Python Programming for Beginners

Table Of Content

Using the ***with*** Statement (Context Managers)

Object-Oriented Programming (OOP)
Classes and Objects
Constructors and Destructors
Inheritance
Polymorphism
Encapsulation
Abstraction

Modules and Packages
Creating and Importing Modules
Standard Library Modules
Using External Packages (e.g., pip)

Introduction to Python

What is Python?
Python is a high-level, general-purpose programming language known for its simplicity and readability. It was created by *Guido van Rossum* and first released in 1991. Python is designed with an emphasis on code readability and has a clean and easy-to-understand syntax, which makes it an ideal language for beginners and professionals alike. It is a versatile language used in various domains, including web development, data analysis, scientific computing, artificial intelligence, automation, and more.

Why Learn Python?

Python is a popular and versatile programming language, and there are several compelling reasons to learn it:

1. Ease of Learning: Python's clear and concise syntax makes it an excellent choice for beginners. It's often recommended as a first programming language.

2. Versatility: Python is used in a wide range of applications, from web development to data science to artificial intelligence and more. Learning Python opens up diverse career opportunities.

3. Strong Community and Support: Python has a large and active community of developers. You can find abundant resources, libraries, and frameworks to help you with your projects.

4. Job Opportunities: Python is in high demand in the job market. Many companies and organizations seek Python developers for various roles.

5. Data Science and Machine Learning: Python has become the de facto language for data science and machine learning. Libraries like NumPy, Pandas, and TensorFlow are widely used in these fields.

6. Automation and Scripting: Python is excellent for automating repetitive tasks and writing scripts. It simplifies everyday tasks and enhances productivity.

Setting Up Python (Installation)

You are strongly advised to get the help of your instructor in the Installations and setting up of your machine.

Getting Started with Python

Python is an approachable and versatile programming language that's great for beginners. In this guide, we'll take you through some basic and important areas of programming or what we called grammars of programming.

Python Comment

Comments in Python are used to provide explanations and context within your code. They are not executed by the Python interpreter and are solely for the benefit of developers. Comments improve code readability, aid in documentation, and make it easier to understand the purpose and functionality of your code. This guide covers various types of Python comments.

I. Single-Line Comments

In Python, you can create single-line comments using the **#** symbol. Anything following the **#** on the same line is treated as a comment and is ignored by the interpreter.

```
# This is a single-line comment
x = 100  # This comment is at the end of the line
```

II. Multi-Line Comments

Python does not have a dedicated syntax for multi-line comments. However, you can create multi-line comments by placing multiple single-line comments consecutively. There's also an alternative approach using multi-line strings (triple-quoted strings).

Consecutive Single-Line Comments

```
# This is a multi-line comment
# Line 2 of the comment
# Line 3 of the comment
```

III. Multi-Line Strings (Triple-Quoted Strings)

You can use triple-quoted strings (either single-quotes or double-quotes) to create multi-line comments. These strings are usually used for docstrings, which are also a form of documentation.

```
'''
This is a multi-line comment using triple-quotes.
It can span multiple lines.
'''
```

Comments for Documentation

Comments are not only for adding explanations to your code but also for documenting your code for other developers (including your future self). A common practice is to use comments to document functions, classes, and modules using docstrings.

```
def salery(a, b):
    """
    This function adds two numbers.

    Args:
        a (int): The first number.
        b (int): The second number.
    Returns:
        int: The sum of a and b.
    """
    return a + b
```

Python Variable

A variable is a name that refers to a value, object, or data structure stored in memory. It is also referred to as container that are used to store and manage data in your Python programs. Examles:

```
x = 11
name = "Mark"
is_student = True
```

The ***x, name*** *and* ***is_student*** are what is called variables that stores their specific values.

Python Operators

Operators in Python are symbols or special keywords used to perform operations on variables and values. They enable you to manipulate data, perform mathematical computations, compare values, and control the flow of your program.

Types of Operators

1. Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numeric values.

```
Addition (+), Subtraction (-), Multiplication (*),
Division (/), Modulus (%), Exponentiation (**),
Floor Division (//)
```

Example:

```
a = 100
b = 5

addition = a + b
subtraction = a - b
multiplication = a * b
division = a / b
modulus = a % b
exponentiation = a ** b
floor_division = a // b
```

2. Comparison Operators

Comparison operators are used to compare two values.
Equal to (==): Checks if two values are equal.
Not equal to (!=): Checks if two values are not equal.
Greater than (>): Checks if the left value is greater than the right value.

