matrix, with three rows and three columns. The first row is [1, 2, 3], the second row is [4, 5, 6], and the third row is [7, 8, 9].

You can access individual elements of a multi-dimensional list using two indices, one for the row and one for the column. For example, to access the element in the second row and third column of the matrix above, you can use the following code:

print(matrix[1][2]) # Output: 6

You can also use list slicing with multi-dimensional lists to extract submatrices. For example, to extract the first two rows and the first two columns of the matrix, you can use the following code:

submatrix = [row[:2] for row in matrix[:2]]

print(submatrix) # Output: [[1, 2], [4, 5]]

In this code, we used list slicing with two indices to extract the first two rows (matrix[:2]) and then used a list comprehension to extract the first two elements of each row (row[:2]).

Multi-dimensional lists can have any number of dimensions, not just two. For example, a three-dimensional list might represent a cube, and a four-dimensional list might represent a hypercube. The principles of indexing and slicing are the same regardless of the number of dimensions.

**List methods -**

| **Sr. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1 | append() | Adds an element at the end of the list |
| 2 | clear() | Removes all the elements from the list |
| 3 | copy() | Returns a copy of the list |
| 4 | count() | Returns the number of elements with the specified value |
| 5 | extend() | Add the elements of a list (or any iterable), to the end of the current list |
| 6 | index() | Returns the index of the first element with the specified value |
| 7 | insert() | Adds an element at the specified position |
| 8 | pop() | Removes the element at the specified position |
| 9 | remove() | Removes the item with the specified value |
| 10 | reverse() | Reverses the order of the list |
| 11 | sort() | Sorts the list |

**In keyword -**

The in keyword is a membership operator in Python that allows you to check if a value is present in a sequence or collection, such as a string, list, tuple, or dictionary.

The syntax of the in keyword is as follows:

value in sequence

Here, value is the value you want to check for membership, and sequence is the sequence or collection you want to search. The in keyword returns True if the value is present in the sequence, and False otherwise.

Here are some examples:

# Check if a value is in a string

text = "hello world"

if "hello" in text:

print("Found!")

else:

print("Not found.")

# Check if a value is in a list

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

if 3 in numbers:

print("Found!")

else:

print("Not found.")

# Check if a key is in a dictionary

person = {"name": "John", "age": 30}

if "name" in person:

print("Found!")

else:

print("Not found.")

In each of these examples, the **in** keyword is used to check if a value is present in a different type of sequence or collection. If the value is found, the code inside the if statement is executed, otherwise the code inside the else statement is executed.

The **in** keyword can also be used with the **not** keyword to check if a value is not present in a sequence or collection:

# Check if a value is not in a string

text = "hello world"

if "foo" not in text:

print("Not found.")

else:

print("Found.")

# Check if a value is not in a list

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

if 6 not in numbers:

print("Not found.")

else:

print("Found.")

# Check if a key is not in a dictionary

person = {"name": "John", "age": 30}

if "email" not in person:

print("Not found.")

else:

print("Found.")

In each of these examples, the **not in** keywords are used to check if a value is not present in a sequence or collection. If the value is not found, the code inside the if statement is executed, otherwise the code inside the else statement is executed.

**len() vs count() -**

In Python, len() is a built-in function that returns the number of elements in an object. For example, if you have a list my\_list with 5 elements, you can use len(my\_list) to get the number of elements, which would be 5.

On the other hand, count() is a method that is available on certain objects, including strings and lists. This method returns the number of times a specified value appears in the object. For example, if you have a string my\_string that contains the word "hello" three times, you can use my\_string.count("hello") to get the count, which would be 3.

So the main difference between len() and count() is that len() returns the total number of elements in an object, while count() returns the number of times a specific value appears in the object.

**sort() and Sorted() -**

Both sort() and sorted() are used to sort elements in a list, but they work in slightly different ways:

sort() is a method that is available on a list object. When you call my\_list.sort(), it sorts the elements in place, meaning that the original list is modified. sort() has two optional arguments: key, which specifies a function to be called on each element before sorting, and reverse, which specifies whether to sort in descending order (reverse=True) or ascending order (the default).

Here's an example:

my\_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

my\_list.sort()

print(my\_list) # [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

sorted() is a built-in function that takes an iterable as an argument and returns a new sorted list. Unlike sort(), it does not modify the original list. sorted() also has the optional key and reverse arguments.

Here's an example:

my\_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

sorted\_list = sorted(my\_list)

print(sorted\_list) # [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

print(my\_list) # [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

So the main differences between sort() and sorted() are:

sort() modifies the original list in place, while sorted() returns a new sorted list.

sort() is a method on a list object, while sorted() is a built-in function that takes an iterable as an argument.

sort() does not return a value (or returns None), while sorted() returns a new list.

**.copy() method of list -**

The copy() method of a list creates a shallow copy of the original list.

The syntax of the copy() method is as follows:

new\_list = original\_list.copy()

Here, original\_list is the list that you want to copy, and new\_list is the new list that will contain a copy of the elements in original\_list.

A shallow copy means that a new list object is created, but the elements of the new list are still references to the same objects as the elements in the original list. If any of the elements in the original list are mutable objects, such as lists or dictionaries, changes made to those objects in the new list will affect the objects in the original list.

Here is an example that demonstrates the use of the copy() method:

original\_list = [1, 2, [3, 4]]

new\_list = original\_list.copy()

# Modify an element in the new list

new\_list[0] = 5

# Modify an element in a nested list

new\_list[2][0] = 6

print(original\_list) # Output: [1, 2, [6, 4]]

print(new\_list) # Output: [5, 2, [6, 4]]

In this example, original\_list is a list that contains three elements: the integers 1 and 2, and a nested list [3, 4]. We then create a new list new\_list that is a copy of original\_list. We modify the first element of new\_list to be the integer 5, and we also modify the first element of the nested list in new\_list to be the integer 6.

