Lesson:

Java

Python One Shot

PW SKILLS

Introduction to Programming

Programming is the process of creating a set of instructions that tell a computer how to perform a task. Programming can be done using a variety of computer programming languages, such as JavaScript, Python, Java, and C++ etc.

Eg: MS Word, MS Excel, Adobe Photoshop, Internet Explorer, Chrome, etc., are examples of computer programs.

<For Editor: Change text below Source Code to - Written in High-level language eg: C, C++, Java, Python etc.>

Introduction to Python and DSA

Python is a versatile, beginner-friendly and a high-level programming language that has gained immense popularity for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python is widely used in various domains such as web development, data analysis, scientific computing, artificial intelligence, and automation.

It provides a solid foundation for learning data structures and algorithms (DSA), which are fundamental concepts in computer science.

By mastering DSA, you'll gain the skills to:

- Optimize code
- Design efficient algorithms
- Tackle real-world challenges more effectively

But why Python?

1. Beginner-friendly and easy to learn.
2. Clear and readable syntax.
3. Well-documented standard library.
4. Open-source with a large developer community.
5. Scalable for large-scale projects.
6. High productivity due to concise and expressive code.
7. Cross-platform compatibility.
8. Supports multiple programming paradigms (procedural, object-oriented, functional).
9. Versatile and widely used across various domains.
10. Strong in data analysis and scientific computing.

Installation of Python Interpreter

You can follow the given steps to install Python Interpreter to start coding:

1. Visit the official Python website at **http://www.python.org.**
2. Navigate to the Downloads section and choose the appropriate Python version for your operating system (Windows, macOS, or Linux).
3. Click on the download link for the desired version, such as Python 3.9.5(Note: Python 2 versions and Python 3 versions are not compatible. In this series, we are going to learn Python 3, so you can download any version of Python 3, like Python 3.9.5.).

4. Once the download is complete, locate the installer file and double-click to run it.
5. In the installer, select the option to "Install Now" or customize the installation if desired.
6. On Windows, make sure to check the box that says "Add Python to PATH" to make it accessible from the command line.
7. Follow the prompts and agree to the license terms to proceed with the installation.
8. The installer will install Python and its associated tools onto your system.
9. Once the installation is complete, you can verify it by opening a terminal or command prompt and typing "python --version" or "python3 --version" to see the installed Python version.
10. Congratulations! You now have Python installed on your system and can start writing and executing Python programs.

Note: You can follow this **https://code.visualstudio.com/docs/python/python-tutorial** for python installation with vs code.

Basic Program in Python

Let's Code in Python
Once you are done with the installation, you can write Python code in a text editor or an Integrated Development Environment (IDE) like PyCharm or Visual Studio Code, and then execute it using the Python interpreter.

Python uses a straightforward syntax that emphasizes code readability. Here's an example of a simple Python program that prints "Hello, World!":

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

To be able to run this program:

- First, write the program.
- Save this in a file, say "hello.py".
- Now, run this program using "python hello.py" or "python3 hello.py".

Congratulations!! You have successfully written your first Python program. In Python, the print() function is used to display output on the screen. It allows you to print text, variables, and expressions.

How to move to next line?
To move to the next line while printing in Python, you can use the special character \n, which represents a newline. Here's an example:

```
print("Hello,\nWorld!")
```

In the above code, the \n character is inserted between "Hello," and "World!" to create a line break.

You can also achieve the same effect by using the end parameter of the print() function. By default, the end parameter is set to "\n", which adds a newline character at the end of each print() statement. You can change the value of end to achieve different behaviors:

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

In this example, the end parameter is set to a space character " ", which appends a space instead of a newline character. Therefore, the second print() statement starts on the same line as the first one.

Using the end parameter, you can customize the behavior of the print() function to move to the next line or stay on the same line as needed.

Python CLI

- A command-line interface or command language interpreter (CLI) is a means of interacting with a computer program where the user (or client) issues commands to the program in the form of successive lines of text.
- Having even just a very basic command-line interface (CLI) for your program can make everyone's life easier for modifying parameters, including programmers, but also non-programmers.

Comments in Python

Comments in Python are used to add explanatory or descriptive text to your code. They are ignored by the Python interpreter and do not affect the execution of the program. Comments help improve code readability and make it easier for others (or yourself) to understand the purpose or functionality of specific code sections.

There are two types of comments in python:

1. Single-Line Comments:
Single-line comments begin with the # symbol and continue until the end of the line. They are used to comment on a single line or provide brief explanations.

```
# This is a single-line comment
x = 5  # Assigning a value to x
```

2. Multi-Line Comments:
Multi-line comments, also known as docstrings, are used for longer comments that span multiple lines. They are enclosed between triple quotes (' ' ' or " " ") and can be used to describe functions, classes, or modules.

```
'''
This is a multi-line comment or docstring.
It can span multiple lines and is commonly used to provide
detailed explanations about code sections or documentation.
'''
```

Comments can also be used to temporarily disable or "comment out" sections of code that you don't want to execute. By commenting out code, you can easily enable or disable specific sections without having to delete or rewrite the code.

Remember, comments are a valuable tool for communicating and documenting your code, so use them wisely to make your code more understandable and maintainable.

Python Indentation

In Python, indentation plays a crucial role in determining the structure and execution of code blocks. Unlike many other programming languages that use braces or brackets, Python uses indentation to define blocks of code.

Important points about indentation:

1. **Consistent Indentation:** Python requires consistent indentation using spaces or tabs throughout the code. It is recommended to use four spaces for indentation.

2. **Code Blocks:** Indentation is used to define code blocks, such as loops, conditionals, and functions. The statements within a block are indented at the same level.

3. **Indentation Level:** The level of indentation determines the grouping of code. Blocks at the same level of indentation are considered part of the same code block, while a decrease in indentation indicates the end of a block.

4. **No Explicit Braces:** Python does not use braces ({}) or explicit keywords (e.g., "begin" and "end") to define code blocks. Instead, the indentation itself determines the scope and nesting of code.

5. **Syntax Error Prevention:** Indentation errors, such as inconsistent or missing indentation, will result in syntax errors and can cause the code to fail during execution.

Therefore, paying attention to proper indentation is essential in Python.

Example:

```
if condition:
   statement1
   statement2
   if nested_condition:
      nested_statement1
      nested_statement2
   statement3
statement4
```

Braces are not used in Python, instead Indentation is used. However, for clear understanding let us look at how will braces representation of above pseudo code will look like:

```
if condition{
   statement1
   statement2
   if nested_condition{
      nested_statement1
      nested_statement2
      }
   statement3
   }
statement4
```

Note: Proper indentation in Python is not just a matter of style; it is a fundamental part of the language's syntax. So, it's important to be mindful of indentation to ensure your code is both readable and functional.

Variables

Variables in Python are used to store and manipulate data. They act as containers that hold values of different types, such as numbers, strings, lists, or more complex data structures. Here's how you can work with variables in Python:

Note: Variables do not need to be declared with any particular type in Python.

- **Variable Assignment:**

You can assign a value to a variable using the assignment operator =. The variable is created if it doesn't exist or updated with the new value if it already exists.

```
# Assigning values to variables
x = 10
name = "Alice"
pi = 3.14
```

- **Variable Naming Rules:**

a. Variable names can contain letters (a-z, A-Z), digits (0-9), and underscores (_).
b. The first character of a variable name cannot be a digit.
c. Variable names are case-sensitive, so myVar and myvar are different variables.
d. It's good practice to use descriptive variable names that reflect their purpose.
e. Generally, we follow the convention of naming variables using snake_case, rather than lowerCamelCase.

- **Variable Types:**

Python is dynamically typed, meaning you don't need to specify the variable type explicitly. The type is inferred based on the assigned value. For example: In C++ or Java, we explicitly tell for each variable its data type, i.e. int x = 5; But in python we need not do that.

```
x = 10      # Integer
name = "Alice"  # String
pi = 3.14   # Float
is_student = True  # Boolean
```

- **Variable Reassignment:**

Variables can be reassigned with new values, even of different types.

```
x = 5
print(x)  # Output: 5

x = "Hello"
print(x)  # Output: Hello
```

- **Variable Concatenation:**

Variables of string type can be concatenated using the + operator.

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)  # Output: John Doe
```

- **Multiple Assignments:**

You can assign multiple variables simultaneously in a single line.

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

- **Variable Scope:**

The scope of a variable determines where it can be accessed within the code. Variables defined inside a function have local scope and can only be accessed within that function. Variables defined outside any function have global scope and can be accessed throughout the program.

```
# Global variable
global_var = "I'm a global variable"

def my_function():
    # Local variable
    local_var = "I'm a local variable"
```

```
    print(local_var)
    print(global_var)# Global variable accessible

my_function()
print(global_var)  # Global variable accessible
```

Keywords in Python

Keywords in Python are reserved words that have special meanings and predefined functionalities within the Python language. These keywords cannot be used as variable names or identifiers because they are already used to define the syntax and structure of the language.

Here are the keywords in Python:

```
and       as        assert    async     await
break     class     continue  def       del
elif      else      except    False     finally
for       from      global    if        import
in        is        lambda    None      nonlocal
not       or        pass      raise     return
True      try       while     with      yield
```

These keywords serve specific purposes in Python and play a vital role in constructing valid and meaningful code. For example, keywords like if, else, for, and while are used in control flow statements, while def is used to define functions. Other keywords, such as import, from, and as, are used for importing modules and managing namespaces.

Note: It's important to note that keywords are case-sensitive, so True, False, and None must be written in title case.
As a best practice, avoid using these keywords(with different casing) as variable names or identifiers to prevent conflicts and ensure code clarity and readability.

Question: Which of the following variable names are valid in Python and which ones are invalid?
1. my_var
2. 123abc
3. _count
4. first name
5. total#sales
6. True
7. user@name
8. var2
9. $price
10. function()

Answer:
Valid: my_var, _count, var2
Invalid: 123abc (starts with a number), first name (contains a space), total#sales (contains a special character), True (reserved keyword), user@name (contains special character "@"), $price (starts with a special character), function() (contains parentheses).

Print Statement

In Python, the print() function is used to display output on the screen. It allows you to print text, variables, and expressions. Here are some examples of how to use the print() function:

1. Printing Text

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

2. Printing Variables

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

Or we can also modify the print statement as:

```
print("My name is" + name + "and I am" + age + "years old.")
```

Both statements give the same output.

3. Printing Expressions

```
x = 10
y = 5
print("The sum of", x, "and", y, "is", x + y)
```

4. Printing Formatted Strings

This functionality is available only for versions of Python 3.6 and after.

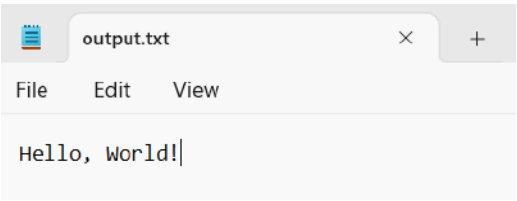
```
name = "Bob"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

5. Printing with Separator

```
x = 1
y = 2
z = 3
print(x, y, z, sep="-")  # Output: 1-2-3
```

6. Printing to a File

```
with open("output.txt", "w") as f:
    print("Hello, World!", file=f)
```



In the output.txt file, the content is printed.

Updation of Variables

Variables in Python can be updated by assigning new values to them using the assignment operator (=).

Example:

```
my_variable = 42
print("Initial Value:", my_variable)

my_variable = 23
print("Changed Value-1:", my_variable)

my_variable = "School"
print("Changed Value-2:", my_variable)
```

Data Types

In Python, data types represent the kind of values that can be stored and manipulated in variables. Python provides several built-in data types to handle different types of data. Here are some commonly used data types in Python:

a. Numeric Types:

- int: Represents integers (whole numbers) like 3, -7, 100, etc. (In Python 3, the int type has no max limit. You can handle values as large as the available memory allows. Python 2 has two integer types, int and long , while Python 3 only has int)
- float: Represents floating-point numbers (real numbers) with decimal points like 3.14, -0.5, 2.0, etc.
- complex: Represents complex numbers with real and imaginary parts, like 2 + 3j, -1 + 0.5j, etc.

b. Text Type:

- str: Represents strings of characters enclosed in single quotes (' ') or double quotes (" "). Strings are used to store textual data, such as names, sentences, or any sequence of characters. For example: 'sentence', "mytext", "abcd", etc.

c. Boolean Type:

- bool: Represents boolean values, which can be either True or False. Booleans are often used in conditional statements and logical operations.

d. Sequence Types:

- list: Represents an ordered collection of items enclosed in square brackets ([]). Lists can contain elements of different types and can be modified (mutable). For example: thislist = ["apple", "banana", "cherry"], mylist = [1,2,3,4].
- tuple: Similar to lists, but enclosed in parentheses (()). Tuples are immutable, meaning they cannot be modified after creation and allow duplicate values. For example: thistuple = ("apple", "banana", "cherry", "apple", "cherry"), mytuple = (1, 2, 2, 3).

e. Mapping Type:

- dict: Represents a collection of key-value pairs enclosed in curly braces ({}) or created using the dict() constructor. Dictionaries allow you to store and retrieve values based on unique keys.

