# Unit 1 ( Introduction, Variables, Expressions and Statements )

## History of python

**1:** The programming language Python was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to ABC capable of exception handling and interfacing with the Amoeba operating system.
**2:** Python 2.0 was released on October 16, 2000, with many major new features, including a cycle-detecting garbage collector (in addition to reference counting) for memory management and support for Unicode.
**3:** However, the most important change was to the development process itself, with a shift to a more transparent and community-backed process.
**4:** Python 3.0, released in December 2008, was indeed a major, backwards-incompatible release. It introduced several fundamental changes to the language syntax and semantics, aiming to improve clarity and remove inconsistencies.

---

## Compiler

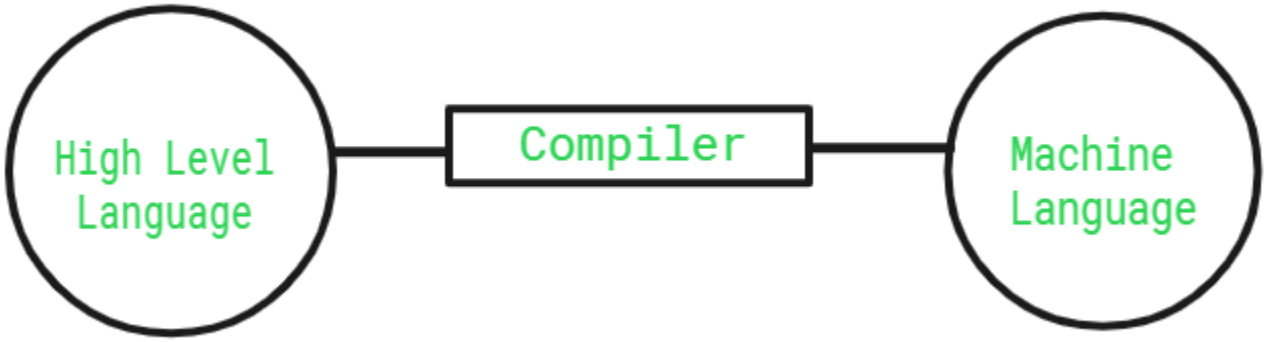
**1:** For Converting the code written in a high-level language into machine-level language so that computers can easily understand, we use a compiler.

**2:** Compiler basically converts high-level language to intermediate assembly language by a compiler and then assembled into machine code by an assembler.

**3:** A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors, etc.

**4:** But its program run time is more and occupies a larger part of memory.

**5:** It has a slow speed because a compiler goes through the entire program and then translates the entire program into machine codes.




---

**Advantages of Compiler**

**1:** Compiled code runs faster in comparison to Interpreted code.

**2:** Compilers help in improving the security of Applications.

**3:** As Compilers give Debugging tools, which help in fixing errors easily.

---

**Disadvantages of Compiler**

**1:** The compiler can catch only syntax errors and some semantic errors.

**2:** Compilation can take more time in the case of bulky code.

---

**Interpreter**

**1:** All high-level languages need to be converted to machine code so that the computer can understand the program after taking the required inputs.

**2:** The software by which the conversion of the high-level instructions is performed line-by-line to machine-level language, other than compiler and assembler, is known as INTERPRETER.

**3.** The interpreter in the compiler checks the source code line-by-line and if an error is found on any line, it stops the execution until the error is resolved.

**4:** Error correction is quite easy for the interpreter as the interpreter provides a line-by-line error.

**5:** But the program takes more time to complete the execution successfully


---

**Advantages of Interpreter**

**1:** Programs written in an Interpreted language are easier to debug.

**2:** Interpreters allow the management of memory automatically, which reduces memory error risks.

**3:** Interpreted language is more flexible than a Compiled language.

---

**Disadvantages of Interpreter**

**1:** The interpreter can run only the corresponding Interpreted program.

**2:** Interpreted code runs slower in comparison to Compiled code.

---

**Debugging**

**1:** Debugging is the process of identifying and resolving errors, or bugs, in a software system.

**2:** It is an important aspect of software engineering because bugs can cause a software system to malfunction, and can lead to poor performance or incorrect results.

**3:** Debugging can be a time-consuming and complex task, but it is essential for ensuring that a software system is functioning correctly.

---

**techniques used in debugging**

**1:** Code Inspection: This involves manually reviewing the source code of a software system to identify potential bugs or errors.

**2:** Debugging Tools: There are various tools available for debugging such as debuggers, trace tools, and profilers that can be used to identify and resolve bugs.

**3:** Unit Testing: This involves testing individual units or components of a software system to identify bugs or errors.

**4:** Integration Testing: This involves testing the interactions between different components of a software system to identify bugs or errors.

**5:** System Testing: This involves testing the entire software system to identify bugs or errors.

**6:** Monitoring: This involves monitoring a software system for unusual behavior or performance issues that can indicate the presence of bugs or errors.

**7:** Logging: This involves recording events and messages related to the software system, which can be used to identify bugs or errors.

---

**Data types**

**1:** Integers: Whole numbers without a decimal point.

**2:** Floats: Numbers with decimal points.

**3:** Strings: Sequences of characters, like text.

**4:** Lists: Ordered collections of items, which can be of different types.

**5:** Tuples: Ordered, immutable collections of items.

**6:** Dictionaries: Unordered collections of key-value pairs.

**7:** Sets: Unordered collections of unique items.

**8:** Booleans: Represents either True or False.

---

**keywords**

Keywords in Python are reserved words that have special meanings and are an integral part of the Python programming language. These keywords cannot be used as identifiers (variable names or function names) because they are used to define the structure and control flow of a Python program.

| Keyword | Description |
|---|---|
| and | A logical operator |
| as | To create an alias |
| assert | For debugging |
| break | To break out of a loop |
| class | To define a class |
| continue | To continue to the next iteration of a loo |
| def | To define a function |
| del | To delete an object |
| elif | Used in conditional statements, same as |
| else | Used in conditional statements |
| except | Used with exceptions, what to do when |
| False | Boolean value, result of comparison ope |

### Statement

In programming, statements are individual instructions or commands that makeup a program. Python, like most programming languages, consists of various types of statements that control the flow of a program, perform actions, and define the structure of your code.

**Assignment Statement**: Assigns a value to a variable.

```
x = 10
```

**Expression Statement**: Executes an expression, and the result is sometimes discarded.

```
2 + 3  # This is an expression statement, and the result is not assigned to a variable.
```

**Conditional Statements (if, elif, else)**: Used to make decisions based on conditions.

```
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

---

### Operators and operands

In programming, operators are symbols that represent operations or actions that can be performed on one or more values, called operands. Operators allow you to manipulate and perform computations on data in your programs. Here's a list of common operators in programming, along with explanations and examples:

*Follow this link*

---

### Modulus operator

In Python, the modulus operator is represented by the percent sign `%`. It is used to find the remainder when one number is divided by another. The syntax for using the modulus operator is as follows.

```
result = dividend % divisor
```

Here, `dividend` is the number you want to find the remainder for, and `divisor` is the number you are dividing `dividend` by. The result will be the remainder of the division operation.

Here's an example:

```
# Using the modulus operator to find the remainder
dividend = 10
divisor = 3
remainder = dividend % divisor
print(remainder)  # This will print 1, because 10 divided by 3 is 3
with a remainder of 1
```

---

### String

A String is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it. Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

---

### string operations

Concatenation: You can concatenate (combine) two or more strings together using the `+` operator.

```
str1 = "Hello"
str2 = " World"
result = str1 + str2
print(result)  # This will print "Hello World"
```

**Repetition**: You can repeat a string multiple times using the `*` operator.

```
str1 = "Hello"
repeated_str = str1 * 3
print(repeated_str)  # This will print "HelloHelloHello"
```

**Length**: To find the length (number of characters) of a string, you can use the `len()` function.

```
my_string = "Python"
length = len(my_string)
print(length)  # This will print 6
```

**Indexing and Slicing**: You can access individual characters of a string by their index or extract a portion (substring) of a string using slicing.

```
my_string = "Python"
first_char = my_string[0]  # Accessing the first character, which is
'P'
substring = my_string[1:4]  # Extracting a substring, which is 'yth'
```

**String Conversion**: You can convert other data types to strings using the `str()` function.

```
num = 42
num_str = str(num)  # Converts the integer 42 to the string "42"
```

---

**input statements**

Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with two inbuilt functions to read the input from the keyboard.

- When input() function executes, program flow will be stopped until the user has given input.
- The text or message displayed on the output screen to ask a user to enter an input value is optional i.e. the prompt, which will be printed on the screen is optional.
- Whatever you enter as input, the input function converts it into a string. if you enter an integer value still input() function converts it into a string. You need to explicitly convert it into an integer in your code using *typecasting/*.

---

**Comments**

In Python, you can add comments to your code to provide explanations or documentation for yourself and others who might read your code. Comments are ignored by the Python interpreter and are meant for human readability. There are two primary ways to add comments in Python:

**Single-Line Comments**: Single-line comments start with the `#` character and continue until the end of the line. Everything following `#` on the same line is treated as a comment and is not executed by the Python interpreter.

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

**Multi-Line Comments (Docstrings)**: While Python doesn't have a built-in syntax for multi-line comments like some other programming languages, you can use triple-quotes (`'''` or `"""`) to create multi-line strings, which are often used for documentation (docstrings). While these are not technically comments, they serve a similar purpose and are often used for documenting functions, classes, and modules.

```
'''This is a multi-line docstring.It/ serves as a comment and
documentation.'''
```

---

### Choosing mnemonic

**1:** Use Descriptive Names: Choose variable names that describe the purpose or content of the variable. Avoid single-letter variable names (e.g., `x`, `i`, `j`) unless they are used in very short and well-defined contexts like loop counters.

**2:** Follow Conventions: Adhere to naming conventions to make your code consistent with the Python community. The most common convention is using lowercase letters for variable names with words separated by underscores (snake_case).

**3:** Be Consistent: Maintain a consistent naming style throughout your codebase. If you use a particular naming convention or style, stick with it.

**4:** Avoid Reserved Words: Don't use Python's reserved words or built-in function names as variable names. For example, avoid names like `print`, `for`, `if`, `while`, etc.

**5:** Use Meaningful Names for Loops: When using loop counters, consider using names that indicate what the loop is iterating over or what it's doing.

**6:** Use Singular and Plural Forms Appropriately: When naming collections or objects, use the singular form for individual instances and the plural form for collections.

# Unit 2 ( Conditional Execution and Catching Exceptions )

## Booelan expressions

The **Python Boolean** type is one of Python's *built-in data types/*. It's used to represent the truth value of an expression.

Example

Print a message based on whether the condition is `True` or `False`:

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
output:
b is not greater than a
#which indicates that it is false
```

---

## conditional execution

**1:** Conditional execution refers to the concept in programming and computer science where certain code blocks or instructions are executed based on specified conditions or criteria.

**2:** This allows programs to make decisions and take different actions depending on the values of variables, user input, or other factors.

**3:** In most programming languages, conditional execution is achieved using conditional statements, the most common of which are "if" statements.

**4:** These statements allow you to specify a condition, and if that condition is true, the code block associated with the "if" statement is executed.

**5:** If the condition is false, the code block may be skipped or an alternative code block associated with an "else" statement might be executed.

example:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

```
else:
    print("x is not greater than 5")
```

---

## Alternative execution

**1:** Alternative execution, also known as branching or conditional branching, is a programming concept that involves executing different blocks of code based on the evaluation of specific conditions.

**2:** This allows a program to take one of several possible paths of execution depending on the values of variables, user input, or other factors.

**3:** The most common way to achieve alternative execution is through the use of conditional statements, such as "if" statements and "else" statements. **4:** However, unlike standard "if" statements that execute one block of code when a condition is true, alternative execution involves executing one block of code if a condition is true and a different block of code if the condition is false.

```
x = 15
if x > 10:
    print("x is greater than 10")
else:
    print("x is not greater than 10")
```

---

## Chained conditionals

**1:** Chained conditionals, also known as nested conditionals, refer to the practice of using multiple layers of conditional statements within a program to create more complex decision-making logic.

**2:** This involves using combinations of "if" statements, "else if" (or "elif") statements, and "else" statements to handle various possible cases and outcomes.

**3:** Chained conditionals allow you to consider multiple conditions in a hierarchical manner, executing different code blocks based on the combination of conditions that are true.

**4:** This is particularly useful when you need to address various levels of decision-making within your program.

example:

```
x = 7
y = 5
if x > 10:
    if y > 10:
        print("x and y are both greater than 10")
    else:
        print("x is greater than 10, but y is not")
elif x > 5:
    print("x is greater than 5 but not greater than 10")
else:
    print("x is not greater than 5")
```

In this example, the program first evaluates whether `x` is greater than 10. If that condition is met, it then evaluates whether `y` is also greater than 10. Depending on the combination of values for `x` and `y`, different code blocks will be executed.

---

## nested conditionals

**1:** Nested conditionals, also known as nested if statements, are a programming construct where one "if" statement is contained within another "if" statement.

**2:** This allows for more complex decision-making scenarios where the outcome depends on multiple conditions or levels of conditions.

**3:** Nested conditionals are useful when you need to consider secondary conditions only if a primary condition is true.

**4:** They can also help you handle various cases and outcomes in a structured manner.

**5:** However, care should be taken when using nested conditionals, as excessive nesting can make code harder to read and maintain.

example:

```
x = 7
y = 5
if x > 5:
    print("x is greater than 5")
    if y > 5:
        print("y is greater than 5")
    else:
        print("y is not greater than 5")
```

```
else:
    print("x is not greater than 5")
```

---

## error handling

Error handling in Python involves using mechanisms to detect and manage exceptions that can occur during the execution of a program. Python provides a robust set of tools for handling errors and exceptions, allowing you to gracefully handle unexpected situations and prevent your program from crashing.

**Try-Except Blocks:** The core of error handling in Python is the `try` and `except` block. The `try` block contains the code that might raise an exception, and the `except` block catches and handles the exception. You can have multiple `except` blocks to handle different types of exceptions.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero")
```

**Catching Specific Exceptions:** You can catch specific exceptions using the appropriate exception class. For example, the `ZeroDivisionError` class handles division by zero errors.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input. Please enter a valid number.")
except ZeroDivisionError:
    print("Error: Division by zero")
```

**Handling Multiple Exceptions:** You can catch multiple exceptions using a single `except` block by using parentheses and specifying the exception classes.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError):
    print("An error occurred")
```

**Handling Any Exception:** If you want to catch any exception, you can use a bare `except` block, but this is generally discouraged as it makes debugging difficult.