When we print both lists, we can see that the modification to the first element of the nested list in new\_list also affected the nested list in original\_list. This is because both lists contain references to the same nested list object.

Therefore, when using the copy() method on a list in Python, it is important to keep in mind the concept of shallow copying, and to use it appropriately depending on the needs of your program. If you want to make a deep copy of a list, where all the elements are also copied recursively, you can use the copy.deepcopy() method from the copy module.

**range() -**

In Python, range() is a built-in function that returns an immutable sequence of numbers. It is commonly used to create a sequence of integers to use in a for loop or to generate a list of numbers.

The basic syntax of the range() function is **range(start, stop, step)**, where start is the first number in the sequence (default is 0), stop is the last number in the sequence (not included in the sequence), and step is the difference between each number (default is 1).

Here are a few examples of using range():

# Print the numbers from 0 to 4

for i in range(5):

print(i)

# Print the even numbers from 0 to 8

for i in range(0, 9, 2):

print(i)

# Generate a list of squares of numbers from 1 to 5

squares = [i\*\*2 for i in range(1, 6)]

print(squares) # [1, 4, 9, 16, 25]

Note that range() returns an object of type range, which is an immutable sequence type. If you need a list of numbers instead, you can convert the range object to a list using the list() function:

# Generate a list of numbers from 1 to 10

numbers = list(range(1, 11))

print(numbers) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In Python 2, the range() function returns a list, whereas in Python 3, it returns a range object. However, Python 3 also has a built-in function called list() that can be used to convert a range object to a list.

**List unpacking -**

In Python, list unpacking is a way to assign each element of a list to a separate variable, in a single line of code. It is a convenient way to access and manipulate the individual elements of a list, without having to reference them by their index.

The syntax of list unpacking is as follows:

a, b, c = my\_list

Here, my\_list is the list that you want to unpack, and a, b, and c are the variables that you want to assign the values of the corresponding elements in the list to.

Here is an example that demonstrates the use of list unpacking:

my\_list = [1, 2, 3]

a, b, c = my\_list

print(a) # Output: 1

print(b) # Output: 2

print(c) # Output: 3

In this example, we have a list my\_list that contains three elements: the integers 1, 2, and 3. We then use list unpacking to assign the values of these elements to the variables a, b, and c, respectively. Finally, we print out the values of these variables to verify that they contain the expected values.

List unpacking can also be used with variable-length lists, as long as the number of variables on the left-hand side of the assignment operator matches the number of elements in the list:

my\_list = [1, 2, 3, 4, 5]

a, b, \*c = my\_list

print(a) # Output: 1

print(b) # Output: 2

print(c) # Output: [3, 4, 5]

In this example, we have a list my\_list that contains five elements. We use list unpacking to assign the first two elements to the variables a and b, and the remaining elements to the list variable c, using the special \* syntax. The \* syntax indicates that any remaining elements in the list should be packed into a list.

List unpacking can also be used to swap the values of two variables without using a temporary variable:

a = 1

b = 2

a, b = b, a

print(a) # Output: 2

print(b) # Output: 1

In this example, we have two variables a and b that contain the integers 1 and 2, respectively. We use list unpacking to swap their values, by assigning the value of b to a, and the value of a to b, in a single line of code.

Overall, list unpacking is a powerful and convenient feature of Python that can simplify the manipulation of lists and the assignment of values to variables.

**None -**

In Python, None is a special value that represents the absence of a value. It is often used to indicate that a variable or function argument does not have a meaningful value, or that a function call has no return value. None is also used as the default return value for functions that do not explicitly return anything.

Here are some examples of how None is used in Python:

Assigning None to a variable:

my\_var = None

This assigns the value None to the variable my\_var, indicating that it does not have a meaningful value.

Using None as a default function argument:

def my\_function(arg1, arg2=None):

if arg2 is None:

arg2 = 0

return arg1 + arg2

In this example, the function my\_function takes two arguments, arg1 and arg2. If arg2 is not provided by the caller, its default value is set to None. The function then checks if arg2 is None, and if so, sets its value to 0 before returning the sum of arg1 and arg2.

Using None as a return value:

def my\_function():

# do some work...

return None

In this example, the function my\_function does some work, but does not have a meaningful return value. It therefore returns None by default.

Comparing a variable to None:

my\_var = None

if my\_var is None:

print("my\_var has no value")

In this example, the variable my\_var is compared to None using the is operator. This is a common way to check if a variable has been assigned a meaningful value.

Overall, None is a useful and common feature of Python that is used to represent the absence of a value. It is important to keep in mind that None is not the same as an empty string (''), an empty list ([ ]), or a zero value (0). These are all valid values that have meaning in certain contexts, whereas None specifically indicates the absence of a value.

**Dictionaries -**

In Python, a dictionary is a collection of key-value pairs, where each key is associated with a value. Dictionaries are an extremely useful data structure for storing and organising data, particularly when the data can be accessed and updated using keys.

Here is an example of a dictionary in Python:

my\_dict = {

'name': 'John',

'age': 30,

'gender': 'male'

}

In this example, we have created a dictionary called my\_dict that contains three key-value pairs. The keys are 'name', 'age', and 'gender', and the corresponding values are 'John', 30, and 'male'.

We can access the values in a dictionary using the keys. For example, to access the value associated with the 'name' key in my\_dict, we can use the following code:

print(my\_dict['name'])

This will output:

John

We can also add new key-value pairs to a dictionary using the following syntax:

my\_dict['email'] = 'john@example.com'

This will add a new key-value pair to my\_dict with the key 'email' and the value 'john@example.com'.