Less than (<): Checks if the left value is less than the right value.
Greater than or equal to (>=): Checks if the left value is greater than or equal to the right value.
Less than or equal to (<=): Checks if the left value is less than or equal to the right value.

Example:

```
x = 10
y = 100

is_equal = x == y
is_not_equal = x != y
is_greater = x > y
is_less = x < y
is_greater_or_equal = x >= y
is_less_or_equal = x <= y
```

3. Logical Operators

Logical operators are used to combine conditional statements.

Logical AND (*and*): Returns `True` if both conditions are `True`.
Logical OR (*or*): Returns `True` if at least one condition is `True`.
Logical NOT (*not*): Returns the opposite of the condition.

Example:

```
a = True
b = False

logical_and = a and b
logical_or = a or b
logical_not = not a
```

4. Assignment Operators
Assignment operators are used to assign values to variables.

Assignment (=): Assigns the value on the right to the variable on the left.
Add and assign (+=): Adds the right value to the variable and assigns the result.
Subtract and assign (-=): Subtracts the right value from the variable and assigns the result.
Multiply and assign (*=): Multiplies the variable by the right value and assigns the result.
Divide and assign (/=): Divides the variable by the right value and assigns the result.
Modulus and assign (%=): Calculates the remainder of the variable and assigns the result.
Exponentiation and assign (=):** Raises the variable to the power of the right value and assigns the result.
Floor Division and assign (//=): Performs floor division on the variable and assigns the result.

Example:

```
x = 10
y = 2

x += y  # Is the as the expression x = x + y
x -= y  # Is the as the expression x = x - y
x *= y  # Is the as the expression x = x * y
x /= y  # Is the as the expression x = x / y
x %= y  # Is the as the expression x = x % y
x **= y  # Is the as the expression x = x ** y
x //= y  # Is the as the expression x = x // y
```

5. Identity Operators

Identity operators are used to compare the memory location of two objects.

Identity (is): Returns *True* if both variables refer to the same object.
Non-identity (is not): Returns *True* if both variables refer to different objects.

Example:

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

is_same_object = x is y  # True (x and y reference the
same list)
is_different_object = x is not z  # True (x and z
reference different lists with the same values)
```

6. Membership Operators

Membership operators are used to test if a sequence contains a specific value.

Membership (in): Returns *True* if a value is found in the sequence.
Non-membership (not in): Returns *True* if a value is not found in the sequence.

Example:

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

is_apple_in_fruits = "apple" in fruits  # True
is_mango_not_in_fruits = "mango" not in fruits  # True
```

7. Bitwise Operators

Bitwise operators are used to perform operations on binary representations of values.

Bitwise AND (&): Performs a bitwise AND operation.
Bitwise OR (|): Performs a bitwise OR operation.
Bitwise XOR (^): Performs a bitwise XOR (exclusive OR) operation.
Bitwise NOT (~): Performs a bitwise NOT (complement) operation.
Bitwise Left Shift (<<): Shifts the bits to the left.
Bitwise Right Shift (>>): Shifts the bits to the right.

Python Data Structure And Data Types

In programming, a data structure is a way of organizing, storing, and managing data to perform operations efficiently. Example: Lists, Tuples, Dictionaries, Sets, and Strings.

Data Type is a classification that specifies which type of value a variable can hold, what operations can be performed on it, and how the variable is stored in memory.

1. Numeric Data Types

Three types of numerical data types:

I. Integers (int)

Integers are whole numbers, both positive. Examples -5, 0, and 42.

```
1    x = 10
2    y = -5
3    x = 100
```

II. Floating-Point Numbers (*float*)

Floating-point numbers represent real numbers with a decimal point. Examples 3.14, -0.1, and 2.71828.

```
1    x = 2.71828
2    y = -0.1
3    x = 3.14
```

III. Complex Numbers (*complex*)

Complex numbers consist of a real part and an imaginary part, denoted with a j or J. Examples include 3+2j and -1-4J.

```
1    x = -1-4J
2    y = 3+2j
```

2. Text Data Type

I. Strings (*str*)

Strings are sequences of characters, enclosed in single (' ') or double (" ") quotes. Examples "Hello, How has this material helped you?" and '12345'.

```
message = "Hello, How is has this material helped you?"    # letter string on double quote
info = '1234565'    # text string on single quote
```

3. Sequence Data Types

I. Lists (*list*)

Lists are ordered collections of elements. They can hold items of different data types and are mutable (they can be changed or modified). They are defined with square bracket [].