```
try:
    result = 10 / 0
except:
    print("An error occurred")
```

**Finally Block:** You can use the `finally` block to specify code that should be executed regardless of whether an exception occurred or not. For example, closing files or resources.

```
try:
    file = open("example.txt", "r")
    # perform operations on the file
except FileNotFoundError:
    print("File not found")
finally:
    file.close()
```

**Raising Exceptions:** You can intentionally raise exceptions using the `raise` statement. This can be useful when you want to signal an error or exceptional condition in your code.

```
try:
    num = int(input("Enter a positive number: "))
    if num <= 0:
        raise ValueError("Number must be positive")
except ValueError as e:
    print(e)
```

# Unit 3 ( Function and Iterations )

## What is a function

**1:** Functions is a block of statements that return the specific task.
**2:** The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.
**3:** Functions Increase Code Readability and Increase Code Reusability.
**4:** Python has *Built-in library function:* These are *Standard functions/* in Python that are available to use. and *User-defined function:* We can create our own functions based on our requirements.

---

## function calls

**1:** In Python, a function call is a way to execute or invoke a function that has been defined elsewhere in your code or in a library.

**2:** Function calls are used to perform specific tasks or operations encapsulated within a function.

**3:** Here's an example of a simple function call:

```
def my_function():
  print("Hello from a function")
my_function()
```

---

## built in functions

**1:** print(): Used to display output on the console.

**2:** input(): Reads a line of text from the user via the console.

**3:** len(): Returns the number of items in an iterable (e.g., string, list, tuple, dictionary).

**4:** type(object): Returns the data type of an object.

**5:** int(), float(), str(): Converts a value to an integer, float, or string, respectively.

**6:** range(start, stop, step): Generates a sequence of numbers from `start` to `stop`, with an optional `step` value.

**7:** list(iterable), tuple(iterable), set(iterable): Converts an iterable into a list, tuple, or set, respectively.

**8:** dict(key=value, ...): Creates a dictionary with specified key-value pairs.

**9:** sorted(iterable):  Returns a sorted list from the elements of an iterable.

**10:** round(number, ndigits): Rounds a floating-point number to a specified number of decimal places.

---

# Type conversion functions

Type conversion functions in Python allow you to convert data from one type to another. They are built-in functions that are commonly used when you need to change the data type of a value. Here are some frequently used type conversion functions:

1. int(x): Converts `x` to an integer. For example:

```
 num_str = "42"
   num_int = int(num_str)
```

2. float(x): Converts `x` to a floating-point number. For example:

```
num_str = "3.14"
   num_float = float(num_str)
```

3. str(x): Converts `x` to a string. For example:

```
  num = 42
   num_str = str(num)
```

4. bool(x): Converts `x` to a Boolean value (`True` or `False`). Commonly used to check if a value is "truthy" or "falsy." For example:

```
 result = bool(0)  # Result is False
   result = bool(42)  # Result is True
```

5. list(iterable): Converts an iterable (e.g., a tuple or string) into a list. For example:

```
tuple_data = (1, 2, 3)
   list_data = list(tuple_data)
```

6. tuple(iterable): Converts an iterable into a tuple. For example:

```
 list_data = [1, 2, 3]
   tuple_data = tuple(list_data)
```

7. set(iterable):  Converts an iterable into a set. For example:

```
 list_data = [1, 2, 2, 3, 3, 3]
   set_data = set(list_data)
```

8. dict(iterable): Converts an iterable of key-value pairs into a dictionary. For example:

```
key_value_pairs = [("a", 1), ("b", 2)]
dictionary = dict(key_value_pairs)
```

9. chr(x)`: Converts an integer `x` into a Unicode character. For example:

```
   unicode_char = chr(65)  # Result is 'A'
```

10. ord(x): Converts a Unicode character `x` into its corresponding integer Unicode code point. For example:

```
unicode_code_point = ord('A')  # Result is 65
```

---

# Math  functions

Basic Mathematical Operations:

- `math.sqrt(x)`: Returns the square root of `x`.
- `math.pow(x, y)`: Returns `x` raised to the power of `y`.
- `math.exp(x)`: Returns the exponential value of `x` (e^x).
- `math.log(x, base)`: Returns the natural logarithm (base e) of `x`. You can specify the base as an optional second argument.
- `math.isinf(x)`: Checks if `x` is positive or negative infinity

---

# random numbers

**`1: random.random()`**: Generates a random float between 0 and 1 (inclusive of 0, exclusive of 1).

```
import random
rand_num = random.random()
```

**`2: random.randint(a, b)`**: Generates a random integer between `a` (inclusive) and `b` (inclusive)

```
import random
rand_int = random.randint(1, 10)  # Generates a random integer between
1 and 10 (inclusive).
```

**`3: random.uniform(a, b)`**: Generates a random float between `a` and `b`, where both `a` and `b` are inclusive.

```
import random
rand_float = random.uniform(2.5, 5.5)  # Generates a random float
between 2.5 and 5.5 (inclusive).
```

**`4: random.choice(seq)`**: Selects a random element from a sequence (e.g., a list or tuple).

```
import random
my_list = [1, 2, 3, 4, 5]
rand_choice = random.choice(my_list)  # Selects a random element from
my_list.
```

**`5: random.shuffle(seq)`**: Shuffles the elements of a sequence randomly (in-place).

```
import random
my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)  # Shuffles the elements of my_list randomly.
```

**`6: random.sample(population, k)`**: Returns `k` unique random elements from a population without replacement.

```
import random
```

```
population = [1, 2, 3, 4, 5]
random_sample = random.sample(population, 3)  # Returns 3 unique
random elements from population.
```

---

# Adding functions in python

1. Use the *def* keyword to begin the function definition.
2. Name your function.
3. Supply one or more parameters. *Actually parameters are optional, but the parentheses are not.* Followed by a colon.
4. Enter lines of code that make your function do whatever it does. Enter the code where the *# logic here* is shown in the sample code below.
5. Use the return keyword at the end of the function to return the output. This is also optional. If omitted, the function will return None.

```
def greet(name):
    """This function greets the person passed in as a parameter."""
    print("Hello, " + name + "!")
# Call the function
greet("Alice")
```

---

# Uses of Functions in Python:

**1:** Modularization: Functions allow you to break down a complex program into smaller, more manageable pieces. Each function can be responsible for a specific task, making the code easier to understand and maintain.

**2:** Reusability: Once you've defined a function, you can call it as many times as needed throughout your program. This promotes code reuse and reduces duplication.

**3:** Abstraction: Functions hide the implementation details of a task, allowing you to use them without needing to understand how they work internally.

**4:** Parameterization: Functions can take parameters as input, allowing you to customize their behavior based on the values you pass in.

**5:** Recursion: Functions can call themselves, allowing for elegant solutions to problems that involve repetitive or recursive behavior.

---

# flow of execution of functions

**1:** In Python, functions are defined with the 'def' keyword, including a name and optional parameters, followed by a colon.

**2:** The indented function body contains the code executed when the function is called.

**3:** To execute the function, you simply call it by name, providing any required arguments in parentheses.

**4:** Functions create their own local scope, isolating internal variables from the outside.

**5:** Code within the function runs sequentially, including statements, calculations, and control flow structures.

**6:** A 'return' statement, if present, specifies values to be sent back to the caller, after which the program continues from where the function was called, with local variables inaccessible unless explicitly returned.

---

# Parameters and Arguments

*Parameters*

Definition: Parameters are variables or placeholders defined in the function or method signature. They act as input slots that the function expects to receive when it is called.

Purpose: Parameters specify the kind and order of data that a function requires to perform its task. They define the interface of the function, indicating what data should be provided for it to work correctly.

Example:

```
def greet(name):
   print("Hello, " + name + "!")
```

In this example, `name` is a parameter of the `greet` function, and it represents the data the function needs to execute.

*Arguments:*

Definition: Arguments are the actual values or data passed into a function when it is called. These values are supplied in the function call and correspond to the parameters defined in the function's signature.

Purpose: Arguments are the concrete data that fulfill the requirements of the function's parameters. They provide the actual values to be used within the function's code.

Example:

```
greet("Alice")
```

Here, "Alice" is an argument passed to the `greet` function. It corresponds to the `name` parameter defined in the function.

---

# Fruitful Function

**1:** A "fruitful function" in programming, also commonly known as a "function that returns a value," is a type of function that performs some computation or task and then returns a result or value as its output.

**2:** These functions are designed to produce an outcome that can be used elsewhere in a program.

**3:** Here are some key characteristics and examples of fruitful functions:

Return Statement: Fruitful functions use the `return` statement to specify the value they are going to produce and send back to the caller.

Result Usage: The value returned by a fruitful function can be assigned to a variable, used in an expression, or otherwise manipulated by the calling code.

---

# void funtions

**1:** In Python, functions that do not return a value are often referred to as "void functions" or "functions with no return value."

**2:** These functions perform a specific task or set of tasks but do not produce any output that can be assigned to a variable or used in expressions.

**3:** Instead, they may have side effects, such as modifying data or performing actions without returning a result.

Here's how void functions work in Python:

1. No Return Statement**:** Void functions do not contain a `return` statement, or if they do, it does not specify any value to be returned. Instead, they may have `return` statements with no value, which effectively means they return `None` (Python's representation of "no value").
2. Side Effects: Void functions often perform actions or modify data within the function itself or in the program's environment. These actions can include printing information, updating variables, interacting with files or databases, or altering the program's state in some way.

**Example:**

```
def greet(name):
    """This void function greets the person passed in as a
parameter."""
    print("Hello, " + name + "!")
```

```
# Calling the void function
greet("Alice")
```

---

# Updating variables in python

**1:** Simple Assignment:

You can update a variable by assigning it a new value using the assignment operator '='

```
x = 5  # Assign the value 5 to the variable x
x = 10  # Update the value of x to 10
```

**2:** Arithmetic Operations:

Variables can be updated using arithmetic operations, which combine the current value of the variable with a new value.

```
count = 0
count += 1  # Increment count by 1
count -= 2  # Decrement count by 2
count *= 3  # Multiply count by 3
```

**3:** String Concatenation:

When working with strings, you can update variables by concatenating them with new strings.

```
message = "Hello, "
name = "Alice"
greeting = message + name  # Combine message and name
```

**4:** List and Dictionary Updates:

Lists and dictionaries allow you to update their elements or values by index or key.

```
my_list = [1, 2, 3]
my_list[1] = 4  # Update the second element of the list to 4
my_dict = {"name": "Alice", "age": 30}
my_dict["age"] = 31  # Update the value associated with the key "age"
```

**5:** Using Functions:

Functions can update variables by performing calculations and returning new values.

```
def double_value(x):
    return x * 2
number = 5
number = double_value(number)  # Update number by doubling its value
```

---

# while loop

**1:** a `while` loop is a control structure that allows you to repeatedly execute a block of code as long as a specified condition remains true. The basic syntax of a `while` loop is as follows:

```
while condition:
```

```
    # Code to be executed while the condition is True
```
Here's how a `while` loop works:

1. Condition: The `condition` is an expression that is evaluated before each iteration of the loop. If the condition is `True`, the code inside the loop block is executed. If the condition is `False`, the loop terminates, and the program continues with the code after the loop.
2. Loop Block**:** The indented code block under the `while` statement contains the instructions to be executed as long as the condition is `True`. This block can contain any valid Python code, including variable updates and control flow statements.
3. Iteration: The loop continues to execute as long as the condition remains `True`. After each iteration, the condition is re-evaluated. If the condition eventually becomes `False`, the loop exits.

Here's a simple example of a `while` loop that counts from 1 to 5:
```
count = 1
while count <= 5:
    print(count)
    count += 1
```

---

# infinite loops

**1:** An "infinite loop" is a type of programming error where a loop (usually a `while` or `for` loop) continues to execute indefinitely without a condition that can make it stop.

**2:** Infinite loops can lead to programs that never terminate or become unresponsive, which is typically not the desired behavior.

**3:** Here's an example of an infinite loop:

```
while True:
    print("This is an infinite loop")
```
**4:** In this example, the condition `True` is always true, so the loop will keep executing indefinitely, printing "This is an infinite loop" to the console.
**5:** Infinite loops can be problematic and may cause your program to consume excessive CPU resources or become unresponsive.
**6:** They are usually the result of a programming mistake or a missing exit condition.
**7:** To prevent infinite loops, always ensure that your loops have a condition that can eventually become `False`, allowing the loop to exit.
**8:** Common ways to exit loops include using counter variables, user input, or specific conditions based on the problem you're solving.

---

## Finishing iterations with continue

**1:** In Python, the `continue` statement is used within loops (such as `for` and `while` loops) to prematurely end the current iteration and proceed to the next iteration without executing the remaining code within the loop block for the current iteration.

**2:** In other words, when `continue` is encountered, it skips the rest of the current iteration and jumps to the next iteration.

**3:** Here's an example of how the `continue` statement works within a `for` loop:

```
for i in range(1, 6):
    if i == 3:
        continue  # Skip iteration when i is equal to 3
    print("Current value of i:", i)
```

**4:** In this example, when `i` is equal to 3, the `continue` statement is executed, and the loop skips the `print` statement for that iteration. As a result, the output will be:

```
Current value of i: 1
Current value of i: 2
Current value of i: 4
Current value of i: 5
```

---

## definite loops using for

**1:** In Python, definite loops are loops where you have a known, fixed number of iterations.

**2:** The `for` loop is commonly used for definite loops, especially when you want to iterate over a sequence (like a list, tuple, string, or range) or perform a set number of repetitions.

**3:** Here's an example of using a `for` loop to iterate over a sequence:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

**4:** In this example, the `for` loop iterates through the list of fruits, and during each iteration, it assigns the current fruit to the `fruit` variable, which is then printed. The loop runs as many times as there are elements in the `fruits` list.

---

## loop patterns

**1:** Loop patterns are recurring structures or common usage patterns that are often encountered when working with loops in programming.

**2:** These patterns represent specific ways loops can be structured to accomplish various tasks efficiently.

**3:** Here are some common loop patterns in programming:

Counting Loop (Numerical Range):

This pattern involves using a loop to iterate over a numerical range of values, typically generated by the `range()` function. It's often used for a fixed number of iterations.

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

Iterating Over a Sequence:

In this pattern, you use a loop to iterate over the elements of a sequence (e.g., lists, strings, tuples) to process each element.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Loop with Accumulator (Summation):

This pattern involves accumulating a value or result during each iteration of the loop, such as calculating the sum of a series of numbers.