To update the value associated with a key, we can simply assign a new value to that key:

my\_dict['age'] = 31

This will update the value associated with the 'age' key to 31.

We can also remove key-value pairs from a dictionary using the del keyword:

del my\_dict['gender']

This will remove the key-value pair with the key 'gender' from my\_dict.

Dictionaries can also be used in more complex ways, such as storing nested data structures or using them to represent real-world objects. Here is an example of a nested dictionary:

my\_dict = {

'person1': {

'name': 'John',

'age': 30

},

'person2': {

'name': 'Jane',

'age': 25

}

}

In this example, we have a dictionary that contains two key-value pairs, each of which is itself a dictionary. This allows us to store multiple pieces of data for each person.

Overall, dictionaries are a powerful and flexible data structure in Python that can be used in a variety of ways. They are particularly useful for storing and organising data that can be accessed and updated using keys.

Properties of dictionary keys -

In Python dictionaries, keys have several properties that distinguish them from other data types:

Unique: Keys in a dictionary must be unique. If you try to assign a value to a key that already exists in the dictionary, the new value will overwrite the existing value.

Immutable: Dictionary keys must be immutable. This means that they cannot be changed after they are created. Common examples of immutable data types in Python include strings, numbers, and tuples. Mutable data types like lists cannot be used as dictionary keys.

Hashable: Since dictionaries are implemented using a hash table, keys must be hashable. This means that the hash value of a key must remain constant throughout the lifetime of the dictionary. Immutable data types like strings and numbers are hashable, while mutable data types like lists and dictionaries are not.

In addition to these properties, it's important to note that dictionary keys are case-sensitive. This means that 'Name' and 'name' would be considered two different keys in a dictionary.

Here's an example that demonstrates these properties:

# Valid dictionary with string keys

my\_dict = {

'name': 'John',

'age': 30

}

# Invalid dictionary with mutable keys

invalid\_dict = {

['name']: 'John',

['age']: 30

}

# Raises TypeError: unhashable type: 'list'

# Invalid dictionary with non-unique keys

invalid\_dict = {

'name': 'John',

'name': 'Jane'

}

# Second value overwrites first value

# Valid dictionary with integer keys

valid\_dict = {

1: 'John',

2: 'Jane'

}

Uncommon way to create dictionary

user = dict(name=’John doe’)

**Dictionary methods -**

| **Sr. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1 | clear() | Removes all the elements from the dictionary |
| 2 | copy() | Returns a copy of the dictionary |
| 3 | fromkeys() | Returns a dictionary with the specified keys and value |
| 4 | get() | Returns the value of the specified key |
| 5 | items() | Returns a list containing a tuple for each key value pair |
| 6 | keys() | Returns a list containing the dictionary's keys |
| 7 | pop() | Removes the element with the specified key |
| 8 | popitem() | Removes the last inserted key-value pair |
| 9 | setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| 10 | update() | Updates the dictionary with the specified key-value pairs |
| 11 | values() | Returns a list of all the values in the dictionary |

**Tuples-**

In Python, a tuple is a collection of ordered and immutable elements, enclosed in parentheses. Tuples are similar to lists, but they cannot be modified once created. This means that you can't add, remove, or change elements in a tuple.

Here's an example of creating a tuple:

my\_tuple = (1, 2, 3)

You can also create a tuple using the built-in tuple() function:

my\_tuple = tuple([1, 2, 3])

One important thing to note about creating a tuple with a single element is that you must include a comma after the element, to distinguish it from just a regular value in parentheses.

For example:

single\_tuple = (1,)

Now that we've created a tuple, let's look at some of its features:

Accessing elements: You can access elements in a tuple using indexing, just like with a list. For example:

my\_tuple = (1, 2, 3)

print(my\_tuple[0]) # Output: 1

Slicing: You can also slice a tuple to extract a subset of its elements, just like with a list. For example:

my\_tuple = (1, 2, 3, 4, 5)

print(my\_tuple[1:4]) # Output: (2, 3, 4)

Concatenation: You can concatenate tuples using the + operator, which creates a new tuple containing the elements of both tuples. For example:

tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

combined\_tuple = tuple1 + tuple2

print(combined\_tuple) # Output: (1, 2, 3, 4, 5, 6)

Tuple unpacking: You can use tuple unpacking to assign the elements of a tuple to separate variables. For example:

my\_tuple = (1, 2, 3)

a, b, c = my\_tuple

print(a) # Output: 1

print(b) # Output: 2

print(c) # Output: 3

Length: You can find the length of a tuple using the built-in len() function. For example:

my\_tuple = (1, 2, 3)

print(len(my\_tuple)) # Output: 3

Immutable: Unlike lists, tuples are immutable, which means you cannot add, remove or change elements once a tuple has been created. For example:

my\_tuple = (1, 2, 3)

my\_tuple[0] = 4 # Raises TypeError: 'tuple' object does not support item

assignment

Tuples are commonly used for returning multiple values from a function, as they can be unpacked to separate variables. They are also used to create a collection of elements that should not be modified, or to represent an ordered collection of items where the order should not be changed.

**Tuple methods -**

| **Sr. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1 | count() | Returns the number of times a specified value occurs in a tuple |
| 2 | index() | Searches the tuple for a specified value and returns the position of where it was found |

**Sets -**

In Python, a set is an unordered collection of unique elements enclosed in curly braces {} or created using the built-in set() function. Sets are similar to lists and tuples, but they cannot contain duplicate values.