For example:
```
student = {
  "first_name": "Ally",
  "last_name": "Joy",
  "rollno": 96378
}
```

f. Set Types:

- set: Represents an unordered collection of unique elements enclosed in curly braces ({}) or created using the set() constructor. Sets are useful for performing mathematical set operations, such as union, intersection, etc. For example: thisset = {"apple", "banana", "cherry"}

e. Other Types:

- None: Represents a special value indicating the absence of a value or a null value.

These are the fundamental data types in Python. However, Python is a dynamic language, allowing you to create and work with more complex data structures and custom types using classes and objects. Additionally, Python also supports type conversion and type-checking mechanisms to handle different data types effectively during the runtime.

ASCII Values

ASCII (American Standard Code for Information Interchange) is a widely used character encoding standard that provides a way to represent characters as numeric codes. It was developed in the 1960s and became a standard in the computing industry.

ASCII uses 7 bits to represent a total of 128 characters, including uppercase and lowercase letters, digits, punctuation marks, control characters, and some special symbols. Each character is assigned a unique numeric code from 0 to 127.

Here are some key ASCII values:

- A to Z: ASCII values 65 to 90 represent uppercase letters.
- a to z: ASCII values 97 to 122 represent lowercase letters.
- 0 to 9: ASCII values 48 to 57 represent digits.
- Space: ASCII value 32 represents a space character.
- Special Characters: Various special characters like punctuation marks, symbols, and control characters have their own ASCII values, such as:
  - ! (Exclamation Mark): ASCII value 33
  - @ (At Symbol): ASCII value 64
  - $ (Dollar Sign): ASCII value 36
  - % (Percent Sign): ASCII value 37
  - & (Ampersand): ASCII value 38
  - * (Asterisk): ASCII value 42
  - ...and many more.

ASCII was widely used in older systems and programming languages. However, its limitations in representing diverse languages and scripts led to the development of Unicode, a more comprehensive encoding standard capable of representing a broader range of characters from various writing systems.

Obtain ASCII and Unicode in Python

In Python, you can obtain the Unicode code of a special character by using the built-in ord() function and the ASCII value of characters. The ord() function accepts a string of unit length as an argument and returns the Unicode equivalence of the passed argument.

Example:

```
string = "€"
unicode_code = ord(string)
print("Unicode of", string, "=", unicode_code)
```

Output:

```
Unicode of €=8364
```

Conversely, Python chr() function is used to get a string representing of a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds from the specified range. The standard range of the argument is from 0 to 1,114,111.

Taking Input

In Python, you can take user input using the input() function. It allows you to prompt the user for input and retrieve the entered value as a string. Here's how you can use the input() function:

```
name = input("Enter your name: ")
```

In the above example, the input() function is used to prompt the user with the message "Enter your name: ". The program waits for the user to enter input, and once the user presses Enter, the entered value is assigned to the variable name.

Note that the input is always captured as a string.

If you want to convert the user input to a specific data type, such as integer or float, you can use type casting. Type Casting is the method to convert a Python variable datatype into a certain data type in order to perform the operation required by users.
To know the type of data stored in a variable, we can use the type() function.
For example, to capture an integer input, you can use the int() function:

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

In this case, the input is first captured as a string using input(), and then it is converted to an integer using int().

Taking user input using input() allows your Python programs to interact with users and make them more interactive and dynamic.
Consider this example:

Code:

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

print("\nHello,", name + "! You are", age, "years old.")
```

Question:
Write a program to find the sum of two numbers entered by the user.

Solution: https://pastebin.com/t9cwVV5r

Operators in Python

These are symbols or special characters that perform various operations on operands (variables, values, or expressions). Python supports a wide range of operators, including arithmetic operators, assignment operators, comparison operators, logical operators, and more. The different types of operators in Python are:

1. Arithmetic Operators:

1. + (Addition): Adds two operands.
2. - (Subtraction): Subtracts the right operand from the left operand.
3. * (Multiplication): Multiplies two operands.
4. / (Division): Divides the left operand by the right operand (returns a float).
5. // (Floor Division): Divides the left operand by the right operand and rounds down to the nearest whole number.
6. % (Modulus): Returns the remainder of the division between the left operand and the right operand.
7. ** (Exponentiation): Raises the left operand to the power of the right operand.

Example code: https://pastebin.com/rnCpV7GV

2. Assignment Operators:

a. = (Assignment): Assigns the value on the right to the variable on the left.
b. +=, -=, *=, /=, //=, %=: Perform the corresponding operation and assign the result to the left operand.

Example code: https://pastebin.com/8SFP8fvJ

3. Comparison Operators:

a. = (Equal to): Checks if the operands are equal.
b. != (Not equal to): Checks if the operands are not equal.
c. < (Less than), > (Greater than): Checks if the left operand is less than or greater than the right operand, respectively.
d. <= (Less than or equal to), >= (Greater than or equal to): Checks if the left operand is less than or equal to, or greater than or equal to the right operand, respectively.

Example code: https://pastebin.com/cmpBcLrQ

4. Logical Operators:

a. and: Returns True if both operands are True.
b. or: Returns True if at least one of the operands is True.
c. not: Returns the opposite boolean value of the operand.

Example code: https://pastebin.com/N81SMQdS

5. Bitwise Operators:

a. & (Bitwise AND): Performs a bitwise AND operation between the binary representation of the operands.
b. | (Bitwise OR): Performs a bitwise OR operation between the binary representation of the operands.
c. ^ (Bitwise XOR): Performs a bitwise XOR (exclusive OR) operation between the binary representation of the operands.
d. << (Left Shift): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
e. >> (Right Shift): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

Example code: https://pastebin.com/D4PLnZgK

6. Membership Operators:

a. in: Returns True if a value is found in a sequence.
b. not in: Returns True if a value is not found in a sequence.

Example code: https://pastebin.com/rd0cbA8C

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

a .is: Returns True if two operands refer to the same object.
b. is not: Returns True if two operands do not refer to the same object.

Example code: https://pastebin.com/96FnmbsG

Example: Volume of Sphere
Take the radius from the user and display the volume of sphere.

Code: https://pastebin.com/rA798mzq

Example: Calculate Simple Interest
Take the principal, rate and time from the user and display the simple interest.

Code: https://pastebin.com/3wRK7A7Q

Operator Precedence

Operator precedence in Python determines the order in which operators are evaluated in an expression. Here is the precedence of operators in Python, listed in descending order from highest to lowest:

1. Parentheses: ()
2. Exponentiation: **
3. Multiplication, Division, Floor Division, Modulus: *, /, //, %
4. Addition and Subtraction: +, -
5. Bitwise Shifts: <<, >>
6. Bitwise AND: &
7. Bitwise XOR: ^

8. Bitwise OR: |
9. Comparison Operators: ==, !=, >, <, >=, <=, is, is not, in, not in
10. Logical NOT: not
11. Logical AND: and
12. Logical OR: or

It is important to note that parentheses can be used to override the default precedence and explicitly define the order of evaluation.

Example:

```
result = 3 + 2 ** 4 / 2 * 5 - 8 // 2
print(result)
```

Output:
39.0

Let's break down the expression according to operator precedence:

1. Exponentiation (**): 2 ** 4 evaluates to 16.
2. Division (/): 16 / 2 evaluates to 8.0.
3. Multiplication (*): 8.0 * 5 evaluates to 40.0.
4. Floor Division (//): 8 // 2 evaluates to 4.
5. Addition (+): 3 + 40.0 evaluates to 43.0.
6. Subtraction (-): 43.0 - 4 evaluates to 39.0.

Therefore, the final result assigned to the variable result is 39.0.

Typecasting

Typecasting, also known as type conversion, is the process of changing the data type of a variable from one type to another. In Python, you can perform typecasting using built-in functions or constructor functions for the desired data type.

type()

- Python type() is a built-in function that is used to return the type of data stored in the objects or variables in the program.
- For example, if a variable contains a value of 45.5 then the type of that variable is float.
- If variable 'subj' stores the value 'Python', then the type of 'subj' is a string.

Here are some common type casting functions in Python:

1. int(): Converts a value to an integer data type.
2. float(): Converts a value to a floating-point data type.
3. str(): Converts a value to a string data type.
4. bool(): Converts a value to a boolean data type.
5. list(): Converts a value to a list data type.
6. tuple(): Converts a value to a tuple data type.
7. set(): Converts a value to a set data type.
8. dict(): Converts a value to a dictionary data type.

Example of Typecasting:

A. Integer to String

```
num = 42
num_str = str(num)
print(num_str, type(num_str))
```

B. String to Integer

```
num_str = "123"
num = int(num_str)
print(num, type(num))
```

C. Float to Integer

```
float_num = 3.14
integer_num = int(float_num)
print(integer_num, type(integer_num))
```

Note: it is important to ensure compatibility and proper handling of data in different contexts.
Eg: You can't typecast string "apple" to int.

Question: Write a program to convert temperature from Celsius to Fahrenheit.

Solution: https://pastebin.com/YcPYnvtf

MCQs

1. What will be the output of the following code snippet?

```
x = 10
y = 5
print(x > y and x < 15)
```

(a) True
(b) False
(c) Error
(d) None

2. What will be the output of the following code snippet?

```
x = 5
y = "2"
print(x + y)
```

(a) 7
(b) 52
(c) TypeError
(d) "52"

3. What is the result of the following expression in Python?

```
8 // 3 + 4 % 2
```

(a) 3
(b) 3.67
(c) 3.5
(d) 2

4. What will be the value of d if d is a float after the operation d = 2 / 7?
(a) 0
(b) 0.2857
(c) Cannot be detemined
(d) None of the above

Answers:
4. (a) True
5. (c) TypeError
6. (d) 2
7. (b) 0.2857

Chapter-2

Control Statements

Control statements are structures in programming languages that enable you to control the flow of execution in your program. They allow you to make decisions, repeat certain actions, and handle exceptional cases.
In Python, there are three main types of control statements:

1. Conditional Statements: Enable execution of specific code blocks based on the evaluation of conditions. These are:
- if
- elif
- else

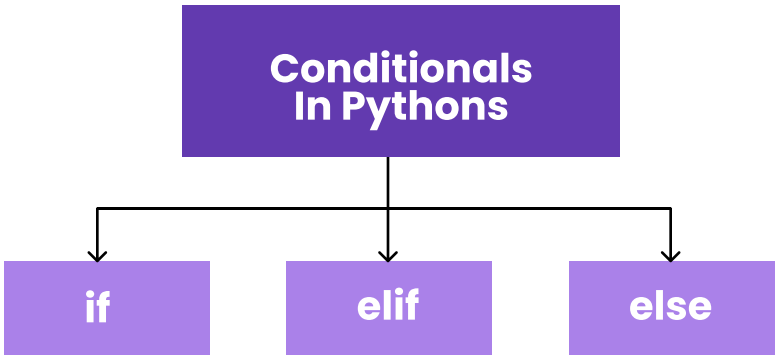
2. Looping Statements: Allow repetition of code blocks either for a specified number of times or until a condition is met. There are two loops in python:
- for
- while

3. Control Flow Statements: Provide control over the flow of execution, such as terminating loops or skipping iterations based on certain conditions. These are:
- break
- continue
- pass

Conditionals

Conditionals, also known as conditional statements or conditional expressions, are structures in programming languages that allow the execution of different code blocks based on specified conditions. They provide the ability to make decisions and control the flow of a program.

There are 3 conditionals in python:



In Python, conditionals are typically implemented using the following statements:

- **if:** The if statement allows you to execute a block of code if a condition is true.
- **elif:** The elif statement is short for "else if" and is used to check additional conditions if the condition for if statement is False. It is optional and can appear multiple times within an if statement.
- **else:** The else statement is used to define a block of code that should be executed if none of the preceding conditions are true.

Conditionals evaluate a condition or expression that results in a Boolean value (True or False). Based on the evaluation of the condition, the program decides which code block(s) to execute.

if else condition

Example: Take an integer input and tell if it is positive or negative (zero not included)

Code: https://pastebin.com/D2PyHGb4

Example: Take a positive integer input and tell if it is even or odd

Code: https://pastebin.com/XU1YUhdT

Example: If cost price and selling price of an item is input through the keyboard, write a program to determine whether the seller has made profit or incurred loss or no profit no loss. Also determine how much profit he made or loss he incurred.

Code: https://pastebin.com/CYvA29Ka

Multiple Conditions Using 'and' and 'or'

In Python, the logical operators "and" and "or" are used for combining multiple conditions.

Using "and" operator:

```
if condition1 and condition2:
    # Code to execute when both condition1 and condition2 are true
```

Using "or" operator:

```
if condition1 or condition2:
    # Code to execute when either condition1 or condition2 is true (or both)
```

Example:

```
x = 5
y = 10

if x > 0 and y < 15:
    print("Both conditions are true.")

if x > 10 or y < 15:
    print("At least one condition is true.")
```

In this example, the first "if" statement will execute the code block since both conditions x > 0 and y < 15 are true. The second "if" statement will also execute the code block because at least one of the conditions x > 10 or y < 15 is true.

Question: Take positive integer input and tell if it is a four digit number or not.
Code: https://pastebin.com/ey65ydhP

Question: Take positive integer input and tell if it is divisible by 5 and 3.
Code: https://pastebin.com/2vPKxZXW

Question: Take positive integer input and tell if it is divisible by 5 or 3.
Code: https://pastebin.com/eNAtPUh4

Question: Take 3 positive integers input and print the greatest of them.
Code: https://pastebin.com/KxCiyWZd

Question: Determine whether the year is a leap year or not.
Code: https://pastebin.com/ztKvKnSP

Nested if else condition

The nested if-else syntax allows you to have an if-else statement inside another if or else block. This allows for more complex branching and decision-making in your code.

Syntax:

```
if condition1:
    # Code block executed when condition1 is true
    if condition2:
        # Code block executed when both condition1 and condition2 are true
    else:
        # Code block executed when condition1 is true and condition2 is false
else:
    # Code block executed when condition1 is false
```

Question: Take positive integer input and tell if it is divisible by 5 or 3 but not divisible by 15.
Code: https://pastebin.com/YXpsTA2B

Question: Take 3 positive integers input and print the greatest of them (without using multiple condition)
Code: https://pastebin.com/H5S4rXV2

Elif Condition

The elif statement in Python is short for "else if" and is used to introduce an additional condition to be checked if the preceding if or elif conditions are false. It allows for multiple conditional branches in your code.

Syntax:

```
if condition1:
    # Code block executed when condition1 is true
elif condition2:
    # Code block executed when condition1 is false and condition2 is true
else:
    # Code block executed when both condition1 and condition2 are false
```

Question: Take input percentage of a student and print the Grade according to marks:
1) 81-100 Very Good
2) 61-80 Good
3) 41-60 Average
4) <=40 Fail

Code:
https://pastebin.com/YLBegzRB **(Using Multiple Condition)**
https://pastebin.com/jJeGSpc3 **(Using elif Condition)**

Match Case

Python does not have a built-in switch statement like some other programming languages (e.g., C/C++, Java). Instead, Python relies on other control flow structures to handle multiple conditions and make decisions based on different cases.

However, Python 3.10 introduced the match statement as a form of pattern matching. The match statement is similar to the traditional switch statement. So we can implement switch statement using the match and case Keywords in Python 3.10.

Syntax: https://pastebin.com/xNKpXVDm
Example: https://pastebin.com/LwZv0ZxP

Ternary Operator

The ternary operator in Python is a concise way to express conditional expressions. It allows you to assign a value to a variable based on a condition in a single line of code.

Syntax:

```
variable = value1 if condition else value2
```

In this syntax, condition is the condition to be evaluated. If the condition is true, the value assigned to variable is value1. If the condition is false, the value assigned to variable is value2.

Question: Write a program to check if number is odd or even using ternary operator.
Code: https://pastebin.com/uPdJm7fk

Loops

Loops in programming allow you to repeatedly execute a block of code based on a specific condition. They are used to automate repetitive tasks and iterate over collections of data.

In Python, there are two types of loops: for loop and while loop.

for loop

- A for loop is used to iterate over a sequence (such as a list, tuple, or range) or any iterable object.
- It executes a block of code for each item in the sequence.
- The loop variable takes on the value of each item in the sequence one by one.
- The loop continues until all items in the sequence have been processed.

Syntax:

```
for item in sequence:
    # Code block to execute
```

Example:

```
for i in [1,2,3,4]:
    print(i)
```

Question: Print hello world 'n' times. Take 'n' as input from user.
Code: https://pastebin.com/Bp50vKm3

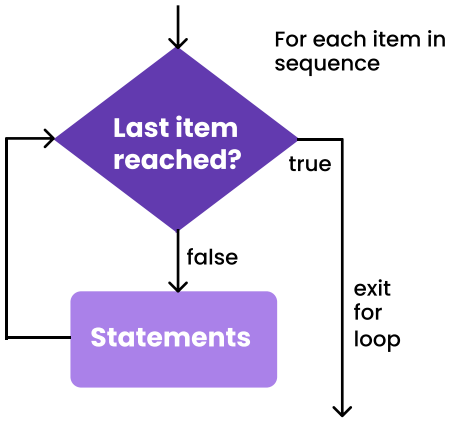
How for loop works?

1. Initialization: The loop starts by initializing a loop variable. This variable can be any valid variable name and is used to represent each item in the sequence. In each iteration, the loop variable takes on the value of the next item in the sequence.

2. Iteration: The loop iterates over the sequence and executes the code block for each item in the sequence. The loop variable takes on the value of each item, one by one, as the loop progresses through the sequence.

3. Execution: The code block within the loop is executed for each iteration. You can include any statements or operations within the code block that you want to perform for each item in the sequence.

4. Termination: The loop continues iterating until it reaches the end of the sequence. Once all items in the sequence have been processed, the loop terminates, and the program proceeds to the next line of code after the loop.



Question: Print numbers from 1 to 100
Code: https://pastebin.com/yUZahTNj

In this program, the range(start, stop) function is used to generate a sequence of numbers from 1 (inclusive) to 101 (exclusive). The loop variable num takes on the value of each number in the sequence, starting from 1 and ending at 100.

Question: Print even numbers from 1 to 100
Code: https://pastebin.com/z4w1q8wr

In this program, the range(start, stop, step) function is used to generate a sequence of numbers starting from 2 (inclusive) and ending at 101 (exclusive), with a step size of 2. This ensures that only even numbers are included in the sequence.

Question: Display this AP - 1,3,5,7,9.. upto 'n' terms.

Code: https://pastebin.com/FP6zpL5d
Code (using steps): https://pastebin.com/NaKgmQTh

Formula for nth term = a + (n-1)d
First term = a
Second term = a + d
Third term = a + 2d = (a + d) + d = Second term + d

Break Statement

The break statement is a control flow statement in Python that allows you to exit or terminate a loop prematurely. It is commonly used to stop the execution of a loop before it reaches its natural end. When encountered, the break statement immediately terminates the innermost loop and resumes execution at the next statement following the loop.

Example:

```
for num in range(1, 11):
    if num == 6:
        break
    print(num)
```

Once the loop encounters the break statement when num is 6, the loop is immediately terminated, and the numbers 6, 7, 8, 9, and 10 are not printed.

The break statement is particularly useful when you need to exit a loop prematurely based on certain conditions or when you have found the desired result. It helps control the flow of your program and allows you to efficiently manage looping behavior.

Continue Statement

The continue statement is a control flow statement in Python that is used to skip the rest of the current iteration of a loop and move on to the next iteration. When encountered, the continue statement immediately stops the execution of the current iteration and goes back to the loop's beginning to start the next iteration.

Example:

```
for num in range(1, 11):
    if num % 2 == 0:
        continue
    print(num)
```

In this example, only the odd numbers are printed. Because, whenever the loop encounters an even number, the continue statement is executed, skipping the print(num) statement and proceeding to the next iteration.

The continue statement is useful when you want to skip certain iterations of a loop based on specific conditions. It allows you to control the flow of your program within the loop and selectively execute the desired statements.

Pass Statement

The pass statement in Python is a placeholder statement that does nothing. It is used when a statement is required syntactically but you don't want to execute any code or perform any action. The pass statement is typically used as a placeholder to indicate that the code will be added later.

Example:

```
for num in range(1, 6):
    if num == 3:
        pass
    else:
        print(num)
```

As you can see, when num is 3, the loop iteration doesn't print anything due to the pass statement. For all other values of num, the print(num) statement is executed as usual.

The pass statement is useful in situations where you want to create a placeholder for future code, handle specific conditions separately in the future, or simply maintain the syntactic correctness of the code without any immediate action.

Question: Find the highest factor of given number 'n'(except n).
Code: https://pastebin.com/9JFyXAaP

while loop

- A while loop is used to repeatedly execute a block of code as long as a specific condition is true.
- It evaluates the condition before each iteration and continues until the condition becomes false.
- It is useful when the number of iterations is not known beforehand or depends on runtime conditions.

Syntax:

```
while condition:
    # Code block to execute
```

Question: Print numbers from 1 to 100
Code: https://pastebin.com/cqXN9BXq

Question: Print even numbers from 1 to 100
Code: https://pastebin.com/x5szMgbj

Predict the output

Question-1:

```
j = 0
while j <= 10:
    print(j)
    j = j + 1
```

Answer:

0
1
2
3
4
5
6
7
8
9
10

Question-2:

```
x = 1
while x == 1:
    x = x - 1
    print(x)
```

Answer:

0

Question-3

```
i = 10
while i == 20:
    print("computer buff!")
```

Answer:

The output of the given code will be nothing (empty output).

Question-4

```
x = 4
y = 0
while x >= 0:
    x -= 1
    y += 1
```

```
if x == y:
    continue
 else:
    print(x, y)
```

Answer:
3 1
1 3
0 4
-1 5

Pattern Printing Questions

Question-1: Print the given pattern
For n = 1
*****
For n = 2
*****
*****
For n = 3
*****
*****
*****
Print five stars in each row where no. of rows = n
Code: https://pastebin.com/VDSzDSrZ

Question-2: Print the given pattern

For n = 4	for n=6
1234	123456
1234	123456
1234	123456
1234	123456
	123456
	123456

Code: https://pastebin.com/MUyf5ygU

Question-3: Print the given pattern
For n = 4
*
**
***
****
Code: https://pastebin.com/iinDpDrP

Question-4: Print the given pattern

For n = 4

```
 1
 1 2
 1 2 3
 1 2 3 4
```

Code: https://pastebin.com/qbz6VTCm

Question-5: Print the given pattern

```
      1
    1 2 3
  1 2 3 4 5
1 2 3 4 5 6 7
```

Code: https://pastebin.com/WKm4aDKG

Chapter-3

Python Arrays

1. Python does not have built-in support for Arrays
2. Datatypes from Python Collections can be used instead.
3. However, to work with arrays in Python you will have to import a library, like the NumPy library.

Python Collections(Arrays)

This collection in python contains 4 built in datatypes:

1. Lists
2. Tuple
3. Set
4. Dictionary

Lists

Lists are used to store multiple items in a single variable.
Lists are created using square brackets.

Syntax:

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

List Items:

1. **Indexed:** List items are indexed. The first item has index [0], the second item has index [1] etc.
2. **Ordered:** the items have a defined order, and that order will not change. If you add new items to a list, the new items will be placed at the end of the list.
3. **Mutable:** The list is mutable, meaning that we can update, add, and remove items in a list after it has been created.
4. **Duplicates:** Lists can have items with the same value.
5. **Any datatype:** All the following lists are correct.
   list1 = ["apple", "banana", "cherry"]
   list2 = [1, 5, 7, 9, 3]
   list3 = [True, False, False]

6. Mix of different data types: This list is also valid
list1 = ["abc", 34, True, 40, "male"]

Type of a list

- Since a list is a built in datatype in python, its type is List.
- This can be checked by using the type() function.
- For example, consider this code snippet:
  mylist = ["apple", "banana", "cherry"]
  print(type(mylist))

Length of a list

- To determine how many items a list has, we use the len() function.
- For example, consider this code snippet:
  my_list = ["apple", "banana", "cherry"]
  print(len(my_list))

The list() constructor

- It is also possible to use the list() constructor when creating a new list.
- For example: this_list = list(("apple", "banana", "cherry"))

Taking input for a list:

Code:

```
this_list = []
# number of elements as input
n = int(input("Enter number of elements : "))

# iterating till the range
for i in range(0, n):
    element = int(input())
    # adding the element
    this_list.append(element)

print(this_list)
```

Checking if item exists in a list:
To determine if a specified item is present in a list use the in keyword.

Accessing Items

List items are indexed and you can access them by referring to the index number in 4 ways:

1. Indexing: any index from the range 0 to length of list-1.

```
8  mylist = ["apple", "banana", "cherry"]
9  print(mylist[1])
10
```

```
banana

...Program finished with exit code 0
Press ENTER to exit console.
```

2. Negative Indexing: Negative indexing means starting from the end; -1 refers to the last item, -2 refers to the second last item etc.

```
mylist = ["apple", "banana", "cherry"]
print(mylist[-2])
```

```
banana

...Program finished with exit code 0
Press ENTER to exit console.
```

3. Range of Indexes: You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items. Herein, the start of the range is inclusive and the end of the range is exclusive, i.e., elements are considered from start index till end index-1.

```
mylist = ["apple", "banana", "cherry"]
print(mylist[1:3])
```

```
['banana', 'cherry']

...Program finished with exit code 0
Press ENTER to exit console.
```

4. Range of Negative Indexes: Specify negative indexes if you want to start the search from the end of the list. Herein, the start of the range is inclusive and the end of the range is exclusive, i.e., elements are considered from start index till end index-1.

```
mylist = ["apple", "banana", "cherry"]
print(mylist[-3: -1])
```

```
['apple', 'banana']

...Program finished with exit code 0
Press ENTER to exit console.
```

Adding items to a list

To add an item to the end of the list, we have 3 methods.

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

```
mylist = ["apple", "banana", "cherry"]
mylist.append("orange")
print(mylist)
```

```
['apple', 'banana', 'cherry', 'orange']

...Program finished with exit code 0
Press ENTER to exit console.
```

2. insert(): To insert a list item at a specified index, we use the insert() method.

```
mylist = ["apple", "banana", "cherry"]
mylist.insert(1, "orange")
print(mylist)
```

```
['apple', 'orange', 'banana', 'cherry']

...Program finished with exit code 0
Press ENTER to exit console.
```

3. extend(): To append elements from another list or any other data type to the current list, use the extend() method.