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

II. Tuples (*tuple*)

Tuples are ordered collections like lists but are immutable (cannot be changed after creation). They are defined with parentheses `()`.

```
fruits = ("apple", "banana", "cherry")
```

III. Ranges (*range*)

Ranges are used to represent a sequence of numbers. They are often used in loops.

```
numbers = range(1, 6)  # Represents numbers from 1 to 5
```

4. Mapping Data Type

I. Dictionaries (*dict*)

Dictionaries store data as key-value pairs. Keys are unique, and values can be of any data type.

```
person = {"name": "Alice", "age": 30}
```

5. Set Data Types

I. Sets (*set*)

Sets are collections of unique elements. They are unordered and mutable.

```
colors = {"red", "green", "blue"}
```

II. Frozen Sets (frozenset)

Frozen sets are like sets, but they are immutable (cannot be changed after creation).

```
frozen_set = frozenset([1, 2, 3])
```

6. Boolean Data Type

I. Boolean (*bool*)

Boolean represent the truth values *True* and *False*. They are often used for logical operations and conditional statements.

```
is_raining = True
is_sunny = False
```

7. None Type

I. None (*NoneType*)

None is a special data type that represents the absence of a value or a null value. It's often used to initialize variables before assigning real values.

```
cashew = None
```

Control Structures in Python

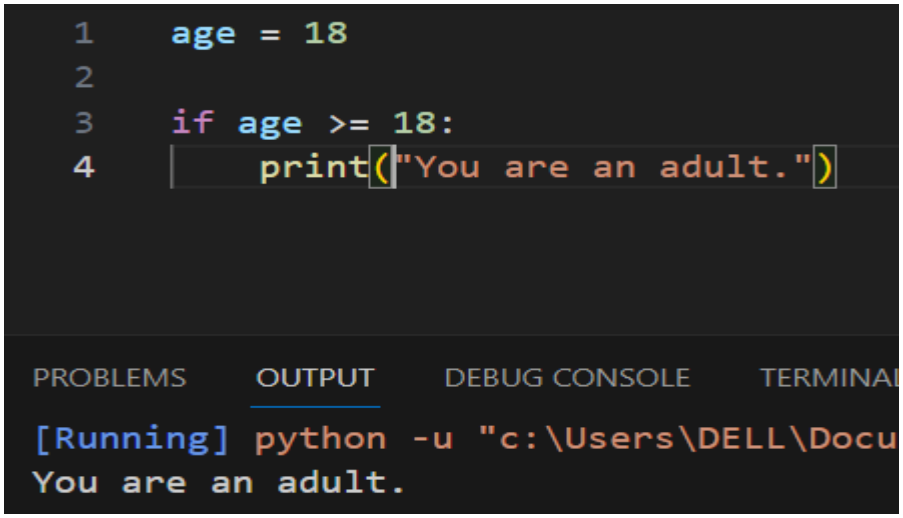
Control structures in Python allow you to control the flow of your programs by making decisions, repeating actions, and managing the execution of your code. In this guide, we'll cover the essential control structures: conditional statements, loops, and control statements.

Conditional Statements (if, elif, else)

Conditional statements in Python are used to make decisions based on specific conditions. The primary keywords for conditional statements are *if, elif (short for "else if"), and else*.

The *if* Statement

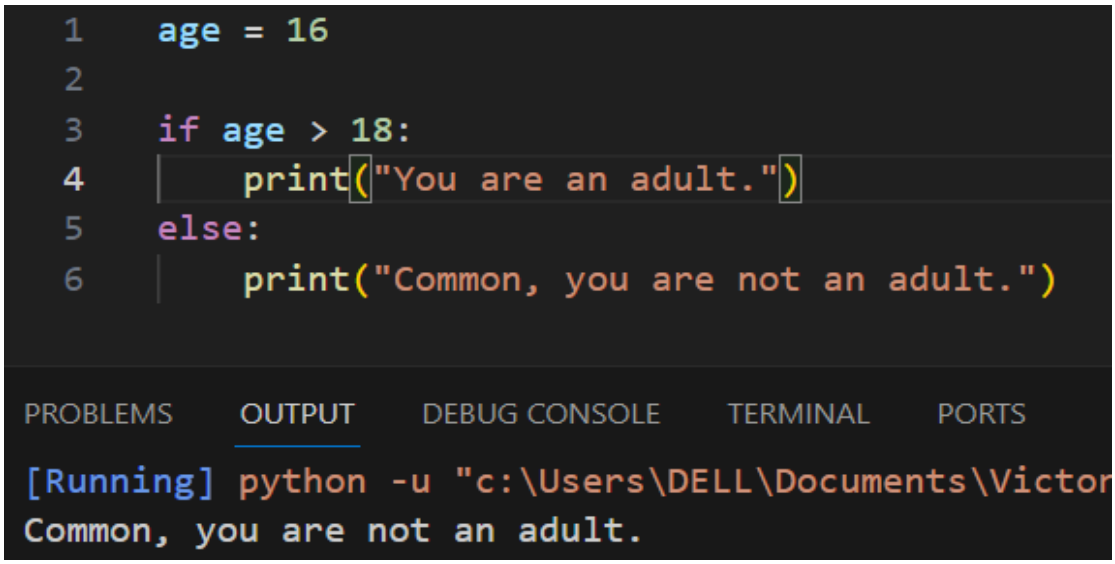
The *if* statement is used to execute a block of code if a condition is true.