Here's an example of creating a set:

my\_set = {1, 2, 3}

You can also create a set using the set() function:

my\_set = set([1, 2, 3])

One of the key features of a set is that it only contains unique elements. If you try to add a duplicate value to a set, it will be ignored. For example:

my\_set = {1, 2, 3}

my\_set.add(3) # Ignored, since 3 is already in the set

print(my\_set) # Output: {1, 2, 3}

Here are some other important features of sets:

Membership testing: You can test whether an element is in a set using the in operator. For example:

my\_set = {1, 2, 3}

print(2 in my\_set) # Output: True

print(4 in my\_set) # Output: False

Set operations: You can perform operations on sets, such as union, intersection, and difference, using various methods or operators. For example:

set1 = {1, 2, 3}

set2 = {2, 3, 4}

union\_set = set1.union(set2)

print(union\_set) # Output: {1, 2, 3, 4}

intersection\_set = set1.intersection(set2)

print(intersection\_set) # Output: {2, 3}

difference\_set = set1.difference(set2)

print(difference\_set) # Output: {1}

Set methods -

| **Sr. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1 | add() | Adds an element to the set |
| 2 | clear() | Removes all the elements from the set |
| 3 | copy() | Returns a copy of the set |
| 4 | difference() | Returns a set containing the difference between two or more sets |
| 5 | difference\_update() | Removes the items in this set that are also included in another, specified set |
| 6 | discard() | Remove the specified item |
| 7 | intersection() | Returns a set, that is the intersection of two or more sets |
| 8 | intersection\_update() | Removes the items in this set that are not present in other, specified set(s) |
| 9 | isdisjoint() | Returns whether two sets have a intersection or not |
| 10 | issubset() | Returns whether another set contains this set or not |
| 11 | issuperset() | Returns whether this set contains another set or not |
| 12 | pop() | Removes an element from the set |
| 13 | remove() | Removes the specified element |
| 14 | symmetric\_difference() | Returns a set with the symmetric differences of two sets |
| 15 | symmetric\_difference\_update() | inserts the symmetric differences from this set and another |
| 16 | union() | Return a set containing the union of sets |
| 17 | update() | Update the set with another set, or any other iterable |

**Python Keywords -**

| **Sr. No.** | **Keyword** | **Description** |
| --- | --- | --- |
| 1 | and | A logical operator |
| 2 | as | To create an alias |
| 3 | assert | For debugging |
| 4 | break | To break out of a loop |
| 5 | class | To define a class |
| 6 | continue | To continue to the next iteration of a loop |
| 7 | def | To define a function |
| 8 | del | To delete an object |
| 9 | elif | Used in conditional statements, same as else if |
| 10 | else | Used in conditional statements |
| 11 | except | Used with exceptions, what to do when an exception occurs |
| 12 | False | Boolean value, result of comparison operations |
| 13 | finally | Used with exceptions, a block of code that will be executed no matter if there is an exception or not |
| 14 | for | To create a for loop |
| 15 | from | To import specific parts of a module |
| 16 | global | To declare a global variable |
| 17 | if | To make a conditional statement |
| 18 | import | To import a module |
| 19 | in | To check if a value is present in a list, tuple, etc. |
| 20 | is | To test if two variables are equal |
| 21 | lambda | To create an anonymous function |
| 22 | None | Represents a null value |
| 23 | nonlocal | To declare a non-local variable |
| 24 | not | A logical operator |
| 25 | or | A logical operator |
| 26 | pass | A null statement, a statement that will do nothing |
| 27 | raise | To raise an exception |
| 28 | return | To exit a function and return a value |
| 29 | True | Boolean value, result of comparison operations |
| 30 | try | To make a try...except statement |
| 31 | while | To create a while loop |
| 32 | with | Used to simplify exception handling |
| 33 | yield | To end a function, returns a generator |

**Conditional Logic -**

Conditional logic in Python allows you to make decisions based on conditions. In Python, the if statement is used for conditional logic. Here's an example:

x = 10

if x > 5:

print("x is greater than 5")

In this example, we first define a variable x with a value of 10. We then use the if statement to test whether x is greater than 5. Since x is indeed greater than 5, the code inside the if block will be executed, and the output will be:

x is greater than 5

Here are some other examples of conditional logic in Python:

If-else statement:

x = 10

if x > 5:

print("x is greater than 5")

else:

print("x is less than or equal to 5")

In this example, we use the if-else statement to print a different message depending on the value of x.

Elif statement:

x = 10

if x > 10:

print("x is greater than 10")

elif x == 10:

print("x is equal to 10")

else:

print("x is less than 10")

Nested if statement:

x = 10

y = 5

if x > 5:

if y > 3:

print("Both x and y are greater than their respective values")

else:

print("x is greater than 5, but y is not greater than 3")

else:

print("x is not greater than 5")

In this example, we use nested if statements to test two conditions. The first if statement checks whether x is greater than 5. If it is, we then check whether y is greater than 3.

Conditional logic is an important aspect of programming, as it allows you to write code that can make decisions based on different conditions.

**Indentation in python -**

In Python, indentation is used to define the scope of a code block. Unlike many other programming languages that use curly braces or other characters to denote code blocks, Python uses indentation as a way of indicating which lines of code belong to which block.

For example, consider the following code that uses an if statement:

x = 10

if x > 5:

print("x is greater than 5")

In this code, the if statement is followed by a colon (:), and the next line is indented to show that it is part of the code block that is executed if the condition is true. If we didn't indent the second line, Python would raise an IndentationError because it wouldn't know which lines are part of the code block and which are not.

Here's another example that uses a for loop:

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

for num in numbers:

if num % 2 == 0:

print(num, "is even")

else:

print(num, "is odd")

In this code, the for loop is followed by a colon (:), and the next line is indented to show that it is part of the code block that is executed for each iteration of the loop. Similarly, the if statement is followed by a colon (:), and the next two lines are indented to show that they are part of the code block that is executed if the condition is true. Again, if we didn't indent the lines correctly, Python would raise an IndentationError.