```
mylist = ["apple", "banana", "cherry"]
thislist = ["mango", "pineapple", "papaya"]
mylist.extend(thislist)
print(mylist)
```

```
['apple', 'banana', 'cherry', 'mango', 'pineapple',

...Program finished with exit code 0
Press ENTER to exit console.
```

Removing elements from a list

There are 2 methods to remove an item from the list.

1. remove(): The remove() method removes the specified item.

```
mylist = ["apple", "banana", "cherry"]
mylist.remove("cherry")
print(mylist)
```

```
['apple', 'banana']

...Program finished with exit code 0
Press ENTER to exit console.
```

2. pop(): The pop() method removes the specified index. If you do not specify the index, the pop() method removes the last item.

```
mylist = ["apple", "banana", "cherry"]
mylist.pop(1)
print(mylist)
```

```
['apple', 'cherry']

...Program finished with exit code 0
Press ENTER to exit console.
```

Changing item in a list

1. At an index:

```
mylist = ["apple", "banana", "cherry"]
mylist[1] = "orange"
print(mylist)
```

```
['apple', 'orange', 'cherry']

...Program finished with exit code 0
Press ENTER to exit console.
```

2. In a range:

```
mylist = ["apple", "banana", "cherry"]
mylist[1: 3] = ["orange", "blueberries"]
print(mylist)
```

```
['apple', 'orange', 'blueberries']

...Program finished with exit code 0
Press ENTER to exit console.
```

Sorting List

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

```
mylist = ["apple", "orange", "cherry"]
mylist.sort()
print(mylist)
```

```
['apple', 'cherry', 'orange']

...Program finished with exit code 0
Press ENTER to exit console.
```

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

```
mylist = ["apple", "orange", "cherry"]
mylist.sort(reverse = True)
print(mylist)
```

```
['orange', 'cherry', 'apple']

...Program finished with exit code 0
Press ENTER to exit console.
```

Reversing a list

To reverse the order of a list, we use the reverse() method.

```
mylist = ["apple", "orange", "cherry"]
mylist.reverse()
print(mylist)
```

```
['cherry', 'orange', 'apple']

...Program finished with exit code 0
Press ENTER to exit console.
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

For Example: Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name. Without list comprehension you will have to write a for statement with a conditional test inside but with list comprehension you can do all that with only one line of code.

```
mylist = ["apple", "orange", "cherry"]
newlist = [x for x in mylist if "a" in x]

print(newlist)
```

```
['apple', 'orange']

...Program finished with exit code 0
Press ENTER to exit console.
```

Copying a list

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

We use the built-in List method copy().

```
mylist = ["apple", "orange", "cherry"]
newlist = mylist.copy()

print(newlist)
```

```
['apple', 'orange', 'cherry']

...Program finished with exit code 0
Press ENTER to exit console.
```

Concatenation of lists

To join 2 or more lists, simply use the + operator.

```
mylist = ["apple", "banana", "cherry"]
newlist = ["orange", "blueberries"]

print(mylist + newlist)
```

```
['apple', 'banana', 'cherry', 'orange', 'blueberries']

...Program finished with exit code 0
Press ENTER to exit console.
```

Similarly, extend() method can also be used to add the elements of any iterable to the end of the list.

Nested Lists

A nested list is a list of lists.

```
nestedList = [1, 2, ['a', 1], 3]

# indexing List: the sublist has now been accessed
subList = nestedList[2]

# access the first element inside the inner list:
element = nestedList[2][0]

print("List inside the nested list: ", subList)
print("First element of the sublist: ", element)
```

```
List inside the nested list:  ['a', 1]
First element of the sublist:  a

...Program finished with exit code 0
Press ENTER to exit console.
```

Q1. Given a list in Python and provided the index of the elements, write a program to swap the two elements in the list.

Examples:

Input : List = [23, 65, 19, 90], index1 = 0, index2 = 2Output : [19, 65, 23, 90]

Input : List = [1, 2, 3, 4, 5], index1 = 1, index2 = 4Output : [1, 5, 3, 4, 2]

Code:

```
def swapIndices(lis, index1, index2):
    temp=lis[index1]
    lis[index1]=lis[index2]
    lis[index2]=temp
    return lis
# Driver function
List = [23, 65, 19, 90]
index1, index2 = 0, 2
```

print(swapIndices(List, index1, index2))

Q2. Given a list of integers with duplicate elements in it. The task is to generate another list, which contains only the duplicate elements. In simple words, the new list should contain elements that appear more than one.

Examples:

Input : list = [10, 20, 30, 20, 20, 30, 40, 50, -20, 60, 60, -20, -20]
Output : output_list = [20, 30, -20, 60]
Input : list = [-1, 1, -1, 8]
Output : output_list = [-1]

Code:

```
list = [1, 2, 1, 2, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 9]
new = []

for a in list:
 n = list.count(a)
 if n > 1:
  if new.count(a) == 0: # condition to check
   new.append(a)
```

print(new)

TUPLES
Tuples are used to store multiple items in a single variable.
Tuples are written with round brackets.
Syntax:

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

Tuple items

- **Ordered:** When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- **Immutable:** Tuples are unchangeable, meaning that we cannot update, add or remove items after the tuple has been created.
- **Duplicates allowed:** Tuple can have items with the same value. For example: this_tuple = ("apple", "banana", "cherry", "apple", "cherry") is a valid tuple.
- **Any datatype:** A tuple can have can type of data. For example: all the 3 following tuples are valid.

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

- Mix of different data types: A tuple can contain different data types. For example: tuple = ("abc", 34, True, 40, "male") is a valid tuple.

Creating a tuple with 1 item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.
For example, this_tuple = ("apple",)

Length of a tuple

To determine how many items a tuple has, use the len() function.

```
mytuple = ("apple", "banana", "cherry")
print(len(mytuple))
```

```
3

...Program finished with exit code 0
Press ENTER to exit console.
```

Type of a tuple

A tuple is a built in datatype in python, hence it is of the 'tuple' datatype. This can be confirmed by using the type() method.

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

```
<class 'tuple'>

...Program finished with exit code 0
Press ENTER to exit console.
```

tuple() constructor

It is also possible to use the tuple() constructor to make a tuple.

```
mytuple = tuple(("apple", "banana", "cherry"))
print(mytuple)
```

```
('apple', 'banana', 'cherry')

...Program finished with exit code 0
Press ENTER to exit console.
```

Accessing items of a tuple

There are 4 different ways of accessing items in a tuple.

1. Positive Index: You can access tuple items by referring to the index number(0 to length of tuple-1), inside square brackets.

```
mytuple = ("apple", "banana", "cherry")
print(mytuple[1])
```

```
banana

...Program finished with exit code 0
Press ENTER to exit console.
```

2. Negative Indexing: Negative indexing means start from the end. -1 refers to the last item, -2 refers to the second last item etc.

```
mytuple = ("apple", "banana", "cherry")
print(mytuple[-1])
```

```
cherry

...Program finished with exit code 0
Press ENTER to exit console.
```

3. Range of index: You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items. Herein, the start of the range is inclusive and the end of the range is exclusive, i.e., elements are considered from start index till end index-1.

```
mytuple = ("apple", "banana", "cherry")
print(mytuple[1:3])

('banana', 'cherry')

...Program finished with exit code 0
Press ENTER to exit console.
```

4. Range of negative indexes: Specify negative indexes if you want to start the search from the end of the tuple. Herein, the start of the range is inclusive and the end of the range is exclusive, i.e., elements are considered from start index till end index-1.

```
mytuple = ("apple", "banana", "cherry")
print(mytuple[-2:-1])

('banana',)

...Program finished with exit code 0
Press ENTER to exit console.
```

Check if item exists in a tuple

To determine if a specified item is present in a tuple, we use the in keyword.

```
mytuple = ("apple", "banana", "cherry")
if "apple" in mytuple:
  print("Yes, 'apple' is in the tuple")

Yes, 'apple' is in the tuple

...Program finished with exit code 0
Press ENTER to exit console.
```

Unpacking a Tuple

When we create a tuple, we normally assign values to it. This is called "packing" a tuple. But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking".

Note: We will get an error if the number of variables used for unpacking does not match the number of items in the tuples.

```
mytuple = ("apple", "banana", "cherry")
(green, yellow, red) = mytuple

print(green)
print(yellow)
print(red)

apple
banana
cherry

...Program finished with exit code 0
Press ENTER to exit console.
```

Traversing through a tuple

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)
```

```
apple
banana
cherry

...Program finished with exit code 0
Press ENTER to exit console.
```

Concatenation of tuples

To join two or more tuples, you can use the + operator.

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = ("orange", "blueberries")
print(tuple1 + tuple2)
```

```
('apple', 'banana', 'cherry', 'orange', 'bluebe

...Program finished with exit code 0
Press ENTER to exit console.
```

Note: Tuple does not support functions like append(), insert() and extend() as tuples are immutable.

Tuples vs Lists

- Iterating through a 'tuple' is faster than in a 'list'.
- 'Lists' are mutable whereas 'tuples' are immutable.
- Tuples that contain immutable elements can be used as a key for a dictionary.

Q1. Reverse a tuple.

Input : tuples = ('z','a','d','f','g','e','e','k')
Output : ('k', 'e', 'e', 'g', 'f', 'd', 'a', 'z')

Input : tuples = (10, 11, 12, 13, 14, 15)
Output : (15, 14, 13, 12, 11, 10)

Approach: Using the reversed() built-in function, reverse the elements of the tuple and put it in a list. Once the list has all the elements in reverse order, typecast it into a tuple.

Code:

```
def Reverse(tuples):
    list = []
    for k in reversed(tuples):
        list = list + [k]
    tuple(list)
    print (list)

tuples = (10, 11, 12, 13, 14, 15)
Reverse(tuples)
```

```
[15, 14, 13, 12, 11, 10]

...Program finished with exit code 0
Press ENTER to exit console.
```

Sets

Sets are used to store multiple items in a single variable.
Sets are written with curly brackets.

Syntax:
this_set = {"apple", "banana", "cherry"}

Set Items

- **Unordered:** Unordered means that the items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- **Immutable:** Set items are unchangeable, meaning that we cannot update the items after the set has been created. Addition and removal is possible.
- **Unindexed:** Sets are unordered, so you cannot be sure in which order the items will appear.
- **Duplicates not allowed:** Sets cannot have two items with the same value. If you put duplicates in the set, they get ignored. Note: 1 and true are considered same values in sets and hence are considered as duplicates id entered in the same set.
- **Any datatype:** A set can have items of any data type. All the 3 following sets are valid.

set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False}

- Mix of different data types: A set can have various data types. For example: set1 = {"abc", 34, True, 40, "male"}.

Length of a set

To determine how many items a set has, use the len() function.

```
8  thisset = {"apple", "banana", "cherry"}
9  print(len(thisset))
10
11
12
```

```
3

...Program finished with exit code 0
Press ENTER to exit console.
```

In the below example, see how both the sets have the same length although the first set has 1 extra duplicate element.

```
9  set1 = {True, False, False}
10 set2 = {True, False}
11
12 print(len(set1))
13 print(len(set2))
14
```

```
2
2

...Program finished with exit code 0
Press ENTER to exit console.
```

Type of set

Set is a built-in datatype in python, hence its of set datatype. This can be verified by using the type() method.

```
thisset = {"apple", "banana", "cherry"}
print(type(thisset))
```

```
<class 'set'>


...Program finished with exit code 0
Press ENTER to exit console.
```

The set() Constructor

It is also possible to use the set() constructor to make a set.

```
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

```
{'apple', 'cherry', 'banana'}


...Program finished with exit code 0
Press ENTER to exit console.
```

Accessing items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

```
banana
cherry
apple


...Program finished with exit code 0
Press ENTER to exit console.
```

Check if item exists in a set

We can check if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

```
True


...Program finished with exit code 0
Press ENTER to exit console.
```

Adding item to a set

To add one item to a set use the add() method.

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

```
{'orange', 'cherry', 'apple', 'banana'}


...Program finished with exit code 0
Press ENTER to exit console.
```

Adding any iterable to a set

To add any iterable, i.e., set, tuple, etc. to a set, use the update() method.

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]

thisset.update(mylist)
print(thisset)
```

```
{'orange', 'apple', 'cherry', 'banana', 'kiwi'}


...Program finished with exit code 0
Press ENTER to exit console.
```

Removing an item from a set

To remove an item in a set, use the remove(), or the discard() method.
remove() throws error if item is not present in the set whereas discard() does not.

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")
print(thisset)
```

```
{'apple', 'cherry'}


...Program finished with exit code 0
Press ENTER to exit console.
```

Joining 2 sets

You can use the union() method that returns a new set containing all items from both sets.
The update() method inserts all the items from one set into another.

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

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


...Program finished with exit code 0
Press ENTER to exit console.
```

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

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

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


...Program finished with exit code 0
Press ENTER to exit console.
```

- Keep only duplicates: The intersection_update() method will keep only the items that are present in both sets.

```
set1 = {"a", "b" , "c"}
set2 = {"e", "f", "b"}

set1.intersection_update(set2)
print(set1)

```

```
{'b'}


...Program finished with exit code 0
Press ENTER to exit console.
```

- Keep all except duplicates: The symmetric_difference_update() method will keep only the elements that are NOT present in both sets.