The statement above displays ***'You are an adult.'*** in the output section because the condition was true. I.e, **age** is truly equals to 18.

The *if - else* Statement

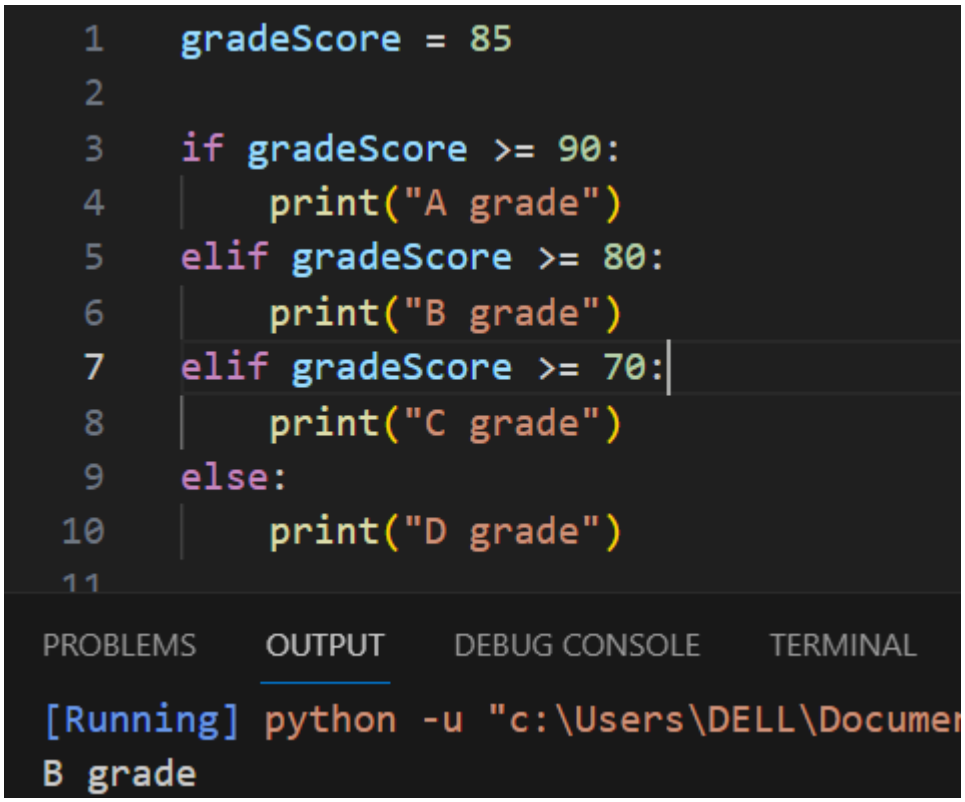
The *if - else* statement allows you to execute one block of code when a condition is true and another block when it's false.



The statement above displays ***'Common, you are not an adult.'*** in the output section because the condition was not true. I.e, **age** is not greater than 18.

The *if - elif - else* Statement

The if - elif - else statement lets you test multiple conditions and execute different blocks of code based on the first condition that is true.