```
total = 0
for i in range(1, 6):
    total += i
```

Loop with Conditional (Filtering):

In this pattern, a loop is used to filter or select elements from a sequence based on a condition

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)
```

Nested Loops:

Nested loops involve one loop inside another. This pattern is useful for working with two-dimensional data structures, like matrices or grids.

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

Loop with Sentinel (Input Processing):

In this pattern, a loop continues to execute until a specific "sentinel" value is entered, often used for input processing.

```
while True:
    user_input = input("Enter a value (or 'q' to quit): ")
    if user_input == 'q':
```

```
        break
    # Process user_input
```

---

# Counting and summing loops

Counting and summing loops are common loop patterns used in programming to count elements or calculate the sum of a series of values. These patterns are essential for various tasks, such as processing data or solving mathematical problems. Here's how counting and summing loops work:

Counting Loop:

A counting loop is used to count elements or perform a fixed number of iterations. You typically use a variable (often called a counter) to keep track of the count. Here's an example of a counting loop:

```
# Counting from 1 to 5
count = 0  # Initialize the counter
for i in range(1, 6):  # Iterate 5 times
    count += 1  # Increment the counter by 1 in each iteration
print("Count:", count)
```

Summing Loop:

A summing loop is used to calculate the sum of a series of values. You initialize a variable (often called a sum or total) to zero and then add each value to the total in each iteration. Here's an example of a summing loop:

```
# Calculating the sum of numbers from 1 to 5
total = 0  # Initialize the total to zero
for i in range(1, 6):  # Iterate 5 times
    total += i  # Add the current value (i) to the total
print("Total:", total)
```

---

# maximum and minimum loops

In Python, you can use loops to repeat a block of code a certain number of times. Two commonly used types of loops for this purpose are the `for` loop and the `while` loop. These loops can be used to find maximum and minimum values from a list of numbers or any iterable. Here's how you can use them:

Finding the Maximum and Minimum Using a `for` Loop:

```
numbers = [5, 2, 9, 1, 7, 3]
# Initialize variables to hold the maximum and minimum values
maximum = numbers[0]
```

```
minimum = numbers[0]
# Iterate through the list to find the maximum and minimum
for num in numbers:
    if num > maximum:
        maximum = num
    if num < minimum:
        minimum = num
print(f"Maximum value: {maximum}")
print(f"Minimum value: {minimum}")
```

Finding the Maximum and Minimum Using a `while` Loop:

```
numbers = [5, 2, 9, 1, 7, 3]
# Initialize variables to hold the maximum and minimum values
maximum = numbers[0]
minimum = numbers[0]
# Using a while loop to find the maximum and minimum
index = 1
while index < len(numbers):
    num = numbers[index]
    if num > maximum:
        maximum = num
    if num < minimum:
        minimum = num
    index += 1
print(f"Maximum value: {maximum}")
print(f"Minimum value: {minimum}")
```

# Unit 4 ( String and Files )

## What  is a String

**1:** When developing software, you often need to interact with users through text input and output.
**2:** Strings are crucial for displaying messages, receiving input, and processing user-generated text.
**3:** A String is a data structure in Python that represents a sequence of characters.
**4:** string is a sequence of characters enclosed within either single (' '), double (" "), or triple ('" "' or """ """) quotes.
**5:** Strings are immutable, meaning their contents cannot be changed once created. You can create a new string with modified content.
**6:** Python provides numerous built-in methods to manipulate strings, such as upper(), lower(), strip(), split(), and more.
**7:** Strings can be formatted using f-strings or methods like format(), allowing you to insert variables into a string.

---

## Getting length of string using len()

In Python, you can get the length of a string (i.e., the number of characters in the string) using the len() function. Here's how you can use it:

```
my_string = "Hello, World!"
length = len(my_string)
print("Length of the string:", length)
```

In this example, len(my_string) will return 13 because there are 13 characters in the string "Hello, World!" including spaces and punctuation. You can replace my_string with any string variable or string literal to find its length.

---

## Traversal through a string with loop

You can traverse through a string in Python using a loop, such as a `for` loop or a `while` loop. Here's an example using a `for` loop to iterate through each character in a string:

```
my_string = "Hello, World!"

# Using a for loop to traverse the string
for char in my_string:
    print(char)
```

In this code, the `for` loop iterates through each character in `my_string`, and `char` represents the current character in each iteration. The loop will print each character one by one.

You can also use a `while` loop for more control over the traversal process:

```
my_string = "Hello, World!"

# Using a while loop to traverse the string
index = 0
while index < len(my_string):
    print(my_string[index])
    index += 1
```

In this `while` loop, we use an `index` variable to keep track of the current position in the string. The loop continues until `index` reaches the length of the string, printing each character along the way.

---

## String slicing

String slicing in Python allows you to extract a portion of a string by specifying a range of indices. The basic syntax for string slicing is as follows:

```
string[start:stop:step]
```

*start:* The starting index (inclusive) from where the slicing begins.
*stop:* The stopping index (exclusive) where the slicing ends.

*step (optional)*: The step or stride to skip characters while slicing. It's typically 1 by default.

Here are some examples of string slicing:

```
my_string = "Hello, World!"

# Extracting a substring from index 0 to 4 (not including 4)
substring = my_string[0:4]
print(substring)  # Output: "Hell"

# Slicing with a step of 2 to get every other character
every_other = my_string[0::2]
print(every_other)  # Output: "Hlo ol!"

# Slicing to reverse a string (with a step of -1)
reversed_string = my_string[::-1]
print(reversed_string)  # Output: "!dlroW ,olleH"
```

In the first example, we slice the string from index 0 to 4 to get the substring "Hell." In the second example, we use a step of 2 to get every other character. In the third example, we use a step of -1 to reverse the string.

Remember that indices in Python are zero-based, and the `start` index is inclusive, while the `stop` index is exclusive, meaning it includes the character at `start` but not the character at `stop`.

---

## Immutability in strings

strings are immutable. This means that once you create a string, you cannot change its content. If you want to modify a string, you actually create a new string with the desired changes.

Here's an example to illustrate string immutability:

```
my_string = "Hello"
```

```
# Attempt to change a character in the string (This will raise
an error)
my_string[0] = 'J'
```

If you run the code above, you will encounter a TypeError because you cannot assign a new character to an existing position in the string.

To modify a string, you would typically create a new string by concatenating or using string manipulation methods like replace(), join(), or slicing to create the desired result. For example:

```
# Creating a new string with the desired change
new_string = "J" + my_string[1:]
print(new_string)  # Output: "Jello"
```

---

## Looping and counting

Looping and counting are common tasks in Python programming. You can use loops to iterate through a sequence (e.g., a string, list, or range of numbers) and count occurrences of specific elements or conditions. Here's a basic example of looping and counting in Python:

```
# Sample list of numbers
numbers = [1, 2, 3, 4, 2, 5, 2]

# Initialize a counter variable
count = 0

# Loop through the list and count occurrences of the number 2
for num in numbers:
    if num == 2:
        count += 1

# Print the count
print("The number 2 appears", count, "times in the list.")
```

In this example:

- We have a list of numbers called `numbers`.
- We initialize a `count` variable to keep track of the occurrences of the number 2.
- We use a `for` loop to iterate through each element in the `numbers` list.
- Inside the loop, we check if the current element (`num`) is equal to 2. If it is, we increment the `count` variable.

After the loop, we print out the count, which tells us how many times the number 2 appears in the list. You can adapt this basic pattern to count occurrences or perform other actions based on your specific requirements.

---

# The in operator

**1:** In Operator in Python comes under the category of Membership Operators.

**2:** The Membership Operators check whether the sequence appears in the object or not.

**3:** So, in Operator in Python is an important operator as it can be used in a variety of scenarios, including searching for specific values, filtering elements based on certain conditions, and checking for membership in a set or dictionary.

**4:** The in operator can be used in conditional statements, loops, and other places where you need to check if a particular element is present in a sequence.

**5:** The syntax of in operator in Python is:

```
element in sequence
```

**6:** Here, the “element” is the value that you want to search for in the sequence.
**7:** The “sequence” is the list, tuple, string, set, or dictionary in which you want to search for the element

Examples:
1: Checking for Existence in a String:

```
my_string = "Hello, World!"
if "World" in my_string:
    print("The word 'World' exists in the string.")
```
2: Checking for Non-Existence with not in:
```
my_list = [1, 2, 3, 4, 5]
if 6 not in my_list:
    print("The number 6 does not exist in the list.")
```

---

## String comparisons

In Python, you can compare strings using various comparison operators to determine their relative order or equality. Here are the common string comparison operations:

*1. Equality (`==`):*

You can use the `==` operator to check if two strings are equal.

```
str1 = "apple"
str2 = "apple"

   if str1 == str2:
       print("Both strings are equal.")
```

*2. Inequality (`!=`):*
The `!=` operator checks if two strings are not equal.

```
str1 = "apple"
str2 = "banana"

   if str1 != str2:
       print("The strings are not equal.")
```

*3. Greater Than (`>`), Less Than (`<`), Greater Than or Equal To (`>=`), Less Than or Equal To (`<=`):*
These operators allow you to compare strings based on their lexicographical (dictionary) order.

```
str1 = "apple"
str2 = "banana"

   if str1 < str2:
       print("str1 comes before str2 in the dictionary.")
```

Note that the comparison is case-sensitive, so uppercase letters come before lowercase letters.

*4. Case-Insensitive Comparison:*
If you want to perform a case-insensitive comparison, you can convert both strings to lowercase (or uppercase) before comparing them.

```
str1 = "Apple"
str2 = "apple"

   if str1.lower() == str2.lower():
       print("Both strings are equal (case-insensitive).")
```

*5. Substring Check :*
You can use the `in` operator to check if one string is a substring of another.

```
main_str = "Hello, World!"
substring = "World"

   if substring in main_str:
       print("The substring exists in the main string.")
```

These are the basic ways to perform string comparisons in Python. Keep in mind that string comparisons are based on the Unicode values of the characters in the strings, so they work for a wide range of characters and languages.

---

# string methods

| Method | Description |
| --- | --- |
| capitalize() | Converts the first character to upper case |
| casefold() | Converts string into lower case |
| center() | Returns a centered string |
| count() | Returns the number of times a specified value occurs |
| encode() | Returns an encoded version of the string |
| endswith() | Returns true if the string ends with the specified valu |
| expandtabs() | Sets the tab size of the string |
| find() | Searches the string for a specified value and returns |
| format() | Formats specified values in a string |
| format_map() | Formats specified values in a string |
| index() | Searches the string for a specified value and returns |
| isalnum() | Returns True if all characters in the string are alphanu |
| isalpha() | Returns True if all characters in the string are in the a |
| isascii() | Returns True if all characters in the string are ascii ch |

---

**parsing strings**

The `split()` method in Python is used to split a string into a list of substrings based on a specified delimiter. Here's how it works:

```
string.split([delimiter[, maxsplit]])
```

1: string: The original string that you want to split.
2: delimeter (optional): The character or sequence of characters used as the delimiter to split the string. If not provided, the string is split by whitespace (spaces, tabs, and newline characters) by default.
3: Maxsplit (optional): This parameter specifies the maximum number of splits to perform. It's an integer value. If not provided, there is no maximum limit, and the string is split as many times as possible.

*String Splitting:*

- `split()`: Splits a string into a list of substrings based on a specified delimiter.

```
text = "apple,banana,kiwi"
fruits = text.split(',')
print(fruits)  # Output: ['apple', 'banana', 'kiwi']
```

*Splitting by Whitespace:*

- You can split a string into words by whitespace using `split()` without specifying a delimiter.

```
text = "This is a sample sentence"
words = text.split()
print(words)  # Output: ['This', 'is', 'a', 'sample', 'sentence']
```

---

# Format operator

The format operator in Python is represented by the `%` symbol and is used for formatting strings by inserting values into placeholders within the string. This is commonly referred to as "string formatting" or "old-style string formatting." The format operator allows you to create formatted strings where placeholders are replaced by values.

Here's the basic syntax of the format operator:

```
formatted_string = "Format this string with %s and %d" %
(string_value, integer_value)
```

- `formatted_string`: The resulting string with placeholders replaced by the specified values.
- `%s`: Placeholder for a string.
- `%d`: Placeholder for an integer.

You can use multiple placeholders in a single string and provide corresponding values as a tuple on the right side of the `%` operator.
Here's an example:

```
name = "Alice"
age = 30
formatted_string = "My name is %s, and I am %d years old" % (name,
age)
print(formatted_string)
```

Here are some common format codes:

- `%s`: String
- `%d`: Integer
- `%f`: Floating-point number
- `%.2f`: Floating-point number with 2 decimal places
- `%x`: Integer in hexadecimal format

the format operator is still available in Python, the newer and more flexible string formatting methods using f-strings and the `.format()` method are generally recommended, especially in Python 3.6 and later versions. F-strings provide a more concise and readable way to format strings in modern Python.

---

# presistence

Persistence in files refers to the ability to store data in a file system for future retrieval and use. It allows data to be preserved between program executions and is crucial for applications that need to maintain state or save information for later reference. Here's a concise point-wise explanation of persistence in files:

**1:** Data Storage: Persistence in files involves saving data to a storage medium, typically a file on a file system, so that it can be accessed even after the program or system has terminated.

**2:** File Handling: Python provides file handling mechanisms, such as the `open()` function, for reading from and writing to files. These mechanisms allow developers to interact with files programmatically.

**3:** Modes: Files can be opened in different modes, such as read (`'r'`), write (`'w'`), and append (`'a'`). These modes determine whether the file is read-only, write-only, or open for both reading and writing.

**4:** Context Management: Using the `with` statement for file handling is a best practice. It ensures that files are automatically closed when they are no longer needed, preventing resource leaks and ensuring data integrity.

**5:** Structured Data: For structured data storage, Python offers libraries like `csv` for handling comma-separated values and `json` for working with JSON data, simplifying the process of reading and writing structured data to and from files.