Indentation in Python is typically done using four spaces. While it is possible to use tabs or a combination of tabs and spaces, it's generally recommended to use spaces to avoid any issues with different editors interpreting tabs differently.

**Truthy vs Falsey -**

In Python, a value is said to be "truthy" if it evaluates to True in a boolean context, and "falsey" if it evaluates to False. When a boolean value is expected in a Python expression, any value can be used, and Python will automatically convert it to a boolean using the following rules:

Falsey values: False, 0, 0.0, "" (empty string), None, [] (empty list), {} (empty dictionary), and () (empty tuple)

Truthy values: anything else that is not falsey

Here are some examples to illustrate truthy vs falsey in Python:

if 0:

print("This will not be printed, because 0 is falsey.")

if "":

print("This will not be printed, because an empty string is falsey.")

if None:

print("This will not be printed, because None is falsey.")

if []:

print("This will not be printed, because an empty list is falsey.")

if {}:

print("This will not be printed, because an empty dictionary is falsey.")

if ():

print("This will not be printed, because an empty tuple is falsey.")

if "hello":

print("This will be printed, because a non-empty string is truthy.")

if [1, 2, 3]:

print("This will be printed, because a non-empty list is truthy.")

if {"a": 1, "b": 2}:

print("This will be printed, because a non-empty dictionary is truthy.")

if (1, 2, 3):

print("This will be printed, because a non-empty tuple is truthy.")

Note that the bool() function can also be used to convert a value to a boolean explicitly. For example:

print(bool(0)) # False

print(bool("")) # False

print(bool(None)) # False

print(bool([])) # False

print(bool({})) # False

print(bool(())) # False

print(bool("hello")) # True

print(bool([1, 2])) # True

print(bool({"a": 1})) # True

print(bool((1,))) # True

To make use of truthy and falsey values in Python, you can use them in conditional statements like if, elif, and while loops. For example:

x = 5

if x > 10:

print("x is greater than 10")

elif x > 5:

print("x is greater than 5 but less than or equal to 10")

else:

print("x is less than or equal to 5")

In this example, the first condition x > 10 is false, so Python moves on to the next condition x > 5, which is also false, so it finally executes the else block.

Another example of using truthy and falsey values is checking if a variable is defined:

# Define a variable

my\_var = "hello"

# Check if the variable is defined

if my\_var:

print("my\_var is defined")

else:

print("my\_var is not defined")

In this example, the if statement checks if my\_var is truthy, which means it's defined and has a value. If my\_var was not defined (e.g. my\_var = None), then the else block would be executed instead.

It's important to keep truthy and falsey values in mind when working with conditional logic in Python, as it can help simplify your code and make it more readable.

**Ternary operator -** (also called as ternary expressions)

In Python, the ternary operator is a shorthand way of writing a simple conditional expression. It allows you to write an expression in a more concise way, by combining the if and else clauses into a single line. The syntax of the ternary operator in Python is:

value\_if\_true if condition else value\_if\_false

Here, condition is the expression that is evaluated to a Boolean value, and value\_if\_true and value\_if\_false are the values that are returned if the condition is True or False, respectively.

Here is an example of how to use the ternary operator in Python:

x = 5

y = 10

max\_value = x if x > y else y

print(max\_value)

In this example, we have two variables x and y with values of 5 and 10, respectively. We want to find the maximum value between these two variables. The ternary operator is used to check if x is greater than y. If the condition is true, x is assigned to max\_value, otherwise y is assigned to max\_value.

When we run this code, the output will be:

10

Since y is greater than x, the value of y is assigned to max\_value.

**Short circuiting -**

Short-circuiting is a behavior in Python (and other programming languages) where the evaluation of a boolean expression stops as soon as the truth value of the expression is determined. This means that if the first part of a boolean expression is enough to determine the truth value of the whole expression, the rest of the expression is not evaluated.

Python has two logical operators that implement short-circuiting: and and or.

For the and operator, if the left operand evaluates to False, the entire expression is False, so the right operand is not evaluated. On the other hand, if the left operand evaluates to True, the right operand is evaluated to determine the final truth value of the expression.

Example:

x = 5

y = 10

if x > 0 and y/x > 2:

print("y is at least twice x")

else:

print("y is not at least twice x")

In this example, the left operand of the and operator is x > 0, which evaluates to True. Therefore, the right operand y/x > 2 is evaluated to determine the final truth value of the expression. If x were negative, the entire expression would be False and the right operand would not be evaluated.

For the or operator, if the left operand evaluates to True, the entire expression is True, so the right operand is not evaluated. On the other hand, if the left operand evaluates to False, the right operand is evaluated to determine the final truth value of the expression.

Example:

x = None

y = 10

z = x or y

print(z)

In this example, the left operand of the or operator is x, which is None. Since None is considered False in a boolean context, the right operand y is evaluated and its value is assigned to z. If x were a non-empty string or any other value that is considered True in a boolean context, the entire expression would be True and the right operand would not be evaluated.

**Why print('a'>'A') evaluates to True in python?**

In Python, the comparison of two strings using the relational operators like > is performed based on their lexicographic order. In lexicographic order, the first character of each string is compared. If they are equal, the second characters are compared, and so on until a difference is found.

In this case, 'a' and 'A' are two strings being compared using the > operator. Since 'a' comes after 'A' in the ASCII table, the expression 'a' > 'A' evaluates to True.