```
set1 = {"a", "b" , "c"}
set2 = {"e", "f", "b"}

set1.symmetric_difference_update(set2)
print(set1)

```

```
{'e', 'c', 'f', 'a'}


...Program finished with exit code 0
Press ENTER to exit console.
```

Q1. Print the maximum and minimum element in a set in Python

Input : set = ([8, 16, 24, 1, 25, 3, 10, 65, 55])
Output : max is 65

Input : set = ([4, 12, 10, 9, 4, 13])
Output : min is 4

Approach: The built-in function max() and min() in Python are used to get the maximum and minimum of all the elements in a set.

Code:

```
def MIN(sets):
    return (min(sets))
def MAX(sets):
    return (max(sets))
# Driver Code
sets = set([4, 12, 10, 9, 4, 13])
print(MIN(sets))
print(MAX(sets))
```

```
def MIN(sets):
    return (min(sets))
def MAX(sets):
    return (max(sets))
# Driver Code
sets = set([4, 12, 10, 9, 4, 13])
print(MIN(sets))
print(MAX(sets))

```

```
4
13


...Program finished with exit code 0
Press ENTER to exit console.
```

Q2. Given three arrays, we have to find common elements in three sorted lists using sets.
Examples :
Input : ar1 = [1, 5, 10, 20, 40, 80]
ar2 = [6, 7, 20, 80, 100]
ar3 = [3, 4, 15, 20, 30, 70, 80, 120]
Output : [80, 20]

Input : ar1 = [1, 5, 5]
ar2 = [3, 4, 5, 5, 10]
ar3 = [5, 5, 10, 20]
Output : [5]

Approach: We have given three arrays, with the help of sets one can easily find out the intersection of these Arrays. Intersection method simply provides the intersection of both the arrays upon which you want to perform the operation of intersection (or, it simply gives out the common elements in both the array). We will be taking three arrays and then we will take out the intersection.

Code:

```
def IntersecOfSets(arr1, arr2, arr3):
    s1 = set(arr1)
    s2 = set(arr2)
    s3 = set(arr3)

    set1 = s1.intersection(s2)       #[80, 20, 100]
    result_set = set1.intersection(s3)

    final_list = list(result_set)
    print(final_list)

if __name__ == '__main__' :

    arr1 = [1, 5, 10, 20, 40, 80, 100]

    arr2 = [6, 7, 20, 80, 100]

    arr3 = [3, 4, 15, 20, 30, 70, 80, 120]

    IntersecOfSets(arr1, arr2, arr3)
```

```
30, 20]


..Program finished with exit code 0
ress ENTER to exit console.
```

Dictionary

Dictionaries are used to store data values in key:value pairs.
Dictionaries are written with curly brackets, and have keys and values

Syntax:

```
this_dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Dictionary Items

- Ordered: When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
- Changeable: Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Unindexed:
- Duplicates not allowed: Dictionaries cannot have two items with the same key.
- Any datatype: The values in dictionary items can be of any data type For example:

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

Length of a dictionary

To determine how many items a dictionary has, use the len() function.

Type of a dictionary

Dictionary is a built-in data type, so its data type is dictionary.
To verify this, use type() method.

The dict() Constructor

It is also possible to use the dict() constructor to make a dictionary.

Accessing items of a Dictionary

You can access the items of a dictionary by referring to its key name, inside square brackets. Also, There is also a method called get() that will give you the same result.

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

```
thisdict = dict(name = "John", age = 36, country = "Norway")
x = thisdict.keys()
print(x)
```

```
dict_keys(['name', 'age', 'country'])

...Program finished with exit code 0
Press ENTER to exit console.
```

Change item in a dictionary

You can change the value of a specific item by referring to its key name.

```
thisdict = dict(name = "John", age = 36, country = "Norway")
thisdict["name"] = "Raye"
print(thisdict)
```

```
{'name': 'Raye', 'age': 36, 'country': 'Norway'}

...Program finished with exit code 0
Press ENTER to exit console.
```

Adding item to a dictionary

- Adding an item to the dictionary is done by using a new index key and assigning a value to it.

```
thisdict = dict(name = "John", age = 36, country = "Norway")
thisdict["surname"] = "Smith"
print(thisdict)
```

```
{'name': 'John', 'age': 36, 'country': 'Norway', 'surname': 'Smith'}

...Program finished with exit code 0
Press ENTER to exit console.
```

- Any other dictionary or any iterable having key value pairs can also be added to a dictionary using update() method.

```
this_dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
new_dict = {"color": "red"}
this_dict.update(new_dict)
print(this_dict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}

...Program finished with exit code 0
Press ENTER to exit console.
```

Removing item from a dictionary

There are several methods to remove items from a dictionary.

- The pop() method removes the item with the specified key name.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}

...Program finished with exit code 0
Press ENTER to exit console.
```

- The popitem() method removes the last inserted item.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang'}

...Program finished with exit code 0
Press ENTER to exit console.
```

- del: The del keyword removes the item with the specified key name.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}

...Program finished with exit code 0
Press ENTER to exit console.
```

The clear() method empties the dictionary.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

```
{}

...Program finished with exit code 0
Press ENTER to exit console.
```

Looping through a dictionary

- **To Print all key names in the dictionary:**

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(x)
```

```
brand
model
year

...Program finished with exit code 0
Press ENTER to exit console.
```

- **To print all values in the dictionary:**

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
1964


...Program finished with exit code 0
Press ENTER to exit console.
```

Or use values() method.

- **To print key value pairs:**

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in thisdict.items():
  print(x, y)
```

```
rand Ford
odel Mustang
ear 1964


..Program finished with exit code 0
ress ENTER to exit console.
```

Copying a dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

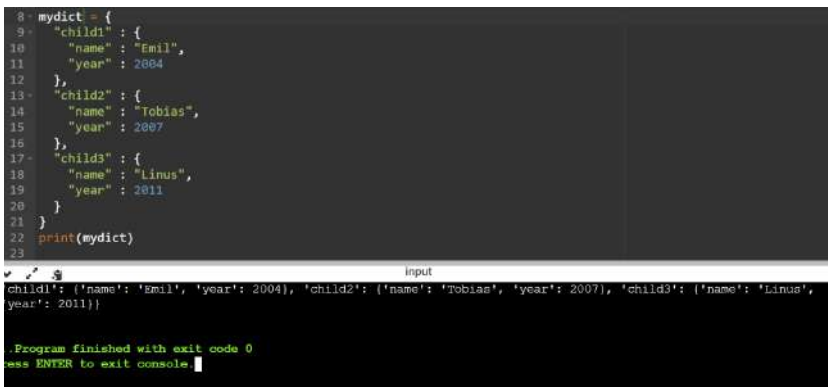
There are ways to make a copy, one way is to use the built-in Dictionary method copy().

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

```
and': 'Ford', 'model': 'Mustang',


rogram finished with exit code 0
s ENTER to exit console.
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.



Access Items in Nested Dictionaries

To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary.

Q1. Given a dictionary in Python, write a Python program to find the sum of all items in the dictionary.

Examples:

Input : {'a': 100, 'b':200, 'c':300}Output : 600
Input : {'x': 25, 'y':18, 'z':45}Output : 88

Approach: Use the sum function to find the sum of dictionary values.

Code:

```
this_dict = {
  'a': 100,
  'b': 200,
  'c': 300
}
print(sum(this_dict.values()))
```

```
this_dict = {
  'a': 100,
  'b': 200,
  'c': 300
}
print(sum(this_dict.values()))
```

```
600

...Program finished with exit code 0
Press ENTER to exit console.
```

Q2. Given a string and a number N, we need to mirror the characters from the N-th position up to the length of the string in alphabetical order. In mirror operation, we change 'a' to 'z', 'b' to 'y', and so on.

Examples:

Input : N = 3
paradox
Output : paizwlc
We mirror characters from position 3 to end.

Input : N = 6
pneumonia
Output : pneumlmrz

Code:

```
def mirrorChars(input,k):

 # create dictionary
 original = 'abcdefghijklmnopqrstuvwxyz'
 reverse = 'zyxwvutsrqponmlkjihgfedcba'
 dictChars = dict(zip(original,reverse))

 # separate out string after length k to change
 # characters in mirror
 prefix = input[0:k-1]
 suffix = input[k-1:]
 mirror = ''

 # change into mirror
 for i in range(0,len(suffix)):
  mirror = mirror + dictChars[suffix[i]]

 # concat prefix and mirrored part
 print (prefix+mirror)
# Driver program
if __name__ == "__main__":
 input = 'paradox'
 k = 3
 mirrorChars(input,k)
```

```
paizwlc


...Program finished with exit code 0
Press ENTER to exit console.
```

MCQ-1

What will be the output of the following code snippet?
colors = ["red", "green", "burnt sienna", "blue"]
colors[2]

What is the output of the colors[2] expression?

1. 'Red'
2. It causes a run-time error.
3. 'burnt sienna'
4. 'green'

MCQ-2

What will be the output of the following code snippet?
colors = ["red", "green", "burnt sienna", "blue"]
"yellow" in colors

What is the result of the yellow in colors expression?

a. False
b. Correct
c. 4
d. ValueError: 'yellow' is not in list

MCQ-3

Square brackets in an assignment statement will create which type of data structure?
(s=[])

a. List
b. Tuple
c. Set
d. Dictionary

MCQ-4

What type of data is: arr = [(1,1),(2,2),(3,3)]?

a. Array of tuples
b. Tuples of lists
c. List of tuples
d. Invalid type

MCQ-5

What is the output of the following program : print((1, 2) + (3, 4))?

a. (1, 2), (3, 4)
b. (4, 6)
c. (1, 2, 3, 4)
d. Invalid Syntax

Chapter-4

Function

- Functions in Python are blocks of reusable code that perform a specific task.
- They help organize code, promote reusability, and make programs easier to understand and maintain.
- Functions are defined using the def keyword, followed by the function name, required and optional parameters in parentheses, return type, and a colon.
- The function block is indented and contains the code to be executed when the function is called.

Why Function?

1. **Modularity:** Functions break down complex programs into smaller, manageable parts, improving code organization and structure.
2. **Reusability:** Functions allow code to be reused, saving time and effort by avoiding code duplication.
3. **Abstraction:** Functions provide a high-level overview, hiding implementation details and making code more readable and understandable.
4. **Decomposition:** Functions help break down complex problems into smaller, more manageable tasks, simplifying the problem-solving process.
5. **Code Maintenance:** Functions make code easier to maintain and debug by isolating and focusing on specific parts of the program.

Types of Function

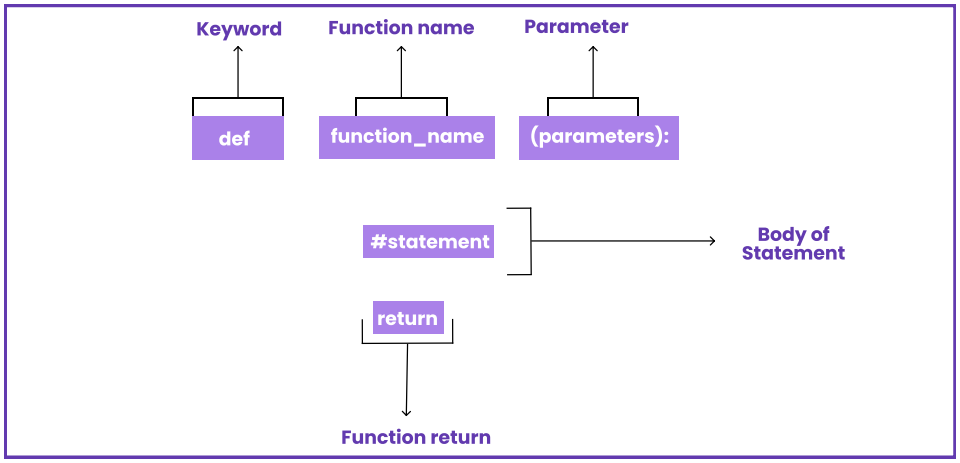
1. Build-in function: These are pre-defined functions that come with Python, such as print(), len(), input(), range(), and type(). They are readily available for use without requiring any additional code.

2. User-defined function: These functions are created by the user to perform specific tasks. They provide a way to modularize code and promote reusability. User-defined functions are defined using the def keyword.

Creating a function

Syntax:

```
def function_name(parameters):
    # Function body (code to be executed)
    # ...
    # Optional return statement
    return value
```



Let's break down the different components:

- def: Keyword used to define a function.
- function_name: Name given to the function. It should follow Python naming conventions.
- parameters: Optional input parameters that the function can accept. These are placeholders for values passed into the function.
- :: Colon to indicate the start of the function block.
- Function body: Contains the code to be executed when the function is called. It is indented with consistent whitespace.
- return: Optional statement used to return a value from the function. If not specified, the function returns None.

Calling a function

Once a function is defined, it can be called by using its name followed by parentheses. Arguments can be passed into the function if it expects parameters.

Syntax:

```
function_name(argument1, argument2, ...)
```

Example: Suppose we have a function named say_hello that prints “hello!”.

```
def say_hello():
    print("Hello!")