---

# opening files

Opening files in Python involves using the `open()` function to establish a connection between your Python program and a file on the file system. This function returns a file object that you can use to read from or write to the file. Here's how you open files in Python:

```
file_object = open(file_path, mode)
```
`file_path`: The path to the file you want to open, either as an absolute path or a relative path.
`mode`: The mode in which you want to open the file, which can be one of the following:

**1:** `'r'`: Read mode (default). Opens the file for reading.
**2:** `'w'`: Write mode. Opens the file for writing. If the file doesn't exist, it creates a new empty file. If it does exist, it truncates (empties) the file before writing.
**3:** `'a'`: Append mode. Opens the file for writing, but appends data to the end of the file instead of truncating it.
**4:** `'b'`: Binary mode. Used in conjunction with `'r'`, `'w'`, or `'a'` to indicate that the file should be treated as a binary file (e.g., for reading or writing images or non-text data).
**5:** `'t'`: Text mode (default). Used in conjunction with `'r'`, `'w'`, or `'a'` to indicate that the file should be treated as a text file (e.g., for reading or writing text data).

Here's an example of opening a file in read mode, reading its content, and then closing it:
```
# Open a file in read mode
```

```
file_path = 'example.txt'
file = open(file_path, 'r')

# Read the contents of the file
content = file.read()
print(content)

# Close the file when done
file.close()
```

---

# Text files and lines

Text Files:

- Text files are a type of computer file that stores data in the form of human-readable text. Each character in the file corresponds to a specific symbol, letter, or digit, and the content can be understood by humans.
- Text files can contain various types of information, such as plain text, configuration settings, code, logs, and more.
- Examples of text file formats include `.txt`, `.csv` (comma-separated values), `.html` (Hypertext Markup Language), `.json` (JavaScript Object Notation), and `.xml` (eXtensible Markup Language).

lines in Text Files:

- Lines are fundamental components of text files. Each line typically represents a separate unit of information or a record. Lines are delimited by newline characters (such as '\n' in Unix-based systems or '\r\n' in Windows).
- In a text file, lines are often used to organize and structure data. For example, in a CSV file, each line represents a row of data, and the values within a row are separated by commas.
- Lines in text files can vary in length and content. They may contain text, numbers, whitespace, or any combination of characters.

```
# Example of reading lines from a text file in Python
with open('sample.txt', 'r') as file:
    line = file.readline()  # Read the first line
    while line:
        print(line.strip())  # Remove newline characters and print the
line
        line = file.readline()  # Read the next line
```

```
# Example of writing lines to a text file in Python
with open('output.txt', 'w') as file:
    lines_to_write = ["Line 1", "Line 2", "Line 3"]
    for line in lines_to_write:
        file.write(line + '\n')
```

---

# Reading files

Reading files in Python involves opening a file, reading its content, and processing the data. You can read the entire file at once, read it line by line, or use other methods depending on your specific needs. Here's how you can read files in Python:

1. Reading the Entire File:

- To read the entire content of a file into a string variable, you can use the `read()` method on the file object.

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

2: Reading Lines from a File:
To read a file line by line, you can use the `readline()` method or iterate over the file object directly.
Using `readline()`:

```
with open('example.txt', 'r') as file:
    line = file.readline()
    while line:
        print(line.strip())  # Use strip() to remove newline
characters
        line = file.readline()
```

3: Reading All Lines into a List:
To read all lines from a file into a list, you can use the `readlines()` method. Each element of the list represents a line from the file.

```
with open('example.txt', 'r') as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())  # Use strip() to remove newline
characters
```

4: Binary File Reading:
For reading binary files (e.g., images, audio files), use `'rb'` mode instead of `'r'`.

```
with open('image.jpg', 'rb') as binary_file:
    binary_data = binary_file.read()
```

---

# Searching through a file

**1:** Python can search for file names in a specified path of the OS.

**2:** This can be done using the *os module* with the *walk() function*.
**3:** This will take a specific path as input and generate a 3-tuple involving *dirpath*, *dirnames*, and *filenames*.
**4:** In the below example we are searching for a file named smpl.htm starting at the root directory named "D:".
**5:** The os.walk() function searches the entire directory and each of its subdirectories to locate this file.
**6:** As the result we see that the file is present in both the main directory and also in a subdirectory. We are running this program in a windows OS.

## Example

```
import os
def find_files(filename, search_path):
   result = []
# Wlaking top-down from the root
for root, dir, files in os.walk(search_path):
         if filename in files:
               result.append(os.path.join(root, filename))
return result

print(find_files("smpl.htm","D:"))
```

Output
```
['D:TP\smpl.htm', 'D:TP\spyder_pythons\smpl.htm']
```

---

# Letting the user choose the file name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using `input` as follows:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)
```

We read the file name from the user and place it in a variable named `fname` and open that file. Now we can run the program repeatedly on different files.

output:

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

---

# try and except

searching through a file while using `try` and `except` for error handling:

```
try:
    # Get the file name from the user
    file_path = input("Enter the file name: ")
    search_term = input("Enter the search term: ")
    found_lines = []

    # Open the file in read mode
    with open(file_path, 'r') as file:
        # Read each line in the file
        for line_number, line in enumerate(file, 1):
            # Check if the search term is in the line
            if search_term in line:
                found_lines.append((line_number, line))
```

```
    # Print the lines where the search term was found
    if found_lines:
        print(f"Search term '{search_term}' found in the following
lines:")
        for line_number, line in found_lines:
            print(f"Line {line_number}: {line.strip()}")
    else:
        print(f"Search term '{search_term}' not found in the file.")

except FileNotFoundError:
    print(f"File not found: {file_path}")
except Exception as e:
    print(f"An error occurred: {str(e)}")
```

In this code:

1. We use a `try` block to encompass the code that might raise exceptions.
2. Inside the `try` block, we use the `open()` function to open the file specified by the user's input.
3. If the file is not found (raises a `FileNotFoundError`), it's caught by the `except FileNotFoundError` block, and we print an error message.
4. If any other exception occurs (e.g., file reading errors), it's caught by the `except Exception as e` block, and we print an error message with details about the exception.

---

# Why use try and except?

**1:** Programs often encounter unexpected situations or errors while executing.

**2:** Using try and except blocks allows you to handle these errors gracefully, preventing your program from crashing and providing a better user experience.

**3:** Except blocks let you specify how to respond to different types of errors. **4:** By catching specific exceptions, you can identify the nature of the problem and take appropriate actions.

**5:** For example, you can distinguish between file not found errors and file access errors.
**6:** When an exception is caught, you can choose to display informative error messages to the user or log them for debugging purposes. This helps in diagnosing issues and aids in troubleshooting.

---

# Unit 5 ( Collections: Arrays, Lists, Dictionaries and Tuples )

## Arrays

**1:** In Python, arrays are commonly implemented using lists or the `array` module.
**2:** However, the term "array" is often used interchangeably with "list" in Python, as Python lists are quite flexible and can store elements of different data types.
**3:** Additionally, Python provides the `numpy` library for working with arrays in a more efficient and specialized manner.
Here's a brief overview of arrays/lists in Python and their basic syntax:

Lists(commonly used Arrays)

1. Definition: A list is an ordered collection of elements, and it can hold elements of different data types. Lists are mutable, meaning you can change their contents (add, remove, modify elements) after creation.
2. Syntax: Lists are created using square brackets `[]`, with elements separated by commas. For example:

```
my_list = [1, 2, 3, 4, 5]
```

---

## Basic operations on Array

The `array` module provides arrays with a fixed data type, which can be more memory-efficient than lists when dealing with large datasets of the same data type.

**1:** Importing the Module: You need to import the `array` module to use it:

```
from array import array
```
**2:** Creating an Array: You create an array using the `array()` constructor, specifying the data type code and the initial elements. For example:
```
int_array = array('i', [1, 2, 3, 4, 5])
```
**3:** Accessing Elements: Elements in an `array` are accessed similarly to lists using indexing.
```
first_element = int_array[0]
```
**4**: Slicing: You can slice an `array` just like a list.

```
sub_array = int_array[1:4]
```
**5:** Unlike lists, array objects have a fixed size, so you can't dynamically add or remove elements. You can only modify existing elements.

Note that for more advanced array operations, especially when working with numerical data, you might want to consider using the `numpy` library, which offers powerful array manipulation capabilities.

---

# Lists

**1:** A list in Python is a versatile data structure used to store an ordered collection of items.

**2:** These items can be of different data types, and they are enclosed in square brackets, separated by commas.

**3:** Lists are zero-indexed, which means the first item in the list has an index of 0, the second has an index of 1, and so on.

**4:** You can access elements in a list using their indices.

**5:** Lists are mutable, which means you can change their contents by adding, removing, or modifying elements.

**6:** This makes them suitable for dynamic data storage.

**7:** Python provides a wide range of built-in methods for manipulating lists.

**8:** You can use methods like `append()`, `insert()`, `remove()`, `pop()`, `sort()`, and `reverse()` to perform various operations on lists.

**9:** You can extract a portion of a list using slicing.

**10:** Slicing allows you to create a new list containing elements from a specified start to end index.

---

# Traversing a list

Traversing a list in Python means iterating or going through each element of the list, typically to perform some operation on each element. Here's how you can traverse a list using a for loop:

```
my_list = [1, 2, 3, 4, 5]

for item in my_list:
    # Do something with each element, for example, print it
    print(item)
```

In this example, the for loop iterates over each element in my_list, and the item variable takes on the value of each element in turn. You can replace the print(item) line with any operation you want to perform on each element.

---

# list operations

**1:** Creating a List: You can create a list by enclosing items in square brackets and separating them with commas.

```
my_list = [1, 2, 3, 4, 5]
```

**2:** Accessing Elements : You can access elements in a list by their index. Lists are zero-indexed, so the first element has an index of 0.

```
first_element = my_list[0]  # Access the first element
```

**3:** Modifying Elements: Lists are mutable, so you can change their contents by assigning new values to specific indices.

```
my_list[2] = 99 # Change the third element to 99
```

**4:** Adding Elements: You can add elements to a list using methods like `append()` and `insert()`.

```
my_list.append(6)
# Add an element to the end
```

```
   my_list.insert(1, 7)    # Insert 7 at index 1
```

**5:** Removing Elements: You can remove elements using methods like `remove()`, `pop()`, and `del`.

```
  my_list.remove(3) # Remove the first occurrence of 3
    popped_element = my_list.pop(2)  # Remove and return the element at
index 2
   del my_list[0]     # Delete the first element
```

**6:** List Concatenation: You can combine two or more lists using the `+` operator.

```
  combined_list = my_list + [8, 9, 10]
```

**7:** Slicing: You can extract a portion of a list using slicing.

```
 subset = my_list[1:4]
 # Get elements from index 1 to 3 (exclusive)
```

**8:** Sorting: You can sort a list using the `sort()` method (in-place) or the `sorted()` function (returns a new sorted list).

```
  my_list.sort() # Sort the list in ascending order
    sorted_list = sorted(my_list)  # Create a new sorted list
```

**9:** Finding Length: You can find the length of a list using the `len()` function.

```
 length = len(my_list)
# Get the length of the list
```

---

# List slices

you can create slices of lists using the slice notation. The basic syntax for creating a slice is:

```
start:end:step
```

Here's what each part means:

- `start`: The index at which the slice begins (inclusive).
- `end`: The index at which the slice ends (exclusive).
- `step`: The step or increment value for selecting elements.

Here are some examples of list slices:

**1:** Get a portion of a list from index 1 to 4 (excluding 4):

```
my_list = [0, 1, 2, 3, 4, 5]
slice_result = my_list[1:4]
# Result: [1, 2, 3]
```

**2:** Get every other element in a list:

```
my_list = [0, 1, 2, 3, 4, 5]
slice_result = my_list[::2]
# Result: [0, 2, 4]
```

**3:** Get a portion of a list from the beginning up to index 3:

```
my_list = [0, 1, 2, 3, 4, 5]
slice_result = my_list[:3]
# Result: [0, 1, 2]
```

**4:** Get a portion of a list from index 2 to the end:

```
my_list = [0, 1, 2, 3, 4, 5]
slice_result = my_list[2:]
# Result: [2, 3, 4, 5]
```

**5:** Reverse a list using a slice:

```
my_list = [0, 1, 2, 3, 4, 5]
slice_result = my_list[::-1]
# Result: [5, 4, 3, 2, 1, 0]
```

---

# List methods

**1:**`append(item)`: Adds an item to the end of the list.

```
my_list = [1, 2, 3]
my_list.append(4)
# Result: [1, 2, 3, 4]
```

**2:** `extend(iterable)`: Appends the elements of an iterable (e.g., another list) to the end of the list.

```
my_list = [1, 2, 3]
my_list.extend([4, 5, 6])
# Result: [1, 2, 3, 4, 5, 6]
```

**3:** `insert(index, item)`: Inserts an item at a specific index in the list.

```
my_list = [1, 2, 3]
my_list.insert(1, 4)
# Result: [1, 4, 2, 3]
```

**4:** `remove(item)`: Removes the first occurrence of the specified item from the list.

```
my_list = [1, 2, 3, 2]
my_list.remove(2)
# Result: [1, 3, 2]
```

**5:**`pop(index)`: Removes and returns the item at the specified index. If no index is provided, it removes and returns the last item.

```
my_list = [1, 2, 3]
popped_item = my_list.pop(1)
# Result: popped_item = 2, my_list = [1, 3]
```

**6:**`index(item)`: Returns the index of the first occurrence of the specified item in the list.

```
my_list = [1, 2, 3, 2]
index = my_list.index(2)
# Result: index = 1
```

**7:**`count(item)`: Returns the number of times a specific item appears in the list.

```
my_list = [1, 2, 3, 2]
```

```
count = my_list.count(2)
# Result: count = 2
```

**8:**`sort()`: Sorts the list in ascending order.

```
my_list = [3, 1, 2]
my_list.sort()
# Result: my_list = [1, 2, 3]
```

**9:**`reverse()`: Reverses the order of elements in the list.

```
my_list = [1, 2, 3]
my_list.reverse()
# Result: my_list = [3, 2, 1]
```

**10:** `clear()`: Removes all items from the list.

```
my_list = [1, 2, 3]
my_list.clear()
# Result: my_list = []
```

---

## Deleting elements in list

**1:** Using `del` statement: You can use the `del` statement followed by the list and the index of the element you want to delete.