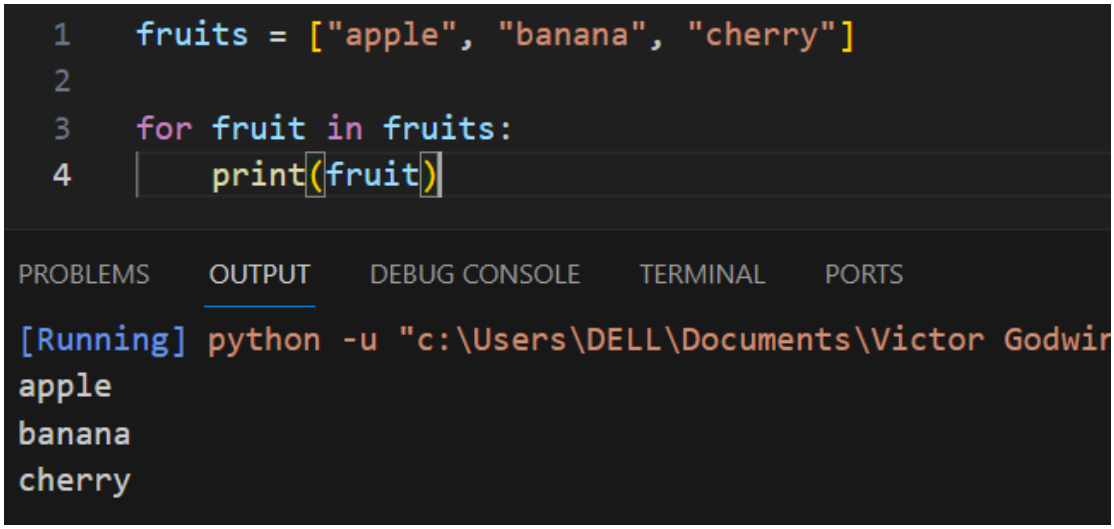
The statement above displays ***'B grade'*** in the output section because the condition was found true in the first *elif statement*. I.e, **gradeScore** is greater than 80.

Loops (*for and while*)

Loops are used to execute a block of code repeatedly. Python provides two main types of loops: *for loop* and *while loop*.

The for Loop

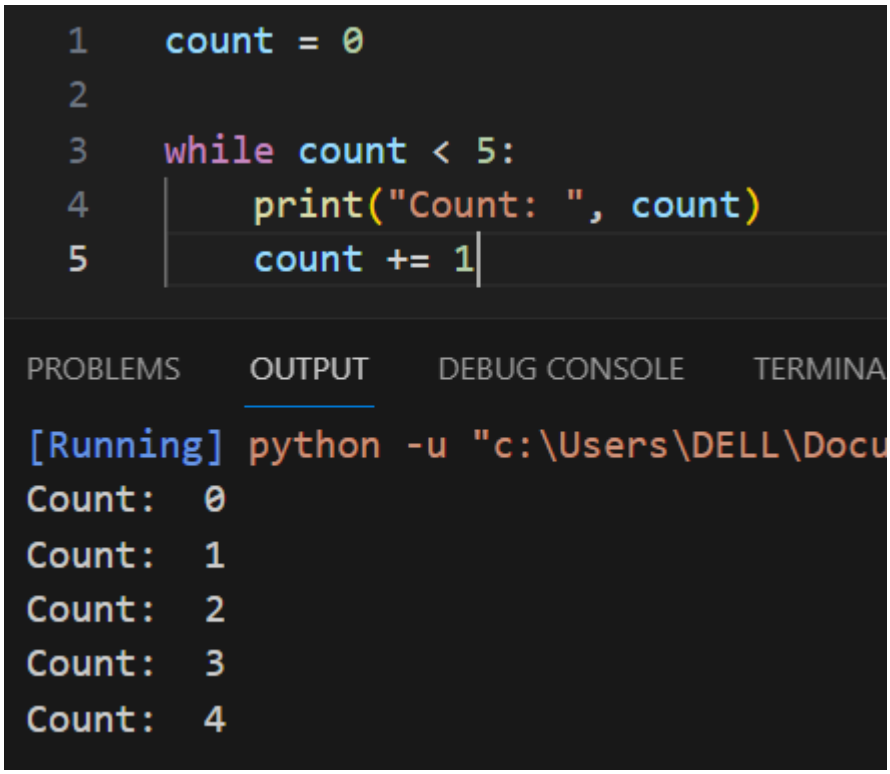
A *for* loop is used to iterate over a sequence (such as a list, tuple, or range) or other iterable objects.



The statement above displays the printed list of fruits in the output section. List in python is a type of data rendering in python. The ***fruit*** is the *key* that represents all the values in the fruits variable. It's what the for loop rely on to be able to display list of items in short time.

The while Loop

A *while* loop continues executing a block of code as long as a condition remains true.