# Call the say_hello function
say_hello()
```

Output:

Hello!

Arguments

In Python, arguments are values passed to a function when calling it. They provide input to the function and allow it to perform its task using the provided data.

Syntax:

```
function_name(argument1, argument2, ...)
```

Types of Arguments

1. Default argument

These are parameters in a function that have a default value assigned. If an argument is not provided for a default parameter, the default value is used.

Example:

```
def greet(name, message="Hello"):
    print(message, name)

greet("Bob")  # Uses the default value "Hello" for the message parameter
```

2. Keyword arguments (named arguments)

These arguments are passed to a function using the name of the parameter they correspond to, followed by the = sign. The order of keyword arguments does not matter.

Example:
```
def greet(name, age):
    print("Hello", name, "you are", age, "years old.")

greet(name="Alice", age=25)  # Keyword arguments: name="Alice", age=25
```

This will also work fine:
```
greet(age=25, name="Alice")
```

3. Positional arguments
These are arguments passed to a function in the order specified by the function's parameter list. The values are assigned to the corresponding parameters based on their position.

Example:
```
def add_numbers(x, y):
    sum = x + y
    print("Sum:", sum)

add_numbers(5, 3)  # Positional arguments: 5 is assigned to x, 3 is assigned to y
```

4. Arbitrary arguments (variable-length arguments *args and **kwargs)
Variable-length arguments in Python allow a function to accept a varying number of arguments.
There are two types of variable-length arguments:

- non-keyword variable-length arguments (*args)
- keyword variable-length arguments (**kwargs).

Non-Keyword Variable-Length Arguments (*args):

- The *args syntax allows a function to accept any number of non-keyword arguments.
- Within the function, args becomes a tuple that holds all the passed arguments.
- You can access and process the individual arguments using indexing or iterating over args.

Example:
```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

result = sum_numbers(1, 2, 3, 4)
print(result)  # Output: 10
```

Keyword Variable-Length Arguments (**kwargs):

- The **kwargs syntax allows a function to accept any number of keyword arguments.
- Within the function, kwargs becomes a dictionary that holds the passed keyword arguments.
- You can access and process the individual arguments using dictionary operations.

Example:
```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(key + ": " + value)

display_info(name="Alice", age="25", city="New York")
# Output:
# name: Alice
# age: 25
# city: New York
```

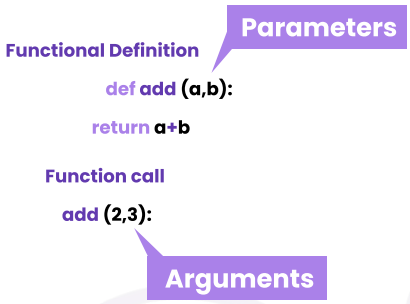
Difference between parameters and arguments

Parameters:

- Parameters are the variables listed in the function definition.
- They act as placeholders for the values that will be passed to the function when it is called.
- Parameters are specified within the parentheses in the function definition.
- Parameters define what values a function expects to receive when it is called, and they have local scope within the function.

Arguments:

- Arguments are the actual values passed to a function when it is called.
- They correspond to the parameters defined in the function.
- Arguments are specified within the parentheses in the function call.
- Arguments provide the necessary data to the function for it to perform its task.



Return Type

The return type of a function refers to the type of value that the function returns when it is executed. The return type can be explicitly specified using type annotations or implicitly determined based on the value returned by the function.

Example:

```
def add_numbers(a: int, b: int) -> int:
    return a + b
```

In this example, the function add_numbers takes two parameters a and b, both of type int, and it explicitly specifies the return type as int using the -> syntax. The function performs the addition operation and returns the result, which is of type int.

In the absence of type annotations, Python infers the return type based on the value returned by the function.

Example:

```
def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

In this case, the function is_even does not have an explicitly specified return type. However, based on the True and False values returned, Python infers that the return type is bool.

Note: It's important to note that in Python, a function can also return None if there is no explicit return statement, or if the return statement does not provide a value.

<SKIP>

return vs yield

return and yield are both used in Python for returning values from functions or generators, but they have different behaviors and use cases.

return is used in regular functions to terminate the function's execution and immediately return a value to the caller. When a return statement is encountered in a function, the function stops executing, and the value provided after return is passed back to the caller. After a return statement is executed, the function is finished, and any subsequent code is not executed.

Example:

```
def calculate_sum(a, b):
    return a + b

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

In this example, the calculate_sum function uses return to return the sum of a and b back to the caller.

yield is used in generators to create iterable objects that can be iterated over using a loop or other iterator-consuming constructs. When a generator function encounters a yield statement, it suspends its execution and yields a value to the caller. The function's state is saved, allowing it to be resumed later. Each time the generator is iterated, it executes until it reaches the next yield statement.

Example:

```
def generate_numbers():
    for i in range(5):
        yield i

numbers = generate_numbers()
for num in numbers:
    print(num)  # Output: 0, 1, 2, 3, 4
```

In this example, the generate_numbers function is a generator that yields numbers from 0 to 4. The yield statement allows the function to generate one value at a time, and the function's state is saved between iterations.

In summary, return is used in regular functions to immediately return a value and terminate the function, while yield is used in generators to create iterable objects and allow incremental value generation.

Nested functions

Nested functions in Python refer to defining a function inside another function. The inner function has access to the scope of the outer function, including its variables, parameters, and other nested functions.

Example:

```
def outer_function():
    x = 1  # Variable in the outer function

    def inner_function():
        y = 2  # Variable in the inner function
        result = x + y
        return result

    return inner_function()

output = outer_function()
print(output)  # Output: 3
```

In this example, we have an outer function called outer_function() that defines a variable x with a value of 1. Inside the outer function, there is an inner function called inner_function(). The inner function defines a variable y with a value of 2 and performs the addition of x and y. The result is returned from the inner function. Now it is important to note why the inner_function() got called. Well, this is because the outer_function's return statement calls the inner_function() i.e. return inner_function().

Need of Nested Functions

Nested functions are useful for encapsulating functionality within a specific scope and promoting code organization. They can help in creating reusable code blocks, implementing closures, and achieving better modularity. The inner function can access the variables of the outer function, even after the outer function has finished executing.

Note that the inner function is not accessible outside the scope of the outer function. It is confined within the scope of the outer function, providing encapsulation and preventing naming conflicts with other parts of the program.

Pass by reference and Pass by value

Pass by Value (Immutable Objects):

- Immutable objects, such as numbers (integers, floats), strings, and tuples, are passed by value.
- When an immutable object is passed to a function, a copy of the object's value is created and assigned to a new local variable within the function.
- Changes made to the local variable do not affect the original object outside the function.

Example:

```
def modify_number(num):
    num += 1
    print("Inside function:", num)

x = 5
modify_number(x)
print("Outside function:", x)
```

Pass by Reference (Mutable Objects):

- Mutable objects, such as lists, dictionaries, and custom objects, are passed by reference.
- When a mutable object is passed to a function, the reference to the object is passed, not a copy of the object itself.
- Changes made to the mutable object inside the function affect the original object outside the function.

Example:

```
def modify_list(lst):
    lst.append(4)
    print("Inside function:", lst)

my_list = [1, 2, 3]
modify_list(my_list)
print("Outside function:", my_list)
```

It's important to note that even though mutable objects are passed by reference, reassigning the parameter within the function will not affect the original object.

Example:

```
def reassign_list(lst):
    lst = [4, 5, 6]
    print("Inside function:", lst)

my_list = [1, 2, 3]
reassign_list(my_list)
print("Outside function:", my_list)
```

When the function reassign_list is called with my_list, a new local variable lst is created within the scope of the function, and it references a new list [4, 5, 6]. This reassignment statement inside the function creates a new list object and points the local variable lst to that new object, but it doesn't affect the original list my_list outside the function. The original list my_list remains unchanged outside the function, and the reassignment only affects the local variable lst within the function's scope.

Important

In summary, Python uses a combination of pass by value and pass by reference depending on the object type:

Pass by Value: Immutable objects, such as numbers (integers, floats), strings, and tuples, are passed by value. A copy of the object's value is created and assigned to a new local variable within the function. Changes made to the local variable do not affect the original object outside the function.

Pass by Reference: Mutable objects, such as lists, dictionaries, and custom objects, are passed by reference. The reference to the object is passed, not a copy of the object itself. Changes made to the mutable object inside the function affect the original object outside the function.

To summarize, reassigning a parameter variable within a function creates a new local variable, regardless of whether the object is mutable or immutable. The reassignment does not affect the original variable outside the function's scope. If you want to modify the original object, you need to directly modify its contents instead of reassigning the parameter variable.

Lambda or Anonymous functions

A lambda function, also known as an anonymous function or a lambda expression, is a way to define small, one-line functions without using the def keyword. Lambda functions are commonly used for simple, one-time operations and are typically defined inline where they are needed.

Syntax:

```
lambda arguments: expression
```

Let's break down the components of a lambda function:

- lambda: Keyword used to define a lambda function.
- arguments: Optional arguments that the lambda function can accept.
- expression: The expression that gets evaluated and returned by the lambda function.

Example: a lambda function that adds two numbers:

```
add = lambda x, y: x + y
result = add(3, 5)
print(result)  # Output: 8
```

In this example, lambda x, y: x + y defines a lambda function that takes two arguments x and y, and returns their sum. The lambda function is assigned to the variable add, and then called with arguments 3 and 5. The returned value, 8, is stored in the result variable and printed.

Lambda functions are particularly useful when you need a simple function without a formal definition or when you want to pass a function as an argument to another function, such as in map(), filter(), or sort() operations.

Built-in functions

Python provides a wide range of built-in functions that are readily available for use without requiring any additional code or imports.

Here are some commonly used built-in functions in Python:

1. **print():** Prints output to the console.
2. **input():** Reads input from the user via the console.
3. **len():** Returns the length of an object, such as a string, list, or tuple.
4. **type():** Returns the type of an object.
5. **range():** Generates a sequence of numbers.
6. **int(), float(), str(), bool():** Converts a value to an integer, float, string, or boolean, respectively.
7. **max(), min():** Returns the maximum or minimum value from a sequence or a set of arguments.
8. **abs():** Returns the absolute value of a number.
9. **sum():** Returns the sum of all elements in a sequence.
10. **round():** Rounds a number to a specified number of decimal places.
11. **sorted():** Returns a sorted version of a sequence or an iterable.
12. **enumerate():** Returns an iterator of tuples containing index-value pairs.
13. **zip():** Combines multiple iterables into a single iterable of tuples.
14. **map():** Applies a function to each item in an iterable and returns an iterator of the results.
15. **filter():** Filters elements from an iterable based on a given function.
16. **any(), all():** Checks if any or all elements in an iterable are True.
17. **open():** Opens a file for reading or writing.
18. **eval():** Evaluates a string as a Python expression.
19. **str.format():** Formats a string by substituting placeholders with values.
20. **dir():** Returns a list of names in the current namespace or of an object's attributes.

Question: Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.

Code: https://pastebin.com/tCwhDeSJ

Question: Write a Python function that takes a number as a parameter and checks whether the number is prime or not.

Code: https://pastebin.com/LV5TavZw

MCQ Time

1. What will be the output of the following code snippet?

```
x = 50
def func(x):
  x = 2
func(x)
print('x is now', x)
```

(a) x is now 50
(b) x is now 2
(c) x is now 100
(d) None

Answer: (a) x is now 50

2. What will be the output of the following code snippet?

```
def say(message, times = 1):
    print(message * times)
say('Hello')
say('World', 5)
```

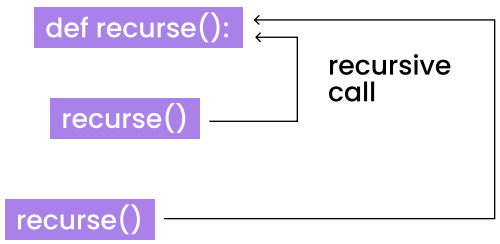
(a) Hello
WorldWorldWorldWorldWorld
(b) Hello
World 5
(c) Hello
World,World,World,World,World
(d) Hello
HelloHelloHelloHelloHello

Answer: (a) Hello
WorldWorldWorldWorldWorld

Recursion

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.



A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problem. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

Example:-
Code:- https://pastebin.com/mkn8VxK6

Key Characteristics of Recursive Functions

- **Self-Calling:** The defining feature of a recursive function is that it calls itself within its body. This allows the function to break down a complex problem into simpler subproblems of the same type.
- **Base Case:** To avoid infinite recursion, a recursive function must have a base case, which is the simplest form of the problem that can be directly solved without further recursion. When the function encounters the base case, it stops calling itself and starts returning values back through the call stack.
- **Recursive Case:** The recursive case is the part of the function where it calls itself with a modified (usually smaller) version of the original problem. This recursive call allows the function to work towards the base case by solving smaller subproblems and combining their results to get the final solution.
- **Repetition and Replication:** Recursive functions typically handle repetitive or replicated structures in a problem. The function addresses the same type of problem multiple times with slightly different inputs until it reaches the base case.
- **Memory Usage:** Recursive functions utilize the call stack to store the intermediate state of function calls. Each time the function calls itself, a new stack frame is created, and when the base case is reached, the stack frames are popped off as the functions return their results. Excessive recursion without a proper base case can lead to stack overflow errors.
- **Readability and Elegance:** In some cases, recursive solutions can provide a more elegant and concise way to express algorithms, especially for problems with clear repetitive patterns.

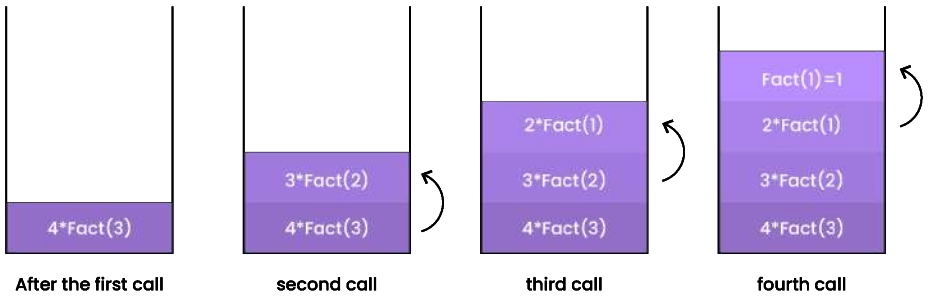
Base Case: The base case is the stopping condition of the recursive function. It is the simplest scenario where the function does not call itself but instead returns a straightforward result. The base case is essential to prevent infinite recursion and allows the recursion to terminate successfully.