```
my_list = [1, 2, 3, 4, 5]
del my_list[2]  # Deletes the element at index 2 (value 3)
# Result: my_list = [1, 2, 4, 5]
```

**2:** Using `remove()` method: You can use the `remove()` method to delete the first occurrence of a specific value in the list.

```
my_list = [1, 2, 3, 2, 4]
my_list.remove(2)  # Removes the first occurrence of 2
# Result: my_list = [1, 3, 2, 4]
```

**3:** Using slicing: You can use slicing to create a new list without the element(s) you want to delete.

```
my_list = [1, 2, 3, 4, 5]
my_list = my_list[:2] + my_list[3:]  # Deletes the element at
index 2 (value 3)
# Result: my_list = [1, 2, 4, 5]
```

**4:** Using `pop()` method: The `pop()` method can be used to remove and return an element at a specific index. If no index is provided, it removes and returns the last element.

```
my_list = [1, 2, 3, 4, 5]
removed_element = my_list.pop(2)  # Removes and returns the
element at index 2 (value 3)
# Result: removed_element = 3, my_list = [1, 2, 4, 5]
```

**5:** Using list comprehension: You can create a new list comprehension that excludes the elements you want to delete.

```
my_list = [1, 2, 3, 4, 5]
my_list = [x for x in my_list if x != 3]  # Deletes all
occurrences of 3
# Result: my_list = [1, 2, 4, 5]
```

---

# Lists and functions

In Python, you can use functions to work with lists in various ways. Functions allow you to encapsulate and reuse code for common list operations. Here are some examples of how you can use functions with lists:

**1:** Creating a Function to Modify a List: You can define a function that takes a list as an argument, performs some operation on the list, and returns the modified list. For example, a function that squares each element of a list:

```
def square_elements(my_list):
    squared_list = [x**2 for x in my_list]
    return squared_list

original_list = [1, 2, 3, 4]
result = square_elements(original_list)
# Result: result = [1, 4, 9, 16]
```

**2:** Passing Lists as Arguments: You can pass one or more lists as arguments to a function, allowing you to perform operations on multiple lists within the function.

```
def combine_lists(list1, list2):
    combined_list = list1 + list2
    return combined_list

list_a = [1, 2, 3]
list_b = [4, 5, 6]
result = combine_lists(list_a, list_b)
# Result: result = [1, 2, 3, 4, 5, 6]
```

**3:** Returning Lists from Functions: Functions can return lists as their output, allowing you to generate and return lists as needed.

```
def generate_even_numbers(n):
    even_numbers = [x for x in range(2, n + 1, 2)]
    return even_numbers

evens_up_to_10 = generate_even_numbers(10)
# Result: evens_up_to_10 = [2, 4, 6, 8, 10]
```

**4:** Modifying Lists in Place: Functions can modify lists in place by using methods like `append`, `extend`, `insert`, or `pop`. These changes are reflected outside the function as well.

```
def add_element(my_list, element):
    my_list.append(element)

original_list = [1, 2, 3]
add_element(original_list, 4)
# Result: original_list = [1, 2, 3, 4]
```

**5:** Using List Comprehensions in Functions: List comprehensions can be used within functions to create new lists based on some criteria.

```
def filter_positive_numbers(my_list):
    positive_numbers = [x for x in my_list if x > 0]
    return positive_numbers

numbers = [-1, 2, -3, 4, -5]
positive = filter_positive_numbers(numbers)
# Result: positive = [2, 4]
```

Functions provide a way to encapsulate logic and make your code more modular and reusable when working with lists in Python.

---

## parsing lines

Parsing lines in Python lists typically involves iterating through the list and processing each string (line) individually. You can use a for loop to go through each string in the list and apply parsing logic as needed. Here's a basic example:

Suppose you have a list of strings representing lines of text, and you want to split each line into words based on spaces:

```
lines = [
    "Hello, World!",
    "Python is awesome",
    "OpenAI is amazing",
    "Programming is fun",
]

# Initialize an empty list to store the parsed lines
parsed_lines = []

# Iterate through each line in the list
for line in lines:
    # Split the line into words based on spaces
    words = line.split()
    # Add the list of words to the parsed_lines list
    parsed_lines.append(words)

# Print the parsed lines
```

```
for words in parsed_lines:
    print(words)
```

In this example, the `split()` method is used to split each line into words based on spaces, and the resulting list of words is added to the `parsed_lines` list.

```
Output:

['Hello,', 'World!']
['Python', 'is', 'awesome']
['OpenAI', 'is', 'amazing']
['Programming', 'is', 'fun']
```

You can modify the parsing logic based on your specific requirements. For more complex parsing tasks, you may need to use regular expressions or other parsing libraries to extract information from the lines.

---

# Objects and values

## *Objects*

**1:** An object is a fundamental concept in Python. Everything in Python is an object, including numbers, strings, lists, functions, and custom-defined classes.

**2:** Each object has a unique identity, a type, and a value.

**3:** The identity of an object is a unique integer that distinguishes it from other objects. You can check the identity of an object using the `id()` function.

**4:** The type of an object defines its behavior and the operations that can be performed on it. You can check the type of an object using the `type()` function.

**5:** The value of an object is the data it holds. It can be mutable (changeable) or immutable (unchangeable), depending on its type.

# Values

1: A value is the actual data stored in an object. It can be a simple value like an integer or a complex value like a list or a dictionary.

2: Values can be of different types, such as integers, floats, strings, lists, dictionaries, tuples, etc.

3: Immutable values, once created, cannot be changed. For example, integers and strings are immutable. When you perform operations that seem to modify an immutable value, you are actually creating a new value.

4: Mutable values can be changed after creation. Lists and dictionaries are examples of mutable types. When you modify a mutable value, the identity of the object remains the same, but its content can change.

Here are some examples to illustrate these concepts:

```
# Immutable objects (integers)
a = 5
b = a  # b references the same object as a
print(id(a), id(b))  # Both have the same identity
b = 10  # Create a new object with the value 10
print(id(a), id(b))  # Now they have different identities

# Mutable objects (lists)
list1 = [1, 2, 3]
list2 = list1  # Both list1 and list2 reference the same object
print(id(list1), id(list2))  # They have the same identity
list2.append(4)  # Modifying the object (list1) also affects list2
print(list1)  # [1, 2, 3, 4], the content has changed, but the
identity is the same
print(id(list1), id(list2))  # They still have the same identity
```

---

# aliasing

Alias refers to the situation where two or more variables (or names) reference the same object in memory. In the context of lists in Python, aliasing occurs when multiple variables point to the same list object. This can lead to unexpected behavior if you're not careful.

**1:** Simple List Assignment:

When you assign one list to another variable, both variables reference the same list in memory:

```
list1 = [1, 2, 3]
list2 = list1  # Alias: list2 refers to the same list as list1
```

Now, if you modify `list1` or `list2`, the changes will be reflected in both variables because they both point to the same list:

**2:** Passing Lists to Functions:
Similar to simple assignment, when you pass a list as an argument to a function, any changes made to the list within the function will affect the original list (since the function parameter becomes an alias for the original list):

```
def modify_list(my_list):
    my_list.append(42)

original_list = [1, 2, 3]
modify_list(original_list)
print(original_list)  # Output: [1, 2, 3, 42]
```

**3:** Slicing Lists:
Slicing a list also creates a new list that is an alias to the original list:

```
list1 = [1, 2, 3, 4, 5]
list2 = list1[1:4]  # Alias: list2 is a new list containing elements
[2, 3, 4]
```

If you modify `list2`, it won't affect `list1`, but they initially share some common elements.

To avoid unintended aliasing, you can create a copy of a list using the `list()` constructor or the slicing technique mentioned below:

```
# Creating a copy of a list
list1 = [1, 2, 3]
list2 = list(list1)  # Now list2 is a separate copy, not an alias
```

---

# List arguments

In Python, you can pass lists as arguments to functions just like any other data type. When you pass a list to a function, you are actually passing a reference to the list, which means that any

changes made to the list within the function will affect the original list. Here's how you can work with list arguments in functions

```
# Example function that modifies a list
def modify_list(my_list):
    my_list.append(42)  # Append an element to the list

# Create a list
original_list = [1, 2, 3]

# Call the function with the list as an argument
modify_list(original_list)

# The original list is modified
print(original_list)  # Output: [1, 2, 3, 42]
```

In this example, `original_list` is passed as an argument to the `modify_list` function. Inside the function, we append an element to the list, and this modification is reflected in the original list outside the function.
Here are some important things to keep in mind when working with list arguments:

1. Mutability: Lists are mutable in Python, so any changes made to a list inside a function will affect the original list.
2. Passing by Reference: When you pass a list as an argument, you are passing a reference to the original list, not a copy of it. This means that the function operates on the same list object in memory.

---

# Dictionary

**1:** A dictionary in Python is a built-in data type.

**2:** It stores a collection of key-value pairs.

**3:** Dictionaries are enclosed within curly braces `{}`.

**4:** Use curly braces to create a dictionary.

**5:** Key-value pairs are separated by colons `:` and comma-separated.

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
```

**6:** Python provides built-in methods for dictionaries. Examples: `keys()`, `values()`, `items()`, `get(key)`, `update(other_dict)`, `pop(key)`, `popitem()`, `clear()`.

**7:** Dictionaries are used for tasks like counting occurrences, storing configuration settings, and more.
**8:** They offer efficient lookup and retrieval of values based on keys.
**9:** Dictionaries are a versatile data structure in Python, providing a way to store and manipulate data with key-value associations.

---

# dictionary as a set of counters

In Python, you can use a dictionary as a set of counters to count the occurrences of elements, values, or items in a collection. This is a common and useful use case for dictionaries. Here's how you can use a dictionary for counting:

**1:** Creating a Counter Dictionary: You start with an empty dictionary and increment the counts as you encounter elements:

```
# Create an empty counter dictionary
counter_dict = {}

# List of elements to count
elements = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]

# Count the occurrences of elements
for element in elements:
    if element in counter_dict:
        counter_dict[element] += 1
    else:
        counter_dict[element] = 1
```

**2:** Resulting Counter Dictionary: After counting, the `counter_dict` will contain the counts of each unique element:

```
print(counter_dict)
  # Output: {1: 1, 2: 2, 3: 3, 4: 4}
```

**3:** Using the `collections` Module: Python's `collections` module provides a convenient `Counter` class that simplifies the counting process:

```
from collections import Counter

elements = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
counter_dict = Counter(elements)
```

```
print(counter_dict)
# Output: Counter({4: 4, 3: 3, 2: 2, 1: 1})
```

**4:** Accessing Counts: You can access the counts of specific elements by using the key in the counter dictionary:

```
print(counter_dict[3])  # Output: 3
```

**5:** Common Uses:

Counting with dictionaries is handy for various tasks, including:

- - Analyzing data frequency (e.g., word frequency in text analysis).
- - Keeping track of occurrences in datasets.
- - Identifying the most common elements in a collection.

---

# Dictionary in files

Dictionaries can be used in Python to work with files for various purposes, such as reading and writing data, parsing structured data formats, and storing configurations. Here are some common scenarios of using dictionaries with files:

**1:** Reading Data from Files into Dictionaries:

You can read data from files, such as CSV or JSON files, and store it in dictionaries for easy access and manipulation. For example, reading data from a CSV file:

```
import csv

data_dict = {}
with open('data.csv', 'r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        data_dict[row['name']] = row['value']
```

**2:** Writing Data from Dictionaries to Files:
You can also write data from dictionaries to files in various formats, like JSON or CSV, for storage or data exchange. For instance, writing data to a JSON file:

```
import json
```

```
data_dict = {"name": "John", "age": 30, "city": "New York"}

with open('data.json', 'w') as file:
    json.dump(data_dict, file)
```

**3:** Parsing Structured Data Formats:
Dictionaries are often used to parse and manipulate structured data formats like JSON and XML. You can read such data into dictionaries and access specific values easily. Here's an example of parsing JSON data:

```
import json

with open('data.json', 'r') as file:
    data_dict = json.load(file)

print(data_dict["name"])  # Accessing a value
```

**4:** Storing Configuration Settings:
Dictionaries are suitable for storing configuration settings, which can be loaded from a file. This allows you to change program behavior without modifying code. For example:

```
import json

# Load configuration settings from a JSON file
with open('config.json', 'r') as file:
    config_dict = json.load(file)

# Accessing configuration settings
print(config_dict["api_key"])
```

**5:** Working with Log Files:
Dictionaries can be used to process log files, allowing you to analyze and extract information from the logs. You can use the log data to populate dictionaries and perform various analyses.

**6:** Storing Data for Later Use:
You can save intermediate data or results in dictionaries and later write them to a file for further processing or to preserve the state of your program.

---

# looping with dictionaries

Looping through dictionaries in Python allows you to iterate over key-value pairs, keys, or values within the dictionary. Python provides various ways to achieve this, depending on your specific needs. Here are some common techniques for looping through dictionaries:

**1:** Looping Through Keys:

You can use a `for` loop to iterate through the keys of a dictionary:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}

   for key in my_dict:
       print(key)
```

This loop will print the keys `"name"`, `"age"`, and `"city"`.

**2:** Looping Through Values:
You can iterate through the values of a dictionary using the `.values()` method:

```
 my_dict = {"name": "John", "age": 30, "city": "New York"}

   for value in my_dict.values():
       print(value)
```

This loop will print the values `"John"`, `30`, and `"New York"`.

**3:** Looping Through Key-Value Pairs:
You can iterate through both keys and values as key-value pairs using the `.items()` method:

```
   my_dict = {"name": "John", "age": 30, "city": "New York"}

   for key, value in my_dict.items():
       print(key, value)
```

This loop will print:

```
 name John
   age 30
   city New York
```

**4:** Iterating with a Conditional:
You can use loops in conjunction with conditionals to filter and process specific key-value pairs:

```
 my_dict = {"name": "John", "age": 30, "city": "New York"}

   for key, value in my_dict.items():
       if key == "age" and value > 25:
           print(f"{key}: {value}")
```

This loop will print `"age: 30"` because it checks the condition that the key is `"age"` and the value is greater than `25`.

**5:** Looping with Dictionary Comprehension:

You can also use dictionary comprehension to create a new dictionary based on the original one:

```
  my_dict = {"name": "John", "age": 30, "city": "New York"}

   filtered_dict = {key: value for key, value in my_dict.items() if
key != "age"}
```

This comprehension creates a new dictionary `filtered_dict` that excludes the key-value pair where the key is `"age"`.

---

# Advanced text parsing

Advanced text parsing involves the extraction and manipulation of structured or unstructured text data using various techniques and libraries in Python. This can be useful for tasks like natural language processing (NLP), data analysis, information retrieval, and more. Here are some advanced text parsing techniques and libraries in Python:

1: Regular Expressions:

- Regular expressions (regex) provide a powerful way to search for and manipulate text patterns.
- The `re` module in Python allows you to work with regular expressions.
- Example: Extracting email addresses or phone numbers from text.