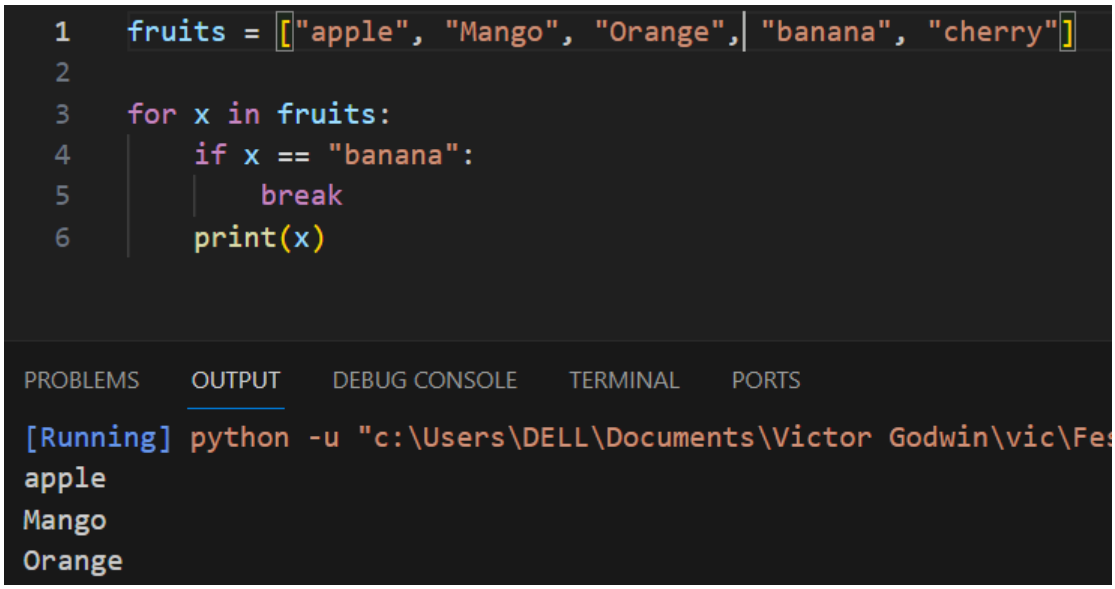
The statement above displays the count from ***'0 - 4'*** in the output section because the condition is true that ***count*** variable is actually less than 5.

Control Statements (*break, continue, pass*)

Control statements allow you to alter the flow of your loops and conditional statements.

***break* Statement**

The *break* statement is used to exit a loop prematurely.



The statement above displays the list of fruits in the output section but when it got to ***"banana"*** it broke the loop and displayed every of the items before banana.

***continue* Statement**

The *continue* statement is used to skip the current iteration and continue with the next one.

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

for x in numbers:
    if x % 2 == 0:
        continue
    print(x)

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] python -u "c:\Users\DELL\Documen
1
3
5
```

Here, even numbers are skipped, and only odd numbers are printed. The operator % is called modulus or remainder operator in programming, it states that if x is divided by 2 and the remainder is 0. Therefore it checks through those numbers in the list that returns remainder of 0.

***pass* Statement**

The *pass* statement is a placeholder for future code. It does nothing and is often used when a statement is syntactically required but you don't want to execute any code.

```python
x = 10

if x < 0:
    pass
else:
    print("Positive or zero")
```

Python Functions

Functions are blocks of reusable and executable code that perform specific tasks when they are being called. They play a fundamental role in structuring and organizing code. We will cover various aspects of Python functions, which includes defining functions, using parameters and arguments, working with return statements, and understanding the scope and lifetime of variables.

Defining Functions

In Python, functions are defined using the ***def*** keyword, followed by the function ***name*** and ***parentheses***. You can also include parameters within the parentheses, and the function body is indented. Functions can be created with or without return values.

```python
# Function without parameters and return value
def greet():
    print("Hello, Python!")

# Function with parameters and return value
def add(a, b):
    result = a + b
    return result
```

Parameters and Arguments

Parameters
Parameters are placeholders in the function definition. They allow you to pass data into a function. Parameters are defined inside the parentheses when you define a function.

```python
def collection(name):
    print(f"Hello, {name}!")

def add(a, b):
    result = a + b
    return result

# name in the parentheses is a parameter
# a, b in the parentheses is a parameter
```

Arguments

Arguments are the actual values passed to a function when it is called. Arguments are provided when you invoke the function.

```python

# Function calls with arguments
collection("David")
sum_result = add(5, 3)
```

Return Statements

Return statements are used to specify the value that a function should return. If a function does not have a return statement, it returns ***None*** by default.