In ASCII encoding, the lowercase letters are assigned a higher code point than their uppercase counterparts. For example, the ASCII code point for 'A' is 65, and the code point for 'a' is 97. This means that when we compare two strings 'a' and 'A', the comparison is based on their ASCII code points, which are 97 and 65 respectively. Since 97 is greater than 65, the expression 'a' > 'A' evaluates to True.

Here is an example to demonstrate this:

print(ord('a'))

print(ord('A'))

In this example, the ord() function is used to get the ASCII code point of each character. The output of this code will be:

97

65

This shows that the ASCII code point of 'a' is greater than the ASCII code point of 'A'. Therefore, the expression 'a' > 'A' evaluates to True.

**What will be the output of print(1<2>3<4<5) in python and why?**

The output of the expression print(1<2>3<4<5) in Python will be False.

This is because the comparison operators in Python, such as < and >, have left-to-right associativity. This means that the expression is evaluated from left to right, and the comparison operations are evaluated in pairs.

Let's break down the expression 1<2>3<4<5 into its individual comparison operations:

* 1 < 2: This evaluates to **True**.
* 2 > 3: This evaluates to **False**, since 2 is not greater than 3.
* 3 < 4: This evaluates to **True**.
* 4 < 5: This evaluates to **True**.

So the final expression that is evaluated is True > False < True < True, which is equivalent to 1 > 0 < 1 < 1.

The comparison 1 > 0 is evaluated to True, but then the comparison True < 1 is evaluated to False, which means the overall result of the expression is False.

**Is vs ==**

In Python, is and == are both comparison operators, but they are used for different purposes.

The == operator is used to compare the values of two objects. It checks whether the two objects have the same value or not. If the values are the same, the == operator returns True. Otherwise, it returns False.

Here is an example:

a = [1, 2, 3]

b = [1, 2, 3]

print(a == b) # Output: True

In this example, we have two lists a and b that have the same values. When we compare them using the == operator, it returns True.

On the other hand, the is operator is used to compare the identity of two objects. It checks whether the two objects refer to the same memory location or not. If they refer to the same memory location, the is operator returns True. Otherwise, it returns False.

Here is an example:

a = [1, 2, 3]

b = [1, 2, 3]

print(a is b) # Output: False

In this example, we have two lists a and b that have the same values. However, they are stored in different memory locations, so when we compare them using the is operator, it returns False.

Note that is is generally used for checking whether two variables refer to the same object or not, while == is used for checking whether two variables have the same value or not. However, in some cases, the behavior of is and == may be the same. For example, when comparing two strings or two numbers that are interned in memory, is and == will give the same result.

**Loops -**

for loop -

In Python, a for loop is used to iterate over a sequence of elements, such as a list, tuple, or string. The basic syntax of a for loop is as follows:

for variable in sequence:

# statements to be executed for each element

In this syntax, variable is a new variable that takes on the value of each element in the sequence on each iteration of the loop. The statements to be executed for each element are indented underneath the for loop header.

Here's an example that shows how to use a for loop to iterate over a list of numbers:

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

for num in numbers:

print(num)

In this example, we have a list of numbers. The for loop iterates over each element in the list and assigns it to the variable num. The print statement inside the loop prints out each number on a new line.

The output of this code will be:

1

2

3

4

5

We can also use the built-in range() function to generate a sequence of numbers and iterate over them using a for loop. Here's an example:

for i in range(1, 6):

print(i)

In this example, the range() function generates a sequence of numbers from 1 to 5 (inclusive). The for loop iterates over each number in the sequence and prints it out.

The output of this code will be:

1

2

3

4

5

In addition to iterating over sequences of elements, for loops can also be used to iterate over the keys or values in a dictionary using the items() method. Here's an example:

person = {"name": "Alice", "age": 25, "city": "New York"}

for key, value in person.items():

print(key, ":", value)

In this example, we have a dictionary person that contains information about a person. The for loop iterates over the items() in the dictionary, which returns a sequence of key-value pairs. On each iteration of the loop, the key variable takes on the key of the current item, and the value variable takes on the value of the current item. The print statement inside the loop prints out the key and value of each item.

The output of this code will be:

name : Alice

age : 25

city : New York

Overall, for loops are a powerful tool for iterating over sequences of elements, and they are a fundamental construct in Python programming.

**Iterables -**

In Python, an iterable is any object that can be looped over using a for loop. An iterable object is an object that has an \_\_iter\_\_() method that returns an iterator object. An iterator object is an object that has a \_\_next\_\_() method that returns the next item in the sequence, and raises a StopIteration exception when there are no more items to return.

Some examples of iterable objects in Python include:

Lists: A list is an ordered collection of elements, and it can be iterated over using a for loop. For example:

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

for num in numbers:

print(num)

Tuples: A tuple is similar to a list, but it is immutable. Tuples can also be iterated over using a for loop. For example:

person = ("Alice", 25, "New York")

for item in person:

print(item)

Strings: A string is a sequence of characters, and it can be iterated over using a for loop. For example:

message = "Hello, World!"

for char in message:

print(char)

Sets: A set is an unordered collection of unique elements, and it can be iterated over using a for loop. For example:

names = {"Alice", "Bob", "Charlie"}

for name in names:

print(name)

Dictionaries: A dictionary is a collection of key-value pairs, and it can be iterated over using a for loop. For example:

person = {"name": "Alice", "age": 25, "city": "New York"}

for key, value in person.items():

print(key, ":", value)

In addition to these built-in iterable objects, you can also create your own iterable objects by defining a class with an \_\_iter\_\_() method that returns an iterator object. This is a powerful feature of Python that allows you to create custom iterable objects tailored to your specific needs.