```
   import re

   text = "My email is example@email.com and my phone number is 123-
456-7890."
   emails = re.findall(r'\S+@\S+', text)
   phone_numbers = re.findall(r'\d{3}-\d{3}-\d{4}', text)
```

**2:** NLTK (Natural Language Toolkit):

- NLTK is a library for working with human language data.
- It provides tools for tokenization, stemming, lemmatization, part-of-speech tagging, and more.
- Example: Tokenizing text into words.

```
 import nltk
   from nltk.tokenize import word_tokenize
```

```
   nltk.download('punkt')

   text = "This is an example sentence."
words = word_tokenize(text)
```

**3:** SpaCy:

- SpaCy is a NLP library that offers fast and efficient text processing.
- It provides capabilities for tokenization, named entity recognition, dependency parsing, and more.
- Example: Extracting named entities from text.

```
import spacy

   nlp = spacy.load("en_core_web_sm")
   text = "Apple Inc. is headquartered in Cupertino, California."
   doc = nlp(text)

   for entity in doc.ents:
print(entity.text, entity.label_)
```

**4:** Beautiful Soup:

- Beautiful Soup is a library for web scraping and parsing HTML and XML documents.
- It allows you to extract specific data from web pages.
- Example: Scraping and parsing data from a webpage.

```
 from bs4 import BeautifulSoup
   import requests

   url = "https://example.com"
   response = requests.get(url)
   soup = BeautifulSoup(response.text, 'html.parser')
```

**5:** TextBlob:

- TextBlob is a simple NLP library that provides a consistent API for diving into common natural language processing tasks.
- It includes functions for sentiment analysis, translation, and more.
- Example: Sentiment analysis of a text.

```
   from textblob import TextBlob

   text = "I love this product! It's amazing."
   sentiment = TextBlob(text).sentiment
```

---

# Tuples

a tuple is an ordered, immutable collection of elements. Tuples are similar to lists, but unlike lists, tuples cannot be modified after they are created. This means you can't add, remove, or change elements in a tuple once it's defined. Tuples are commonly used when you want to group together multiple values, and you want to ensure that the data remains unchanged throughout the program.

Here's how you can create a tuple in Python:

```
my_tuple = (1, 2, 3)
```
Tuples can also be created without parentheses, although it's a good practice to include them for clarity:
```
my_tuple = 1, 2, 3
```
You can access elements in a tuple using indexing, just like you would with a list:
```
print(my_tuple[0])  # Output: 1
```
Tuples are often used for functions that return multiple values:
```
def get_coordinates():
    return 2, 3

x, y = get_coordinates()
print(x)  # Output: 2
print(y)  # Output: 3
```

---

# comparing tuples

In Python, you can compare tuples using comparison operators like `==` (equal), `!=` (not equal), `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal). Tuple comparison is performed element-wise, and the comparison stops as soon as a definitive result can be determined.

Here's how tuple comparisons work:

**1:** Equal (`==`) and Not Equal (`!=`): Tuples are compared element by element, starting from the first element. If the elements at the same positions in both tuples are equal, the comparison proceeds to the next elements. If any pair of elements is found to be not equal, the comparison returns `False`. If all pairs of elements are equal, the comparison returns `True`.

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 3)

print(tuple1 == tuple2)  # Output: True
```

**2:** Less Than (`<`) and Greater Than (`>`): Tuples are compared element by element, starting from the first element. If the elements at the same positions in both tuples are equal, the comparison proceeds to the next elements. If any pair of elements is found to be not equal, the comparison returns the result of comparing those elements. If all pairs of elements are equal, the shorter tuple is considered less than the longer tuple.

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 4)

print(tuple1 < tuple2)  # Output: True (because 3 < 4)
```

**3:** Less Than or Equal To (`<=`) and Greater Than or Equal To (`>=`): These operators work similarly to `<` and `>`, but they return `True` if the tuples are equal as well.

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 3)

print(tuple1 <= tuple2)  # Output: True
```

It's important to note that when comparing tuples, the comparison is done element-wise, so if the first elements are equal, the second elements are compared, and so on. If any pair of elements differs, the comparison result is determined based on that pair. If all elements are equal, the lengths of the tuples are considered.

Keep in mind that the elements within the tuples must be comparable for these comparison operators to work without raising exceptions. If you attempt to compare tuples with elements that cannot be compared (e.g., tuples with nested lists), you may encounter errors.

---

# Tuple Assignment

Tuple assignment in Python refers to the process of simultaneously assigning values to multiple variables using a tuple on the right-hand side of the assignment. This allows you to assign values

from a tuple to individual variables in a single statement. It's a powerful and concise way to swap values between variables, return multiple values from a function, or assign values from iterable objects like lists or tuples to variables.

Here's a basic example of tuple assignment:

```
# Assigning values from a tuple to variables
my_tuple = (1, 2, 3)
a, b, c = my_tuple

print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

**1:** Tuple assignment is particularly useful when you want to swap the values of two variables without using a temporary variable:

```
x = 5
y = 10

# Swap the values of x and y using tuple assignment
x, y = y, x

print("x =", x)  # Output: 10
print("y =", y)  # Output: 5
```

**2:** You can also use tuple assignment with functions that return tuples to capture multiple values returned by the function:

```
def get_coordinates():
    return 2, 3

x, y = get_coordinates()

print("x =", x)  # Output: 2
print("y =", y)  # Output: 3
```

**3:** Tuple assignment can be used with any iterable object, not just tuples. For example, you can use it with lists, strings, or any other iterable:

```
my_list = [10, 20, 30]
first, second, third = my_list

print(first)   # Output: 10
print(second)  # Output: 20
print(third)   # Output: 30
```

```
my_string = "Hello"
a, b, c, d, e = my_string

print(a)  # Output: 'H'
print(b)  # Output: 'e'
print(c)  # Output: 'l'
print(d)  # Output: 'l'
print(e)  # Output: 'o'
```
Tuple assignment is a versatile feature in Python that allows you to work with multiple values efficiently in a single statement.

---

# Dictionaries and Tuples

Dictionaries and tuples are two distinct data structures in Python, each with its own purpose and characteristics. However, they can be used together in various ways to solve specific programming problems or to represent more complex data structures.

Here are some common ways dictionaries and tuples can be used together:

**1:** Using Tuples as Dictionary Keys:

- Tuples are hashable, which means they can be used as dictionary keys, unlike lists which are mutable and not hashable. This can be useful when you want to create dictionaries with composite keys.

```
person = {("John", "Doe"): 30, ("Jane", "Smith"): 28}

# Accessing values using tuple keys
print(person[("John", "Doe")])  # Output: 30
```

**2:** Storing Key-Value Pairs in Tuples:

- You can store key-value pairs as tuples within a list or another tuple. This can be useful when you want to maintain an ordered collection of key-value pairs.

```
data = [("name", "John"), ("age", 30), ("city", "New York")]

# Creating a dictionary from a list of key-value pairs
person = dict(data)

# Accessing values using keys
```

```
print(person["name"])  # Output: "John"
```

**3:** Using Tuples to Iterate Through Dictionary Items:

- When iterating through the items of a dictionary using a `for` loop, you can use tuple unpacking to access both the key and the value in each iteration.

```
person = {"name": "John", "age": 30, "city": "New York"}

for key, value in person.items():
    print(key, value)

# Output:
# name John
# age 30
# city New York
```

**4:** Tuples as Immutable Keys for a Dictionary:

- You can use tuples as keys when you need to create a dictionary where the key should remain unchanged. Since tuples are immutable, they can serve as keys even if their contents are mutable objects.

```
data = {("John", "Doe"): [30, "New York"], ("Jane", "Smith"): [28, "Los Angeles"]}

# Accessing values using tuple keys
print(data[("John", "Doe")])  # Output: [30, "New York"]
```

**5:** Using Tuples to Sort Dictionary Items:

- Tuples can be used to sort dictionary items based on keys or values. You can convert the items of the dictionary into a list of tuples, sort them, and create a new dictionary if needed.

```
person = {"name": "John", "age": 30, "city": "New York"}

# Sorting dictionary items by keys
sorted_items = sorted(person.items())

# Creating a new dictionary from the sorted items
sorted_person = dict(sorted_items)

print(sorted_person)
# Output: {'age': 30, 'city': 'New York', 'name': 'John'}
```

In summary, dictionaries and tuples can complement each other in various Python programming scenarios, allowing you to work with structured data, composite keys, and maintain the order of key-value pairs when necessary.

---

# Multiple assignment with dictionaries

In Python, you can use multiple assignment with dictionaries to simultaneously assign values from a dictionary to multiple variables. This is a convenient way to extract values from a dictionary when you know the keys you want to access. Here's how multiple assignment with dictionaries works:

Suppose you have a dictionary like this:

```
person = {"name": "John", "age": 30, "city": "New York"}
```

You can use multiple assignments to extract values from this dictionary:

```
name, age, city = person["name"], person["age"], person["city"]

print(name)  # Output: "John"
print(age)   # Output: 30
print(city)  # Output: "New York"
```

However, a more concise and readable way to achieve the same result is by using dictionary unpacking with the `**` operator. Here's how you can do it:

```
person = {"name": "John", "age": 30, "city": "New York"}

# Using dictionary unpacking to extract values
name, age, city = person.values()

print(name)  # Output: "John"
print(age)   # Output: 30
print(city)  # Output: "New York"
```

This approach is particularly useful when you have a dictionary with many key-value pairs and you want to extract multiple values at once.
If you only need some of the values and not all of them, you can use a combination of dictionary unpacking and key access. For example:

```
person = {"name": "John", "age": 30, "city": "New York"}

# Extracting 'name' and 'city' values
name, city = person["name"], person["city"]

print(name)  # Output: "John"
print(city)  # Output: "New York"
```

Note that when using dictionary unpacking, the order of values you receive is not guaranteed to be in any specific order, as dictionaries are unordered collections. If you need to maintain order, consider using an ordered dictionary (`collections.OrderedDict`) or another data structure.

---

# Tuples in dictionaries

In Python, you can use tuples as keys in dictionaries. This is possible because tuples are hashable and immutable, which makes them suitable for use as dictionary keys. Using tuples as keys can be useful when you need to create dictionaries with composite keys or when you want to represent structured data as keys.

Here's how you can use tuples as keys in dictionaries:

```
# Creating a dictionary with tuples as keys
student_scores = {("Alice", "Smith"): 95, ("Bob", "Johnson"): 88,
("Charlie", "Brown"): 92}

# Accessing values using tuple keys
print(student_scores[("Alice", "Smith")])  # Output: 95
print(student_scores[("Bob", "Johnson")])  # Output: 88
```

You can also use tuple unpacking when iterating through dictionary items with tuple keys:

```
# Iterating through the dictionary items
for (first_name, last_name), score in student_scores.items():
    print(f"{first_name} {last_name}: {score}")

# Output:
# Alice Smith: 95
# Bob Johnson: 88
# Charlie Brown: 92
```

When using tuples as keys, it's essential to remember that the order of elements within the tuple matters. For example, `("Alice", "Smith")` and `("Smith", "Alice")` would be considered different keys in a dictionary.

---

# Differences between strings, tuples and dictionaries

1. **mutability:**
    - Strings: Strings are immutable, which means you cannot change the characters of a string after it's created. You can create a new string by concatenating or slicing existing strings.

    - Lists: Lists are mutable, allowing you to add, remove, or modify elements within a list after its creation.
    - Tuples: Tuples are immutable like strings; once created, you cannot change the elements inside a tuple.
2. **Syntax:**
    - Strings: Enclosed in single (`'`), double (`"`), or triple (`'''` or `"""`) quotes.
    - Lists: Enclosed in square brackets (`[]`).
    - Tuples: Enclosed in parentheses (`()`) but can also be defined without parentheses.
3. **Typical Use Cases:**
    - Strings: Used for representing and manipulating text data.
    - Lists: Used for storing collections of items when you need to modify or rearrange the elements.
    - Tuples: Used when you want an ordered collection of elements that should not change during the program's execution.
4. **Performance:**
    - Strings: Because they are immutable, creating a new string by concatenating or slicing can lead to inefficient memory usage if done repeatedly. In such cases, consider using a list and converting it to a string when needed.
    - Lists: Lists are generally efficient for adding or removing elements at the end or middle of the list. However, modifying elements deep inside a large list can be less efficient.
    - Tuples: Tuples are efficient for read-only operations since they are immutable.

---

# Similarities

1. **Ordered Collections:** All three data types are ordered collections, meaning they maintain the order of elements as they were inserted. You can access individual elements or slices of elements based on their position within the sequence.
2. **Indexing and Slicing:** You can access individual elements or slices of elements using square brackets `[]` and indices. For example, `my_string[0]`, `my_list[2]`, and `my_tuple[1:3]` are all valid operations.
3. **Iterability:** You can iterate through the elements of strings, lists, and tuples using loops such as `for` loops, making it easy to process each element in the sequence.
4. **Membership Testing:** You can use the `in` operator to check if a specific element exists within a string, list, or tuple. For instance, `'a' in my_string`, `2 in my_list`, and `"apple" in my_tuple` are valid tests.
5. **Length:** You can determine the length (the number of elements) of a string, list, or tuple using the `len()` function. For example, `len(my_string)`, `len(my_list)`, and `len(my_tuple)` will return the respective lengths.

---

# Unit 6 ( Regular Expression and Object Oriented Programming )

## character matching in regular expression

Character matching in regular expressions involves specifying a particular character or set of characters that you want to match within a text string. Regular expressions (regex or regexp) provide a powerful way to describe patterns of characters. Here are some common ways to perform character matching in regular expressions:

**1. Literal Characters:** You can match literal characters by simply including them in the regex pattern. For example, the regex pattern "cat" will match the string "cat" in the text.

**2. Character Classes:** Character classes allow you to specify a set of characters that you want to match. For example:
- `[abc]` will match any of the characters 'a', 'b', or 'c'.
- `[0-9]` will match any digit from 0 to 9.
- `[^abc]` will match any character except 'a', 'b', or 'c'.

**3. Dot (.) Metacharacter:** The dot metacharacter `.` matches any single character except a newline. For example, the regex pattern "c.t" would match "cat," "cut," "cot," etc.

**4. Escape Special Characters:** Some characters have special meanings in regular expressions (e.g., `.` matches any character, `*` matches zero or more of the preceding element). To match these characters literally, you can escape them with a backslash, like `\.` or `\*`.

**5. Character Quantifiers:** You can specify how many times a character should appear using quantifiers:
- `a*` matches zero or more 'a's.
- `a+` matches one or more 'a's.
- `a?` matches zero or one 'a'.
- `a{2}` matches exactly two 'a's.
- `a{2,4}` matches between two and four 'a's.

**6. Anchors:** Anchors are used to match characters at the beginning and end of a line:
- `^` matches the start of a line.
- `$` matches the end of a line.

**7. Word Boundaries:** `\b` is used to match word boundaries, allowing you to match whole words. For example, `\bword\b` would match the word "word" but not "words" or "sword."

**8. Character Escape Sequences:** Some common character escape sequences include:
- `\d` matches any digit (equivalent to `[0-9]`).
- `\w` matches a word character (alphanumeric or underscore).
- `\s` matches whitespace characters (spaces, tabs, line breaks).

- `\D`, `\W`, and `\S` are the negations of `\d`, `\w`, and `\s`, respectively.

**9. Grouping:** You can use parentheses to group characters and apply operators to the entire group. For example, `(abc)+` will match one or more repetitions of the sequence "abc."

**10. Alternation (|):** The vertical bar `|` acts as an OR operator, allowing you to specify alternatives. For example, `cat|dog` will match either "cat" or "dog."