```python
def add(a, b):
    result = a + b
    return result

def multi_return(a, b):
    sum_result = a + b
    product_result = a * b
    return sum_result, product_result
```

Scope and Lifetime of Variables

Local Variables

Variables defined within a function have local scope. They are only accessible within the function. Their lifetime begins when the function is called and ends when the function exits.

```
def funC():
    lv = "I'm local"
    print(lv)
```

Global Variables

Variables defined outside any function have global scope. They are accessible from any part of the code.

```
gv = "I'm global"

def funC():
    print(gv)
```

Modules and Packages in Python

Python modules and packages are essential for organizing, reusing, and sharing code. In this guide, we will explore creating and importing modules, using standard library modules, and incorporating external packages using tools like ***pip***.

Modules

Modules in Python are files that contain Python code. Each module can define bunch of functions, variables, and classes that can be reused in other parts of your program. You too can create your own modules for easy work.

Creating Modules

To create a module, you need to save your Python code in a ***.py*** file extension. For example, if you have a file named ***pmodule.py*** containing functions and variables, you can import and use them in other Python scripts.

```
# module.py

def welcome(name):
    return f"Hello, {name}!"

PI = 3.14159265359
```

Importing Modules

You can import modules into your Python scripts using the ***import*** statement. This allows you to access functions, variables, and classes defined in the module.

```
# main.py

import pmodule

result = pmodule.welcome("Gabriel")
print(result)  # Output: Hello, Gabriel!

print(pmodule.PI)  # Output: 3.14159265359
```

Packages

Packages are a way to organize related modules into directories and subdirectories. Packages are used to avoid naming conflicts and to structure larger projects.

Creating Packages

To create a package, you need to create a directory *(folder)* containing a special file named *__init__.py*. This file can be empty or contain initialization code. Within the package directory, you can place one or more modules.

```
my_package/
    __init__.py
    module1.py
    module2.py
```

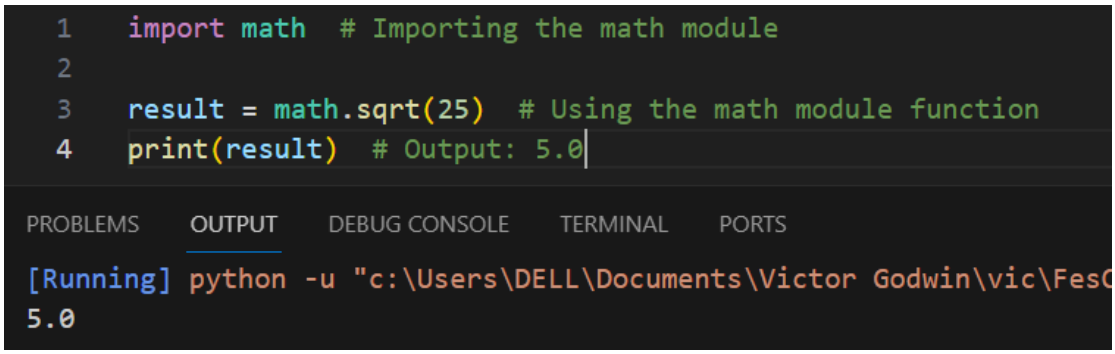
Importing Modules from Packages

You can import modules from packages just like you import regular modules. Use dot notation to specify the package and module you want to import.

```
# main.py

import my_package.module1
import my_package.module2

result1 = my_package.module1.func()
result2 = my_package.module2.func_two()

print(result1)
print(result2)
```

Standard Library Modules

Python's standard library includes a wide range of modules that provide common functionality without the need for external packages. These modules cover areas such as *file handling, mathematical operations, data manipulation, and more.* You can import and use these modules in your code without any additional installation.



Using External Packages (e.g., *pip*)

External packages, also known as third-party packages, extend Python's capabilities. The Python Package Index (***PyPI***) is a repository of thousands of external packages that you can easily install and use in your projects.

Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes for organizing code. In Python, OOP is a fundamental concept, and it allows you to model real-world entities and their interactions. This guide covers various aspects of OOP in Python.

Classes and Objects

Classes

A class is a blueprint for creating objects. It defines the structure and behavior of objects. Classes are like ***templates*** that specify how objects of that class will behave.

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

    def height(self):
        print(f"{self.name} is tall")

# Creating an object (instance) of the Peraon class
My_person= Person("Mark", "Dark")
```