**range()-**

The range() function in Python is used to generate a sequence of numbers that can be used in a for loop or other iteration constructs. The basic syntax of the range() function is:

range(stop)

range(start, stop[, step])

The range() function takes one to three arguments:

start (optional): The starting value of the sequence (inclusive). If not specified, the default value is 0.

stop (required): The stopping value of the sequence (exclusive). This argument must be specified.

step (optional): The step size between each number in the sequence. If not specified, the default value is 1.

Here are a few examples of how the range() function can be used:

# Print the numbers from 0 to 4

for i in range(5):

print(i)

# Print the even numbers from 0 to 8

for i in range(0, 9, 2):

print(i)

# Print the odd numbers from 1 to 9

for i in range(1, 10, 2):

print(i)

In the first example, the range() function generates a sequence of numbers from 0 to 4 (inclusive) because we specified 5 as the stop argument.

In the second example, the range() function generates a sequence of even numbers from 0 to 8 (inclusive) because we specified 0 as the start argument, 9 as the stop argument, and 2 as the step argument.

In the third example, the range() function generates a sequence of odd numbers from 1 to 9 (inclusive) because we specified 1 as the start argument, 10 as the stop argument, and 2 as the step argument.

It's worth noting that the range() function returns a sequence object that is not actually a list. This means that you cannot access the individual elements of a range() object directly like you can with a list. Instead, you must iterate over the range() object using a for loop or convert it to a list using the list() function.

for \_ in range(0,10):

print(‘i’)

**Enumerate -**

The enumerate() function in Python is a built-in function that adds a counter to an iterable object (like a list or a string) and returns an enumerate object. The enumerate object contains pairs of the form (index, value) for each item in the iterable.

The basic syntax of the enumerate() function is:

enumerate(iterable, start=0)

The enumerate() function takes two arguments:

iterable (required): The iterable object that you want to enumerate.

start (optional): The starting value of the index. If not specified, the default value is 0.

Here is an example of how to use the enumerate() function:

fruits = ["apple", "banana", "orange"]

for index, fruit in enumerate(fruits):

print(index, fruit)

In this example, the enumerate() function is used to add a counter to the fruits list. The resulting enumerate object contains pairs of the form (index, value) for each fruit in the list. The for loop then iterates over the enumerate object and prints out the index and value of each fruit.

The output of this code will be:

0 apple

1 banana

2 orange

The enumerate() function can be useful when you want to keep track of the index of each item in an iterable object. It is a handy shortcut that can save you from writing a separate counter variable and incrementing it manually.

**While Loops -**

A while loop in Python is a control flow statement that allows you to execute a block of code repeatedly as long as a specified condition is true. The basic syntax of a while loop in Python is as follows:

while condition:

# Code to execute while the condition is true

The condition is an expression that is evaluated before each iteration of the loop. If the condition is true, the code inside the loop is executed. This process is repeated until the condition becomes false.

Here's an example that uses a while loop to print the numbers from 1 to 5:

i = 1

while i <= 5:

print(i)

i += 1

else:

print(‘Done with all the work’)

In this example, the loop starts with i equal to 1. The condition i <= 5 is true, so the code inside the loop is executed. The print(i) statement outputs the value of i, which is 1. The statement i += 1 increments the value of i by 1, so i is now equal to 2.

The loop then repeats, and the condition is checked again. Since i is still less than or equal to 5, the loop continues to execute. This process is repeated until i becomes 6, at which point the condition i <= 5 is false, and the loop stops executing.

Here's another example that uses a while loop to prompt the user for input until they enter a valid integer:

while True:

try:

num = int(input("Enter an integer: "))

Break

except ValueError:

print("Invalid input! Please enter an integer.")

In this example, the while loop continues to execute until the break statement is encountered. The loop prompts the user for input using the input() function and attempts to convert the input to an integer using the int() function. If the input is not a valid integer, a ValueError exception is raised, and the except block is executed. The except block outputs an error message and then continues the loop, prompting the user for input again. If the input is valid, the break statement is executed, and the loop stops executing.

while loops are useful when you want to execute a block of code repeatedly while a certain condition is true. However, it's important to be careful when using while loops, as an infinite loop can occur if the condition never becomes false.

**While - else -**

In Python, the else statement in a while loop is an optional clause that can be used to execute a block of code after the loop finishes executing normally. This means that if the loop condition becomes false and the loop terminates, the code inside the else block will be executed. However, if the loop is exited prematurely, such as with a break statement, the else block will not be executed.

Here is an example of how the else statement can be used in a while loop:

count = 0

while count < 5:

print(count)

count += 1

else:

print("Loop completed")

In this example, the while loop will iterate five times, printing the value of count each time. After the loop finishes normally, the message "Loop completed" will be printed.

Another example of using else with a while loop is to check whether a condition has been met within the loop. For example, the following code uses a while loop to search for the first occurrence of the letter "e" in a string, and prints a message if the letter is found:

word = "Python"

index = 0

while index < len(word):

if word[index] == "e":

print("Found 'e' at index", index)

break

index += 1

else:

print("Letter 'e' not found")

In this example, the while loop will iterate through each character in the string word, checking if the current character is equal to the letter "e". If the letter is found, a message is printed and the loop is exited with a break statement. If the loop finishes normally without finding the letter, the message "Letter 'e' not found" is printed.