```
import re
# Example 1: Matching a specific word
text = "The quick brown fox jumps over the lazy dog"
pattern = r"fox"
matches = re.findall(pattern, text)
print(matches) # Output: ['fox']

# Example 2: Matching digits using character classes
text = "123 456 789"
pattern = r"\d+" # \d matches digits, + matches one or more
matches = re.findall(pattern, text)
print(matches) # Output: ['123', '456', '789']
```

---

# extracting data using regular expressions

Extracting data from a text using regular expressions in Python involves defining a pattern that matches the specific information you want to retrieve from the text. You can use the `re` module to perform data extraction. Here's an example of how to extract data using regular expressions in Python:

Let's say you have a text containing email addresses, and you want to extract all the email addresses from it:

```
import re

text = "Please contact support@example.com or info@domain.co for
assistance. For more info, email contact@mywebsite.org."

# Define a regular expression pattern to match email addresses
pattern = r"\b\w+@\w+\.\w+\b"
```

```
# Use re.findall to extract all email addresses that match the pattern
matches = re.findall(pattern, text)

# Print the extracted email addresses
for match in matches:
    print(match)
```

In this example, the regular expression pattern `\b\w+@\w+\.\w+\b` is used to match email addresses. Here's what the pattern does:

- `\b` matches word boundaries to ensure we extract complete email addresses.
- `\w+` matches one or more word characters (alphanumeric or underscore) before the "@" symbol.
- `@` matches the "@" symbol literally.
- `\w+` matches one or more word characters after the "@" symbol.
- `\.` matches the dot (.) symbol literally, which separates the domain and the top-level domain (TLD).
- `\w+` matches one or more word characters for the TLD.
- `\b` matches the end of the email address to ensure it's a complete word.

The `re.findall()` function is used to find all non-overlapping matches of this pattern in the input text. The extracted email addresses are then printed.

---

# Combining searching and extracting

Combining searching and extracting using regular expressions in Python typically involves using the `*re.search*()` function to locate a pattern within the text and then using the `re.findall()` function to extract specific data from the found match. This allows you to find a relevant portion of the text and then extract information from it. Here's an example of how to do this:

Let's say you have a text containing information about products, and you want to find and extract the product names and their corresponding prices:

```
import re

text = "Product: Phone, Price: $599.99, Product: Laptop, Price:
$999.00, Product: Headphones, Price: $49.99"

# Define a regular expression pattern to match product names
product_pattern = r"Product: (\w+)"
# Define a pattern to match prices
price_pattern = r"Price: (\$\d+\.\d{2})"
```

```
# Use re.search to find the first product in the text
match = re.search(product_pattern, text)

if match:
    product_name = match.group(1)  # Extract the product name
    print("Product:", product_name)

    # Now, use re.findall to extract all prices from the text
    prices = re.findall(price_pattern, text)

    # Print the extracted prices
    for price in prices:
        print("Price:", price)
```

In this example:

1. We first use `re.search()` to find the first product in the text using the `product_pattern`. If a match is found, we extract the product name using `match.group(1)`.

2. After finding the first product, we use `re.findall()` with the `price_pattern` to extract all prices from the text. This pattern matches the dollar amount in the format "$X.XX".

3. We then print both the product name and all the extracted prices.

You can modify the regular expression patterns and the data extraction process to match and extract different types of information from your text data, depending on your specific requirements.

---

# escape character

In regular expressions, an escape character is a character that changes the meaning of the character that follows it. It is often denoted by a backslash (`\`). The backslash is used to escape special characters in a regular expression, allowing you to match the literal character instead of its special meaning.

Here are some common examples of using the escape character in regular expressions:

**1. Matching a Literal Period (Dot):** The period (dot) is a special character in regular expressions that matches any character. To match a literal period, you need to escape it with a backslash: `\.`.

Example:

- `apple\.com` matches the text "*apple.com*" and not "appleXcom."

**2. Matching a Literal Asterisk (*):** The asterisk is a special character that represents "zero or more" of the preceding element. To match a literal asterisk, you need to escape it: `\*`.

Example:
- `3\*4` matches the text "3*4" and not "34" or "3334."

**3. Matching a Literal Backslash (\):** Since the backslash itself is used as the escape character, to match a literal backslash, you need to escape it with another backslash: `\\`.

Example:
- `C:\\Program Files` matches the text "C:\Program Files" with a literal backslash.

**4. Escaping Parentheses:** Parentheses have special meanings in regular expressions, such as defining capturing groups. To match literal parentheses, you should escape them.

Example:
- `\(abc\)` matches the text "(abc)".

**5. Escaping Square Brackets:** Square brackets are used to define character classes. To match literal square brackets, you need to escape them.

Example:
- `\[A-Z\]` matches the text "[A-Z]".

**6. Escaping Special Sequences:** Some escape sequences have special meanings, like `\n` for a newline character. If you want to match a literal escape sequence, you should escape the backslash as well.

Example:
- `This is a newline: \\n` matches the text "This is a newline: \n".

Using the escape character is essential when you want to match specific characters literally and not interpret their special regex meanings. It allows you to work with characters that would otherwise have a different interpretation in regular expressions.

---

# Managing larger programs

1. Modularity and Functions:
    - Break your code into smaller, self-contained functions or methods. Each function should have a specific purpose.
    - Use modules and packages to organize related functions and classes. This helps in grouping related functionality together.
2. Object-Oriented Programming (OOP):
    - Use classes and objects to encapsulate data and behavior, promoting code reusability and organization.
    - Leverage inheritance and polymorphism for more complex projects.
3. Code Structure:
    - Follow a clear and consistent project structure. Common structures include using directories for modules, scripts, data, and documentation.
    - Use a main script to coordinate the execution of your program.
    - Create a README file with instructions on how to use your program.
4. Docstrings and Comments:
    - Document your code using docstrings (e.g., using triple-quoted strings) to provide information about functions, classes, and modules.
    - Use comments for explaining complex or non-obvious code sections.
5. Version Control:
    - Use a version control system (e.g., Git) to track changes in your codebase and collaborate with others.
    - Host your code on platforms like GitHub or GitLab.
6. Testing:
    - Write unit tests for your functions and classes to ensure they work as expected.
    - Use testing frameworks like unittest or pytest to automate the testing process.
7. Error Handling:
    - Implement proper error handling to make your code more robust.
    - Use try-except blocks to handle exceptions gracefully.

---

# objects

In Python, objects are fundamental entities that represent data and behavior. Everything in Python is an object, including numbers, strings, functions, and classes.

Objects are instances of classes, which are blueprints or templates that define the structure and behavior of objects. Here's how to work with objects in Python:

Creating Objects:

- To create an object, you need to instantiate a class. You do this by calling the class as if it were a function. For example, if you have a class `Person`, you can create an instance of a `Person` object like this:

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

person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
```

Accessing Object Attributes:

- Objects have attributes that you can access using dot notation. In the example above, you can access the `name` and `age` attributes of `person1` and `person2` like this:

```
print(person1.name)  # Output: "Alice"
print(person2.age)   # Output: 25
```

Methods:

- Classes can also have methods, which are functions associated with the class. These methods can operate on the object's attributes. For example:

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

    def greet(self):
        return f"Hello, my name is {self.name}."

person1 = Person("Alice", 30)
print(person1.greet())  # Output: "Hello, my name is Alice."
```

---

# subdividing a program

Subdividing a program in Python is essential for maintaining code readability, organization, and reusability. Breaking a large program into smaller, manageable parts (subdividing) is often achieved by using functions, classes, and modules. Here's how you can do it:

1. Functions:
    - Break your program into smaller functions, each responsible for a specific task or functionality.
    - Keep functions focused on a single responsibility (Single Responsibility Principle).
    - Use function names that clearly indicate their purpose.

```
def data_processing(data):
    # Process and manipulate data
    return result

def user_interface():
    # Handle user input and interaction
    return choice

def main():
    data = get_data_from_source()
    result = data_processing(data)
    choice = user_interface()
```

Classes:

- Use classes to group related data and methods into objects, promoting encapsulation and modularity.
- Organize classes based on their logical relationships within your program.

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

    def study(self):
        # Define student's study behavior

class Teacher:
    def __init__(self, name, subject):
        self.name = name
        self.subject = subject

    def teach(self):
```

```
        # Define teacher's teaching behavior

def main():
    student1 = Student("Alice", 20)
    teacher1 = Teacher("Mr. Smith", "Math")
```

Libraries and Third-Party Packages:

- Leverage Python's extensive library and package ecosystem to avoid reinventing the wheel.
- Import and use external libraries or packages to perform common tasks and extend the functionality of your program.

```
import math  # Built-in library for mathematical functions
import requests  # External library for making HTTP requests

def main():
    value = math.sqrt(25)
    response = requests.get("
https://example.com/
")
```

Documentation:

- Add comments, docstrings, and documentation to clarify the purpose and usage of functions, classes, and modules.
- A well-documented program is easier to understand and maintain.

---

# Classes as types

In Python, classes can be used to define custom types. These custom types are often referred to as "user-defined types" or "class types." Here's how classes serve as types in Python:

1. Defining Custom Types:

When you create a new class, you are effectively defining a new custom type. This custom type can encapsulate both data (attributes) and behavior (methods). For example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
```

```
            self.y = y

        def move(self, dx, dy):
            self.x += dx
            self.y += dy
```

In this example, we've defined a custom type `Point` that represents 2D points, with `x` and `y` as its attributes and `move` as its behavior.

2. Creating Instances:

You can create instances (objects) of a class by instantiating the class. These instances are of the type you defined. For example:

```
point1 = Point(3, 4)
# point1 is an instance of the Point class
point2 = Point(1, 2)
 # point2 is another instance of the Point class
```

`point1` and `point2` are both of the type `Point`.

3. Data Abstraction:

Classes allow you to abstract and encapsulate data. In the `Point` class, the attributes `x` and `y` are encapsulated within the object, providing data abstraction.

```
print(point1.x)
# Accessing the 'x' attribute of point1
```

4. Polymorphism:

Custom types defined by classes can exhibit polymorphism, allowing objects of different classes to be treated as objects of the same base class. This can be useful for writing more generic and reusable code.

```
 class Shape:
        def area(self):
            pass

    class Circle(Shape):
        def __init__(self, radius):
            self.radius = radius

        def area(self):
            return 3.14159 * self.radius ** 2

    class Rectangle(Shape):
```

```
        def __init__(self, width, height):
            self.width = width
            self.height = height

        def area(self):
            return self.width * self.height
```

Here, `Circle` and `Rectangle` are both subclasses of `Shape`, and they provide their own implementations of the `area` method. This enables you to treat instances of different shape types as if they are of the same type (`Shape`) when calculating areas.

5. Using Custom Types:
Custom types defined by classes can be used as any other built-in types in Python. You can pass them as arguments, return them from functions, and use them in various data structures and operations.

```
def calculate_area(shape):
        return shape.area()

   shapes = [Circle(2), Rectangle(3, 4)]
   for shape in shapes:
       print(f"Area: {calculate_area(shape)}")
```

In summary, classes in Python can be used to define custom types with their own attributes and behaviors. These custom types can be instantiated into objects that are of the class type, and they can exhibit data abstraction, polymorphism, and other object-oriented programming principles. This is a powerful feature of Python that allows for the creation of structured and reusable code.

---

# object lifecycle

The object lifecycle in Python refers to the various stages an object goes through during its existence, from creation to destruction. Understanding the object lifecycle is crucial for managing resources, memory, and ensuring proper cleanup. Here are the key stages in the lifecycle of an object:

1. Creation:

- Objects are created using constructors or `__init__` methods when an instance of a class is instantiated.

- Memory is allocated for the object, and the object's attributes are initialized.

```
class MyClass:
       def __init__(self, data):
           self.data = data

   obj = MyClass("Hello")
```

2. Usage:
- During the usage stage, the object is actively utilized, and its methods and attributes are accessed or modified as needed.

```
  value = obj.data
   obj.data = "World"
```

3. Reference Count:
- Python uses a reference count mechanism to keep track of how many references (variables, other objects, etc.) are pointing to an object. When the reference count drops to zero, it means that the object is no longer accessible and can be destroyed.
- The reference count is automatically managed by Python's garbage collector.

```
  obj2 = obj  # Reference count increases
   del obj2    # Reference count decreases
```

4. Destruction (Garbage Collection):
- When an object's reference count reaches zero, the object is no longer accessible, and Python's garbage collector will mark it for destruction.
- The actual memory occupied by the object is released, and the `__del__` method (if defined) is called before the object is destroyed.