Objects

Objects are instances of classes. They are real entities created based on the blueprint defined by a class.

```python
My_person.height()  # Calling a method of the object
```

Constructors and Destructors

Constructors

A constructor is a special method used to initialize objects when they are created. In Python, the constructor is named ***__init__***. It sets initial values for object attributes.

```
class Person_II:
    def __init__(self, name, school, age):
        self.hisName = name
        self.hisSchool = school
        self.hisAge = age
```

Destructors

Destructors are used to clean up resources before an object is destroyed. In Python, the destructor is named ***__del__***. It's not commonly used, and resource cleanup is often handled in other ways.

```
class Person_II:
    def __del__(self):
        print(f"Deleting {self.name}
            {self.school} {self.age}")
```

Inheritance

Inheritance is a mechanism in OOP where a new class (***subclass or derived class***) is based on an existing class (***base class or superclass***). It allows the reuse and extension of class functionality.

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

    def school(self):
        Pass

class Person_II(Person_I):
    def school(self):
        return f"{self.name} BizMarrow
                                Technologies!"

class Person_III(Person_I):
    def school(self):
        return f"{self.name} Udemy!"
```

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables you to write code that can work with objects of multiple classes in a consistent way.

```
animals = [Cow("White"), Goat("Dark")]

for x in animals:
    print(x.colors())
```

Encapsulation

Encapsulation is the concept of restricting access to certain parts of an object, typically through the use of private and protected members (attributes and methods). In Python, this is achieved through naming conventions.

- ***_attribute***: A single underscore indicates a protected attribute.
- ***__attribute***: A double underscore indicates a private attribute.

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

    def area(self):
        return 3.14 * self.__radius ** 2
```

Abstraction

Abstraction is the concept of simplifying complex systems by breaking them into smaller, more manageable parts. In OOP, you can hide the complex implementation details and provide a simplified interface to users.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius ** 2
```

Exception Handling and File Handling in Python

Exception Handling

Handling Errors with ***try*** and ***except***

Exception handling in Python allows you to handle errors or exceptions that may occur during program execution. You use the ***try*** and ***except*** blocks to catch and handle exceptions.
A good example, is the ***ZeroDivisionError***

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Handle the specific exception
    result = "Error: Cannot divide by
       zero"
```

In this example, a ***ZeroDivisionError*** is caught, and the code proceeds to the ***except*** block to handle the exception.

Using *finally*

The *finally* block is used to specify code that should always be executed, whether an exception is raised or not.

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Handle the specific exception
    result = "Error: Division by zero"
finally:
    # This block is executed regardless of
    exceptions
    cleanup_resources()
```

The ***finally*** block is useful for tasks like resource cleanup or closing files.

Custom Exception Classes

You can create custom exception classes by inheriting from Python's built-in **Exception** class. Custom exceptions are useful for handling *application-specific errors*.

```
class MyCustomError(Exception):
    def __init__(self, message):
        super().__init__(message)

try:
    # Code that may raise a custom
        exception
    raise MyCustomError("This is a custom
            exception.")
except MyCustomError as e:
    # Handle the custom exception
    print(f"Custom exception caught: {e}")
```

Custom exceptions can be raised with the ***raise*** statement and caught using ***except*** blocks.

File Handling

Reading and Writing Files

Python provides built-in functions for reading and writing files. You can use ***open()*** function to open a file. It has modes with which it operates to carry out tasks. Modes like (***'r'*** for *reading*, ***'w'*** for *writing*, ***'a'*** for *appending*), and use methods like ***read(), write(), and close()*** to interact with files.

Reading a File

In this examples we shall be using the automatic closing of files with the use of the (***with*** keyword)

```
# using close() method and file on r - read mode
file = open("example.txt", "r")
file.read()
print(file)
file.close()


# Using automatic close() method - (with keyword)
with open("example.txt", "r") as file:
    contents = file.read()
    print(contents)
```

Writing to a File

```
# using close() method, on w- write mode
file = open("output.txt", "w")
file.write()
print(file)
file.close()


# Using automatic close() method - (with keyword)
with open("output.txt", "w") as file:
    file.write("This is a sample text.")
```

Working with Text and Binary Files

Files can be opened in *text mode (default) or binary mode.* Text mode is used for reading and writing text files, while binary mode is used for non-text files, such as images.

Text Mode

```
with open("text.txt", "r") as text_file:
    contents = text_file.read()
```

Binary Mode

```
# Binary files are always on rb - read binary

with open("image.jpg", "rb") as binary_file:
    data = binary_file.read()
```