**break, continue and pass -**

In Python, break, continue, and pass are control flow statements that can be used to alter the normal execution of a loop or a conditional statement.

break: The break statement is used to immediately exit a loop. When break is encountered inside a loop, the loop is terminated and control is passed to the next statement following the loop. Here's an example:

for i in range(1, 11):

if i == 5:

break

print(i)

In this example, the for loop will iterate from 1 to 10. When i becomes 5, the break statement is executed, and the loop is terminated. Therefore, the output will be:

1

2

3

4

continue: The continue statement is used to skip the current iteration of a loop and move on to the next iteration. When continue is encountered inside a loop, the current iteration is skipped and the loop immediately jumps to the next iteration. Here's an example:

for i in range(1, 11):

if i % 2 == 0:

continue

print(i)

In this example, the for loop will iterate from 1 to 10. When i is an even number, the continue statement is executed, and the current iteration is skipped. Therefore, the output will be:

Copy code

1

3

5

7

9

pass: The pass statement is a null operation that does nothing. It is used as a placeholder when a statement is required syntactically, but no action is required. For example, if you want to define a function that does nothing, you can use the pass statement as a placeholder:

def my\_function():

pass

In this example, my\_function does nothing. The pass statement is used as a placeholder to indicate that the function does not have any code to execute.

1. Clean Coding -   
   Use meaningful variable and function names that describe their purpose.
2. Keep functions and classes small and focused on a single task.
3. Follow the DRY (Don't Repeat Yourself) principle by avoiding code duplication.
4. Write comments that explain why you're doing something, not what you're doing.
5. Use docstrings to document classes and functions.
6. Use descriptive names for variables, functions, and classes that reflect their purpose.
7. Use whitespace to improve readability and separate logical blocks of code.
8. Use consistent naming conventions for variables, functions, and classes.
9. Use Python's built-in data types and functions whenever possible.
10. Use list comprehensions or generator expressions instead of for loops when appropriate.
11. Use default argument values instead of None to avoid confusion.
12. Use error handling to catch and handle exceptions.
13. Avoid using global variables when possible.
14. Use object-oriented programming principles to encapsulate data and behavior.
15. Use modules and packages to organize your code into logical units.
16. Write unit tests to verify the correctness of your code.
17. Use pylint or flake8 to check your code for errors and adherence to coding standards.
18. Use logging to record events and errors in your code.
19. Use the contextlib module to manage resources and ensure their proper cleanup.
20. Use Python's standard library whenever possible to avoid reinventing the wheel.

DRY principle -

The DRY (Don't Repeat Yourself) principle in Python coding emphasizes the avoidance of code duplication by promoting code reuse, modularity, and abstraction, resulting in more concise, efficient, and maintainable code.

Functions -

A Python function is a block of reusable code that performs a specific task. Functions allow you to break your program down into smaller, more manageable pieces, making your code easier to read, write, and maintain. In Python, functions are defined using the "def" keyword, followed by the function name, arguments in parentheses, and a colon. The function body is indented below the header, and it contains the code that performs the task.

Here is an example of a Python function that calculates the area of a circle, given its radius:

def calculate\_area(radius):

pi = 3.14159

area = pi \* radius\*\*2

return area

In this function, "calculate\_area" is the function name, and "radius" is the argument that the function takes. The function calculates the area of the circle using the formula pi times the radius squared, and then returns the result using the "return" keyword.

You can call the function by passing a value for the radius, like this:

>>> calculate\_area(5)

78.53975

This will return the area of a circle with a radius of 5.

Functions can also have multiple arguments, like this:

def add\_numbers(x, y):

sum = x + y

return sum

This function takes two arguments, "x" and "y", adds them together, and then returns the result.

You can call this function by passing two values for "x" and "y":

>>> add\_numbers(2, 3)

5

This will return the sum of 2 and 3, which is 5.

Functions can also have optional arguments, like this:

def greet(name, greeting='Hello'):

print(greeting, name)

In this function, "name" is a required argument, while "greeting" is an optional argument that defaults to "Hello" if no value is provided.

You can call this function with or without a value for "greeting":

>>> greet('Bob')

Hello Bob

>>> greet('Alice', 'Hi')

Hi Alice

In the first example, "greeting" defaults to "Hello", while in the second example, it is set to "Hi".

Functions can also return multiple values, like this:

def rectangle\_dimensions(length, width):

area = length \* width

perimeter = 2 \* (length + width)

return area, perimeter

In this function, "rectangle\_dimensions", the area and perimeter of a rectangle are calculated using the length and width passed as arguments, and then returned as a tuple.

You can call this function and assign its output to two variables like this:

>>> area, perimeter = rectangle\_dimensions(3, 4)

>>> print('Area:', area)

Area: 12

>>> print('Perimeter:', perimeter)

Perimeter: 14

This will print the area and perimeter of a rectangle with a length of 3 and a width of 4.

Conventions -

Should do one thing really well

And function should return something

Arguments vs Parameters -

Parameter - define a function

Argument - call , invoke a function

Positional arguments

Keyword arguments

Default parameters

Return keyword - if not gives none, automatically exits the function

Nested functions

Explain these two approaches-

def function\_one(num1, num2):

def another\_function(num1, num2):

return num1 + num2

return another\_function

total = function\_one(10,20)

print(total(10,20))

def function\_one(num1, num2):

def another\_function(num1, num2):

return num1 + num2

return another\_function(num1, num2)

total = function\_one(10,20)

print(total)

Methods vs functions

print()

list()

input()

Method needs to be owned by something (right side of the dot) object or datatype

.capitalize()

.count()

Docstrings

‘’’

Info about the function

‘’’

help(test\_function) → what a function does

print(test.\_\_doc\_\_)

Magic method / dunder method

Args and kwargs

Rule of arguments - actual params, \*args, default parameters, \*\*kwargs

Highest even number

Walrus operator (assignment expressions)

Scope

Scope rules

1 start with a local

2 is there a parent local scope

3. Global – indentation with nothing

4. builtin python functions

Parameters are considered as local to function

Global keyword →

Dependency injection

Nonlocal keyword →

Why do we need scoping?

Imposter syndrome !

Code formatting PEP8

Ternary operator