```
class MyClass:
       def __init__(self, data):
           self.data = data

       def __del__(self):
           print(f"Object with data '{self.data}' is being destroyed")

   obj = MyClass("Hello")
   del obj  # Object will be destroyed after this line
```

5. Finalization (Optional):

- If you want to perform any cleanup operations before an object is destroyed, you can define a `__del__` method in the class. The `__del__` method will be called before the object is destroyed.

```
class ResourceHandler:
       def __init__(self, resource):
           self.resource = resource

       def __del__(self):
           self.release_resource()

       def release_resource(self):
           # Perform resource cleanup
           print(f"Releasing resource: {self.resource}")

   obj = ResourceHandler("File")
   # When obj is deleted or goes out of scope, __del__ is called
```

It's important to note that you typically don't need to worry about memory management and object destruction in Python, as Python's garbage collector takes care of most of these tasks. However, understanding the object lifecycle and when to define `__del__` methods can be useful when dealing with resources that require manual cleanup, such as file handles or database connections. In most cases, it's better to use context managers (`with` statements) or other resource management techniques to ensure proper cleanup.

---

# multiple instances

**1:** In Python, you can create multiple instances of a class by repeatedly calling the class constructor.

**2:** Each instance is a separate object, and they can have their own unique attributes and states.

**3:** Each instance of a class is independent, and changes made to one instance do not affect the others.

**4:** This allows you to model and work with multiple objects of the same type, each with its own unique data and behavior.

**5:** Creating and using multiple instances is a fundamental concept in object-oriented programming, allowing you to work with different objects that share a common structure defined by a class.

**6:** This is useful for representing and managing data in more complex and dynamic scenarios, such as managing a list of employees or storing information about various products in a catalog.

Here's how you can create and work with multiple instances of a class:

**1:** Define a Class: First, define a class that specifies the blueprint for the objects you want to create. In this example, we'll define a simple `Person` class with a name and age attribute.

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

**2:** Create Instances:To create multiple instances of the `Person` class, you can call the constructor (`__init__` method) with different arguments for each instance.

```
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
person3 = Person("Charlie", 40)
```

**3:** Access Attributes: You can access the attributes of each instance to retrieve information specific to that object.

```
print(person1.name)  # Output: "Alice"
print(person2.age)   # Output: 25
print(person3.name)  # Output: "Charlie"
```

**4:** Methods and Behavior: Instances can also have methods defined in the class. These methods can operate on the instance's attributes.

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

    def greet(self):
        return f"Hello, my name is {self.name}."

person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

print(person1.greet())  # Output: "Hello, my name is Alice."
print(person2.greet())  # Output: "Hello, my name is Bob."
```

---

# Inheritance

**1:** In Python, you can use inheritance to create new classes (subclasses or child classes) that inherit attributes and methods from existing classes (superclasses or parent classes).

**2:** It represents real-world relationships well.

**3:** It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.

**4:** It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A. **5:** Less development and maintenance expenses result from an inheritance.

---

# Types of inheritance

1. Single Inheritance:

- Single inheritance involves creating a new class (subclass) that inherits attributes and methods from a single existing class (superclass).

- This is the most common type of inheritance in Python.

```
class Animal:
      def speak(self):
          pass

  class Dog(Animal):
      def speak(self):
          return "Woof!"

  class Cat(Animal):
      def speak(self):
          return "Meow!"
```

2. Multiple Inheritance:

- Multiple inheritance allows a subclass to inherit attributes and methods from multiple superclasses.
- In case of method name conflicts, the method from the leftmost superclass is used.

```
class Bird:
       def speak(self):
           return "Chirp!"

   class Mammal:
       def speak(self):
           return "Growl!"

   class Bat(Bird, Mammal):
       pass

   bat = Bat()
   print(bat.speak())  # Output: "Chirp!" (inherits from Bird)
```

3. Multilevel Inheritance:

- In multilevel inheritance, a subclass derives from another subclass, creating a chain of inheritance.
- Each class in the hierarchy inherits the attributes and methods of the previous class.

```
   class Animal:
       def speak(self):
           pass

   class Mammal(Animal):
       def speak(self):
           return "Growl!"

   class Dog(Mammal):
       def speak(self):
           return "Woof!"
```

4. Hierarchical Inheritance:

- Hierarchical inheritance involves multiple subclasses inheriting from a single superclass.
- Each subclass can have its own attributes and methods in addition to those inherited from the superclass.

```
   class Shape:
       def area(self):
           pass

   class Circle(Shape):
       def area(self):
           return 3.14159 * self.radius ** 2

   class Rectangle(Shape):
       def area(self):
           return self.width * self.height
```

5. Hybrid (or Cyclic) Inheritance:

- Hybrid inheritance combines various forms of inheritance, such as multiple inheritance, single inheritance, and more, in a single program.
- Care should be taken to avoid ambiguity or conflicts that can arise with hybrid inheritance.

```
class A:
      def speak(self):
          return "A"

  class B(A):
      def speak(self):
          return "B"

  class C(A):
      def speak(self):
          return "C"

  class D(B, C):
      pass
```

---

# Unit 8 ( Visualizing Data )

## What is data visualization?

**1:** Data visualization is the graphical representation of data and information to help people understand, analyze, and interpret it more easily.
**2:** It involves the use of visual elements such as charts, graphs, maps, and other visual aids to present data in a way that is meaningful and comprehensible.
**3:** Data visualization is a powerful tool for conveying complex information, patterns, and insights that might be difficult to discern from raw data alone.
**4:** Data visualization transforms data into visual elements like bars, lines, and scatter plots, aiding in trend and anomaly detection.
**5:** By effectively communicating insights, it reaches a broad audience, including non-experts and decision-makers.
**6:** Ultimately, it supports data-informed decision-making by presenting a clear, intuitive view of data for more informed choices and actions.

---

## Data visualization in python

Data visualization in Python can be accomplished using various libraries, with some of the most popular ones being Matplotlib, Seaborn, Plotly, and Bokeh. These libraries offer a wide range of tools and features to create compelling visualizations. Here's a brief overview of each:

1. Matplotlib: Matplotlib is a widely-used library for creating static, interactive, and animated plots in Python. It provides a comprehensive set of functions for creating various types of visualizations, including line charts, bar plots, scatter plots, histograms, and more.

Example code for creating a simple line plot using Matplotlib:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 12]

plt.plot(x, y)
```

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
plt.show()
```

2. Seaborn: Seaborn is built on top of Matplotlib and provides a high-level interface for creating aesthetically pleasing statistical visualizations. It simplifies the process of creating complex plots and is especially useful for data analysis and exploration.

Example code for creating a bar plot using Seaborn:

```
import seaborn as sns
import matplotlib.pyplot as plt

data = {'Category': ['A', 'B', 'C', 'D'],
        'Value': [20, 45, 30, 60]}

sns.barplot(x='Category', y='Value', data=data)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show()
```

3. Plotly: Plotly is a versatile library for creating interactive and web-based visualizations. It supports a wide range of chart types and is well-suited for creating dashboards and web applications.

Example code for creating an interactive line plot using Plotly:

```
import plotly.express as px

data = {'X': [1, 2, 3, 4, 5],
        'Y': [10, 15, 13, 18, 12]}

fig = px.line(data, x='X', y='Y', title='Interactive Line Plot')
fig.show()
```

4. Bokeh: Bokeh is another library for creating interactive visualizations. It is designed to work well with large datasets and is highly customizable. Bokeh can be used for creating dynamic and interactive dashboards.

Example code for creating an interactive scatter plot using Bokeh:

```
from bokeh.plotting import figure, output_file, show

x = [1, 2, 3, 4, 5]
```

```
y = [10, 15, 13, 18, 12]

output_file("scatter.html")

p = figure(title="Interactive Scatter Plot")
p.circle(x, y, size=10)

show(p)
```

---

# matplolib and seaborn

Matplotlib and Seaborn are two popular Python libraries for creating data visualizations, and they are often used in combination. Matplotlib provides the fundamental building blocks for creating a wide variety of plots, while Seaborn simplifies the process of generating aesthetically pleasing statistical visualizations. Here's an overview of both libraries and how they can be used together:

1. Matplotlib:

- Matplotlib is a versatile and low-level library for creating static, animated, or interactive visualizations in Python.

- It provides fine-grained control over plot elements, making it suitable for creating customized, publication-quality plots.

- Matplotlib offers a wide range of chart types, including line plots, bar charts, scatter plots, histograms, and more.

- It is highly customizable, allowing you to control aspects such as colors, line styles, and marker types.

Example of creating a simple line plot with Matplotlib:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 12]
```

```
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
plt.show()
```

2. Seaborn:
   - Seaborn is built on top of Matplotlib and provides a higher-level, easy-to-use interface for creating statistical visualizations.
   - It is particularly useful for data exploration, as it simplifies the process of generating complex plots like heatmaps, pair plots, and violin plots.
   - Seaborn offers a wide range of color palettes and themes to make your visualizations more visually appealing.
   - It is designed for use with Pandas DataFrames and can automatically handle tasks like data aggregation and summarization.

Example of creating a bar plot with Seaborn:

```
import seaborn as sns
import matplotlib.pyplot as plt

data = {'Category': ['A', 'B', 'C', 'D'],
        'Value': [20, 45, 30, 60]}

sns.barplot(x='Category', y='Value', data=data)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show()
```

---

# Line charts

A line chart, also known as a line plot or line graph, is a common type of data visualization that is used to represent data points over a continuous interval or time period. Line charts are particularly useful for showing trends and changes in data over time and for displaying the relationship between two continuous variables. Here's how to create a basic line chart using Matplotlib, one of the popular Python libraries for data visualization:

```
import matplotlib.pyplot as plt
```

```
# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 12]

# Create a line chart
plt.plot(x, y)

# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Chart Example')

# Display the chart
plt.show()
```

In this example:

1. `x` represents the values on the x-axis, typically indicating time or a continuous variable.
2. `y` represents the values on the y-axis, showing the data points you want to plot over time or along the continuous interval.
3. `plt.plot(x, y)` creates the line chart, connecting the data points with lines.

You can customize line charts in various ways using Matplotlib. For instance, you can change line styles, colors, add markers to data points, and adjust axis scales. Here are a few additional options you can use:

- Change the line style: `plt.plot(x, y, linestyle='--')`
- Change the line color: `plt.plot(x, y, color='red')`
- Add markers to data points: `plt.plot(x, y, marker='o')`
- Customize the axis limits: `plt.xlim(0, 6)` for the x-axis, `plt.ylim(0, 20)` for the y-axis.

---

# Bar graphs

Bar graphs, also known as bar charts or bar plots, are a common type of data visualization used to display categorical data and compare values between different categories or groups. They are particularly effective for showing discrete data or data that is not continuous. Here's how to create a basic bar graph using Matplotlib in Python:

```
import matplotlib.pyplot as plt
```

```
# Sample data
categories = ['Category A', 'Category B', 'Category C', 'Category D']
values = [20, 45, 30, 60]

# Create a bar graph
plt.bar(categories, values)

# Add labels and title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Graph Example')

# Display the graph
plt.show()
```

In this example:

1. `categories` represents the categorical labels or groups for the x-axis.
2. `values` represents the corresponding values for each category on the y-axis.

The `*plt.bar*()` function is used to create the bar graph. It generates bars for each category with heights proportional to the values you provide. You can customize your bar graph in various ways using Matplotlib:

- Change the bar color: `*plt.bar*(categories, values, color='blue')`
- Add error bars: `*plt.bar*(categories, values, yerr=[2, 3, 2, 4])`
- Stack bar graphs for multiple datasets: `*plt.bar*(categories, values1, label='Dataset 1')` and then `*plt.bar*(categories, values2, label='Dataset 2', bottom=values1)` for another dataset.
- Create horizontal bar graphs: Use `plt.barh()` instead of `*plt.bar*()`.

---

# Histogram

A histogram is a graphical representation of the distribution of a dataset, showing the frequency or probability of different values or value ranges within the data. It is particularly useful for visualizing the underlying shape of a dataset, identifying patterns, and understanding the distribution of values. Here's how to create a basic histogram using Matplotlib in Python:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
# Generate some sample data
data = np.random.randn(1000)  # Create a random dataset of 1000 values

# Create a histogram
plt.hist(data, bins=20, edgecolor='black')  # 'bins' determine the
number of bins or bars in the histogram

# Add labels and title
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram Example')

# Display the histogram
plt.show()
```
In this example:

1. `data` represents the dataset for which you want to create a histogram. It contains random values in this case.

2. `plt.hist()` is used to create the histogram. The `bins` parameter determines the number of bins or bars used in the histogram.

You can customize your histogram in various ways using Matplotlib:

- Change the color of the bars: `plt.hist(data, bins=20, edgecolor='black', color='blue')`
- Adjust the range of values to be displayed: `plt.hist(data, range=(-3, 3), bins=20, edgecolor='black')`
- Display cumulative histograms: `plt.hist(data, bins=20, edgecolor='black', cumulative=True)`
- Normalize the histogram to show probabilities: `plt.hist(data, bins=20, edgecolor='black', density=True)`

---

# Scatter plot

A scatter plot is a type of data visualization that is used to display individual data points in a two-dimensional coordinate system. Each point in the scatter plot represents a pair of values from a dataset, typically one value on the x-axis and another on the y-axis. Scatter plots are useful for visualizing the relationship between two variables and identifying patterns or correlations in the data. Here's how to create a basic scatter plot using Matplotlib in Python:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 12]

# Create a scatter plot
plt.scatter(x, y)

# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot Example')

# Display the scatter plot
plt.show()
```

In this example:

1. `x` represents the values on the x-axis.
2. `y` represents the values on the y-axis.

The `plt.scatter()` function is used to create the scatter plot, and it places a point at the specified (x, y) coordinates for each data point.

You can customize your scatter plot in various ways using Matplotlib:

- Change the marker style and color: `plt.scatter(x, y, marker='o', color='blue')`
- Adjust the size of markers: `plt.scatter(x, y, s=100)`
- Add labels to individual data points: `for i, txt in enumerate(y): plt.annotate(txt, (x[i], y[i]))`
- Display multiple datasets in the same scatter plot by calling `plt.scatter()` multiple times.

---