```
function countdown(num) {
    if (num <=0) {
       console.log("You've reached the end");
       return;
    }                                          ——————→ Base Case
    console.log(num);
    num--
    countdown(num)      ——————→ Recursive function
}
```

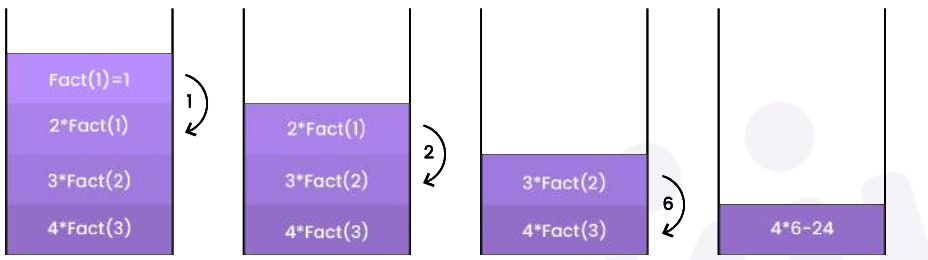
Recursive Case: The recursive case is where the function calls itself to solve a smaller or simpler version of the original problem. By breaking the problem down into smaller subproblems, each handled by the same function, the recursive case aims to reach the base case eventually.

The Call Stack and Recursive Calls

When function call happens previous variables get stored in stack



Returnig values from base case to caller function



- The call stack plays a critical role in managing recursive calls. It is a region of memory used by the program to keep track of function calls and their local variables. When a function is called, its execution context (local variables, parameters, and return address) is pushed onto the call stack. When the function returns, its context is popped off the stack.
- For recursive functions, the call stack is particularly important because it allows the program to maintain multiple active instances of the same function, each with its own set of local variables and arguments. This way, the recursive calls can be nested and tracked until the base case is reached.

Example:-

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

Now, let's calculate factorial(5) step-by-step:

- factorial(5) is called.
  - The function starts executing, and n is 5.
- factorial(5) calls factorial(4).
  - The current state of factorial(5) (with n = 5) is pushed onto the stack.
  - The new instance of factorial(4) is now the active function.
  - factorial(4) calls factorial(3).
    - The current state of factorial(4) (with n = 4) is pushed onto the stack.
    - The new instance of factorial(3) is now the active function.

- factorial(3) calls factorial(2).
  - The current state of factorial(3) (with n = 3) is pushed onto the stack.
  - The new instance of factorial(2) is now the active function.
- factorial(2) calls factorial(1).
  - The current state of factorial(2) (with n = 2) is pushed onto the stack.
  - The new instance of factorial(1) is now the active function.
- factorial(1) calls factorial(0).
  - The current state of factorial(1) (with n = 1) is pushed onto the stack.
  - The new instance of factorial(0) is now the active function.
- Base Case: factorial(0) returns 1.
  - When n becomes 0, factorial(0) is reached, and it returns 1.
  - The call stack starts to unwind.
- Unwinding the Call Stack:
  - The result of factorial(0) (which is 1) is returned to factorial(1).
  - The result of factorial(1) (which is 1 * 1 = 1) is returned to factorial(2).
  - The result of factorial(2) (which is 2 * 1 = 2) is returned to factorial(3).
  - The result of factorial(3) (which is 3 * 2 = 6) is returned to factorial(4).
  - The result of factorial(4) (which is 4 * 6 = 24) is returned to factorial(5).
- Final Result:
  - The result of factorial(5) (which is 5 * 24 = 120) is obtained.

- As the recursive calls return and the call stack unwinds, the results are combined to compute the factorial of the original number (5 in this case). The call stack efficiently manages the recursive function calls, allowing the program to handle complex problems through the recursive approach.

How Recursive Calls Work

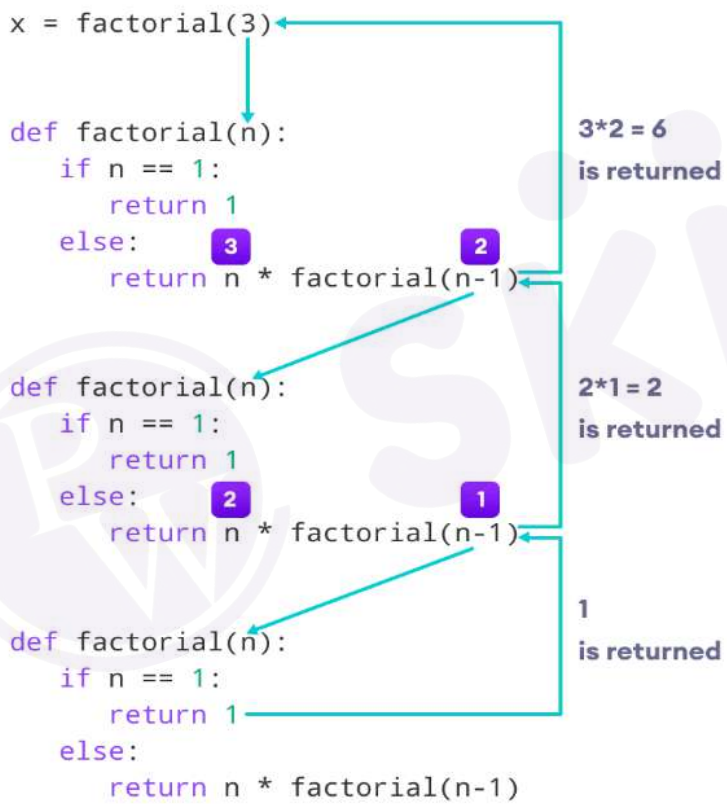
- Recursive calls work by allowing a function to call itself within its own definition. When a function encounters a recursive call, it creates a new instance of itself (a new stack frame) with its own set of local variables, arguments, and return address. This new instance of the function starts executing independently of the previous one, with its own set of data.

Here's a step-by-step explanation of how recursive calls work:

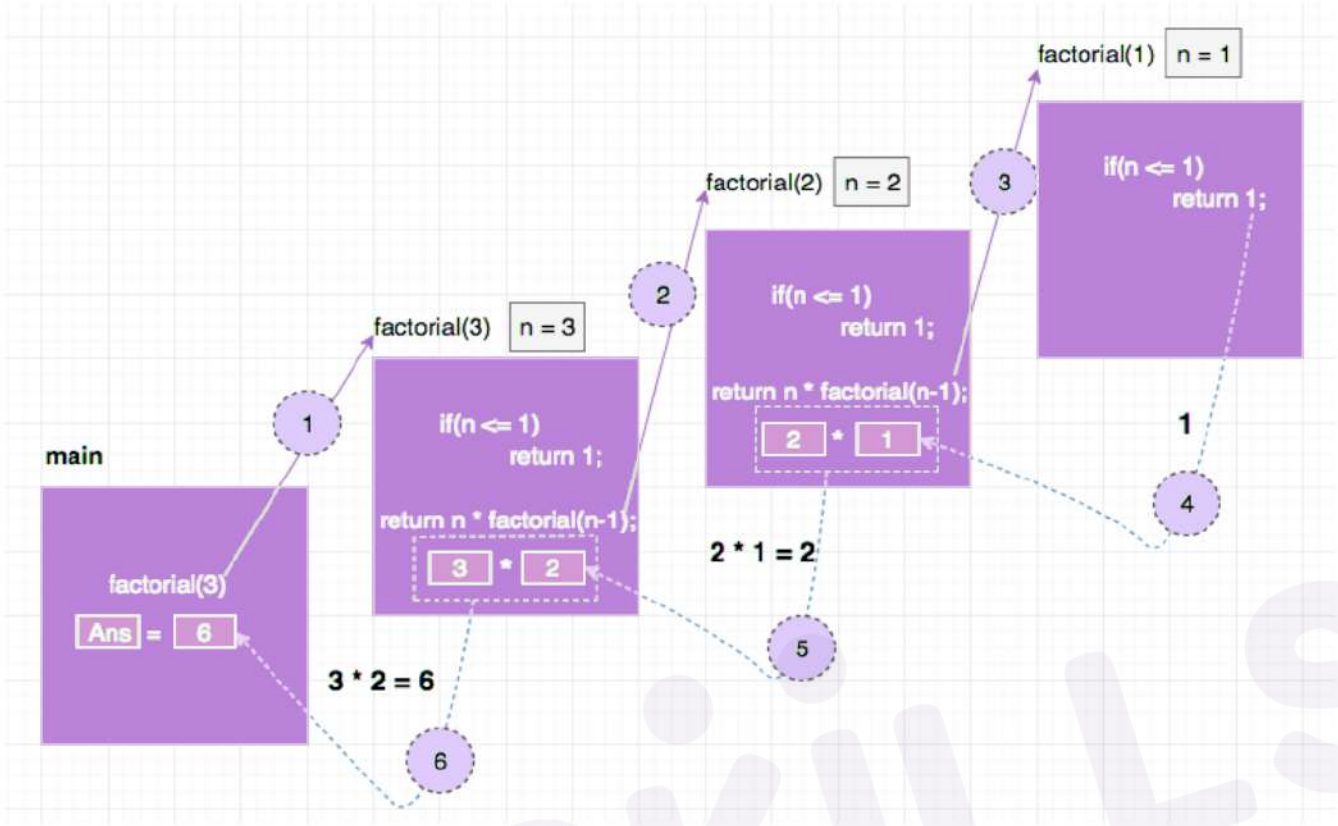
- **Initial Function Call:** The process starts with an initial function call from the main program or another function. The function is called with a specific set of input arguments.
- **Function Execution:** The function begins executing its code, which includes a recursive call to itself. When the recursive call is encountered, the current state of the function (values of local variables, arguments, etc.) is pushed onto the call stack to be saved for later.
- **Recursive Call:** The function makes a call to itself, passing modified (usually smaller) arguments to work on a smaller instance of the problem. The recursive call creates a new stack frame for the new function call and begins executing the function code again from the beginning.
- **Multiple Recursive Calls:** If the recursive function has more than one recursive call within its body, each call will create its own stack frame, and these frames will be stacked on top of each other in a "last in, first out" manner.

- **Base Case Check:** Inside the function, there is typically a conditional statement that checks for a base case - the simplest form of the problem that can be directly solved without further recursion. When the base case is reached, the function will stop making further recursive calls and start unwinding the call stack.
- **Unwinding the Call Stack:** As the function returns from the recursive calls, the call stack starts to unwind. Each function call returns its result, and the corresponding stack frame is removed from the call stack.
- **Combining Results:** As the recursive calls return, the function combines the results from each recursive call to solve the original problem or complete the task.
- **Final Result:** The final result is obtained when the initial function call's stack frame is reached and the function returns the ultimate result to the calling context (main program or another function).

Visualising Recursion



- The topmost function call in the stack diagram is the one that is currently executing. In this case, it is the factorial(3) function call. The next function call in the stack diagram is the factorial(2) function call. This function call was made by the factorial(3) function call, and it is waiting to be executed. The remaining function calls in the stack diagram are all waiting to be executed as well.
- The values of the function's parameters and local variables are also shown in the stack diagram. In this case, the n parameter for each function call is the number that the function is currently calculating the factorial of.
- As the recursive function executes, the stack diagram will grow. Each time the function calls itself, a new function call will be added to the stack diagram. When the function reaches its base case, the function calls will be popped off the stack in reverse order.



Write a program to Print numbers from n to 1.

code:-https://pastebin.com/yyajDRPt

Explanation:-
The function print_n_to_1 recursively prints numbers from 'n' to 1 in descending order. It starts with 'n', prints it, then calls itself with 'n-1' until 'n' becomes 1. If 'n' is not a positive integer, the program asks the user to input a positive integer and then prints the sequence from the given 'n' down to 1.

Time Complexity: The time complexity of the print_n_to_1 function is O(n) because it recursively prints numbers from 'n' to 1 in descending order. The function makes 'n' recursive calls before reaching the base case where 'n' becomes 0, so the time complexity is linear, proportional to the value of 'n'.

Space Complexity: The space complexity of the print_n_to_1 function is O(n) due to the recursion. Each recursive call adds a new stack frame to the call stack, and the maximum depth of the call stack is 'n'. Therefore, the space complexity is linear, proportional to the value of 'n'.

Write a program to Print numbers from 1 to n

Code:-https://pastebin.com/fLZC6Emr

Explanation:-
1. The function print_numbers_recursive(n) is called with an integer n.
2. Inside the function:

a. It checks if n is greater than 0.
b. If n is greater than 0:
    i. The function calls itself with the value n - 1.
    ii. This is the recursive step where the function is invoked with a smaller value.
    iii. The current call is paused, and a new call is added to the call stack with a smaller value of n.
c. After the recursive call returns:
    i. The current value of n is printed.
    ii. This happens as the call stack starts unwinding, and each function call resumes its execution after the recursive call.

3. This process continues until the value of n becomes 0 (base case).
4. As the recursive calls unwind:
    a. The function prints the numbers in ascending order from 1 to the original value of n.

Time Complexity: The time complexity of the print_numbers_recursive function is O(n), where n is the input number. Each recursive call reduces the value of n by 1, and since there are n total calls (from n to 1), the time complexity grows linearly with the input.

Space Complexity: The space complexity of the print_numbers_recursive function is O(n) as well. This is because the recursive calls are added to the call stack, and in the worst case, there will be n calls stacked up due to the recursion.

Write a program to Print sum from 1 to n.

Code:-https://pastebin.com/J13i7d49

Explanation:-
1. We have a function calculate_sum_recursive(n) that takes an integer n as a parameter.
2. Inside the function:
    a. Check if n is equal to 1.
    b. If n is 1, return 1 (base case).
    c. If n is not 1, calculate the sum of n and the result of a recursive call to calculate_sum_recursive(n - 1).
    d. This adds up the current number n with the sum of numbers from 1 to n-1.
3. Take user input for the value of n.
4. Call the calculate_sum_recursive function with the user-provided value of n.
5. The recursive function calculates the sum by breaking down the problem of finding the sum of numbers from 1 to n into smaller subproblems (finding the sum of numbers from 1 to n-1).
6. The recursion continues until the base case is reached (when n becomes 1).
7. As the recursive calls unwind, the function adds up the numbers to compute the total sum.
8. Finally we print the calculated sum.

Time Complexity: The time complexity of the calculate_sum_recursive function is O(n), where n is the input number. Each recursive call reduces the value of n by 1, and there are n total calls (from n to 1) needed to calculate the sum.

Space Complexity: The space complexity of the calculate_sum_recursive function is O(n) as well. This is because each recursive call adds a new function call to the call stack, and in the worst case, there will be n calls stacked up due to the recursion. Therefore, the space used by the call stack grows linearly with the input number n, resulting in a linear space complexity.

Make a function which calculates the factorial of n using recursion.

Code:-https://pastebin.com/6UPs3Etc

Explanation:-
The factorial_recursive function calculates the factorial of a given number 'n' using recursion. If 'n' is 0 (the base case), the function returns 1, as the factorial of 0 is defined to be 1. Otherwise, it returns 'n' multiplied by the factorial of 'n - 1', effectively calculating the factorial of 'n' through a recursive chain of multiplications.

Time Complexity: The time complexity of the factorial_recursive function is O(n) because each recursive call reduces 'n' by 1 until it reaches the base case where 'n' becomes 0. Therefore, the function makes 'n' recursive calls, resulting in linear time complexity.

Space Complexity: The space complexity of the factorial_recursive function is O(n) as well because the recursion stack will have 'n' frames at its peak depth. Each recursive call consumes space on the stack until the base case is reached, at which point the stack frames are gradually removed, and the space is released.

Make a function that calculates ‘a’ raised to the power ‘b’ using recursion.

Code:-https://pastebin.com/qPjcbu1u

Explanation:-

- The power_recursive function calculates 'a' raised to the power 'b' using recursion. If the exponent 'b' is 0, it returns 1 (base case).
- Otherwise, it recursively multiplies 'a' with the result of power_recursive

(a, b -1), reducing 'b' by 1 in each recursive call. The code print(power_recursive(2,4)) calculates 2^4 using the recursive function, which yields the result 16.

Time Complexity: The time complexity of the power_recursive function is O(b) because each recursive call reduces the exponent 'b' by 1 until it reaches the base case where 'b' becomes 0. Therefore, the function makes 'b' recursive calls, resulting in linear time complexity.

Space Complexity: The space complexity of the power_recursive function is O(b) as well because the recursion stack will have 'b' frames at its peak depth. Each recursive call consumes space on the stack until the base case is reached, at which point the stack frames are gradually removed, and the space is released.

Program on Fibonacci Series.

Code:-https://pastebin.com/YumK4iRV

Explanation:-
The given Python code defines a function fibonacci_sequence(n_terms) that generates the Fibonacci sequence up to the specified number of terms (n_terms). The function starts with the first two terms of the sequence (0 and 1) and continues adding new terms by summing the last two terms in the sequence. It does this in a while loop until the desired number of terms is reached. Finally, it returns the generated Fibonacci sequence.

Time complexity: The time complexity of the fibonacci_sequence function is O(n), where 'n' is the number of terms requested by the user. This is because the function uses a while loop to generate the Fibonacci sequence up to the specified number of terms. Each iteration of the loop takes constant time, and the loop runs 'n' times, making the overall time complexity linear.

Space Complexity: The space complexity of the fibonacci_sequence function is O(n), as it creates a list fib_seq to store the Fibonacci sequence. The size of this list grows linearly with the number of terms requested by the user, so the space complexity is also linear.

Write a function to calculate the nth fibonacci number using recursion.

Code:- https://pastebin.com/RB8jN80u

Explanation:-
The function fibonacci(n) calculates the nth Fibonacci number using recursion. The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, starting with 0 and 1.

Time complexity:-

- The time complexity of the fibonacci function is exponential, specifically O(2^n).
- This is because, in the recursive implementation, each call to the fibonacci function results in two additional recursive calls (for n - 1 and n - 2). As a result, the number of recursive calls increases exponentially with the value of n.

Space complexity:-
In this code, fibonacci(7) will require a stack with 7 frames to calculate the 7th Fibonacci number, and thus the space complexity is O(7), which is equivalent to O(n) in this case.

Pros of Recursion:-

- **Clarity and Readability:** Recursive solutions can often be more intuitive and easier to understand for problems that have a clear repetitive structure or involve traversing hierarchical data structures (e.g., trees, graphs).
- **Simplicity:** Recursion can lead to more concise code compared to iterative solutions, as it simplifies the process of breaking down a complex problem into smaller subproblems.
- **Divide and Conquer:** Recursive algorithms naturally follow the "divide and conquer" approach, breaking a problem into smaller parts that are more manageable and easier to solve.
- **Elegant Problem-Solving:** In certain cases, recursion provides a more elegant way to express an algorithm, making it easier to reason about and maintain.
- **Handling Recursive Data Structures:** When working with recursive data structures, such as linked lists or trees, recursive functions often provide a natural and concise way to process or traverse them.

Cons of Recursion:-

- **Stack Space Usage:** Recursive calls consume memory on the call stack. If the recursion is too deep or doesn't have a proper base case, it can lead to a stack overflow, causing the program to crash.
- **Performance Overhead:** Recursive calls often involve additional function call overhead and can be less efficient than iterative solutions for certain problems, due to the constant creation and destruction of stack frames.
- **Debugging Complexity:** Debugging recursive functions can be more challenging compared to iterative code. If a base case is missing or incorrectly defined, it can lead to infinite recursion or incorrect results.
- **Algorithmic Complexity:** Some problems might be better suited for iterative solutions or other algorithms. Recursion is not always the most efficient approach, especially when the problem doesn't naturally exhibit a repetitive structure.
- **Tail Recursion Optimization Limitations:** In some programming languages, tail recursion optimization might not be available or limited, leading to excessive stack space usage for tail-recursive functions.
- **Code Tracing Difficulty:** Tracing the flow of execution in recursive functions can be more complicated than iterative code, which may make it harder to understand the program's behaviour during debugging.

Recursive Algorithms in Practical Applications

- **Tree and Graph Traversal:** Recursive algorithms are often used to traverse and search through hierarchical data structures like trees and graphs. For example, depth-first search (DFS) and breadth-first search (BFS) are classic recursive algorithms for traversing trees and graphs.
- **Sorting Algorithms:** Merge sort and quicksort are popular sorting algorithms that use recursion. They divide the input into smaller parts, sort them recursively, and then combine the sorted subarrays to produce the final sorted output.
- **Dynamic Programming:** Many dynamic programming algorithms are recursive in nature. Dynamic programming is a technique used to solve problems by breaking them down into overlapping subproblems, and recursive calls help find solutions to those subproblems.
- **Fractals and Graphics:** Recursive algorithms are widely used in generating fractals, which are complex geometric patterns that repeat at different scales. Recursive methods are also used in graphics processing and rendering, particularly in generating complex shapes and patterns.
- **Combinatorial Problems:** Recursive algorithms are often employed in solving combinatorial problems, such as generating permutations, combinations, and subsets of a given set.
- P**arsing and Syntax Analysis:** Recursive descent parsing is a technique used in compiler design and natural language processing to analyze and parse the syntax of languages. It involves recursive calls to handle nested structures.
- **Backtracking Algorithms:** Backtracking algorithms use recursion to systematically explore possible solutions to a problem and backtrack when a solution is found to be invalid. Examples include solving puzzles like the N-Queens problem or the Sudoku puzzle.
- **Dynamic Data Structures:** Recursive algorithms are used to manipulate and traverse dynamic data structures, such as linked lists, binary trees, and other tree-like structures.
- **Mathematical Calculations:** Recursive algorithms are employed in mathematical calculations like factorial, Fibonacci numbers, and exponentiation.
- **Image Processing:** Recursive algorithms can be used for image processing tasks like edge detection, region growing, and noise reduction.

Chapter-5

String in Python

In Python, strings are a sequence of characters enclosed within single (' '), double (" "), or triple (''' ''' or """ """) quotes. They are used to represent text data and are one of the fundamental data types in Python. Strings are immutable, which means once they are created, their content cannot be changed. However, you can create new strings by manipulating existing ones.

Syntax

Strings can be created using single, double, or triple quotes. For example:
Using Single quote: 'Hello, World!'
Using Double quotes: "Python is awesome."
Using Triple quotes: '''Strings in Python are super fun'''

Assigning string to a variable

Using single quotes:

```
name = 'John'
```

Using double quotes:

```
message = "Hello, World!"
```

Assigning Multiline string to a variable
Multiline strings are created using triple quotes.
For eg:
paragraph = '''This is a multiline string.
You can write multiple lines within triple quotes.'''

After assigning a string to a variable, you can use that variable throughout your code to access and manipulate the string data.

For example, let's assign a string to a variable and then use it in a print statement:
```
greeting = "Hello, there!"
print(greeting)  # Output: Hello, there!
```

You can also perform various operations on the string using the assigned variable:
```
name = "Alice"
greeting = "Hello, " + name
print(greeting)  # Output: Hello, Alice
```

Array-like indexing in strings

- The strings support array-like indexing, which allows you to access individual characters or substrings using index values.
- The indexing of strings is 0-based, meaning the first character has an index of 0, the second character has an index of 1, and so on.
- You can also use negative indexing to access characters from the end of the string.

Example:
Accessing Individual Characters:
```
text = "Hello, World!"

print(text[0])   # Output: H
print(text[4])   # Output: o
print(text[-1])  # Output: !
```

We access individual characters in the string text using positive and negative indices. Positive indices start from the beginning of the string, while negative indices start from the end.

Traversing a string
Traversing a string in Python means iterating through each character of the string one by one. It's useful when you need to perform specific operations on each character, search for patterns, or count occurrences of certain characters or substrings within the string.

Using a for loop
https://pastebin.com/3rym6FQr

Using a while loop
https://pastebin.com/Gv5n5hBt

Using list comprehension
https://pastebin.com/m3AE1dpe

Using Enumerate for Both Index and Character
https://pastebin.com/zW7pxLSi

String length
In Python, you can determine the length of a string using the built-in len() function. The len() function returns the number of characters in the string, including spaces and special characters.
Example:
```
text = "Hello, World!"
length = len(text)
print(length)  # Output: 13
```

String find()
The find() method finds the first occurrence of the specified value. The find() method returns -1 if the value is not found.

Question: Check if a character or substring is present in a string
Code: https://pastebin.com/HdBYjbsS

Slicing a string
Slicing a string in Python allows you to extract a portion of the string by specifying the start and end indices. The syntax for slicing is string [start:end], where start is the index of the first character to include in the slice (inclusive), and end is the index of the first character to exclude from the slice (exclusive).

Note: Remember that slicing doesn't modify the original string; it creates a new string containing the specified portion of the original string.

Slicing from the start
When slicing a string in Python, you can omit the start index to slice from the beginning of the string. If you don't specify the start index, Python assumes that you want to start from the first character (index 0). This is known as slicing from the start.
Code: https://pastebin.com/UmXEYPsB

Slicing till the end
When slicing a string in Python, you can omit the end index to slice until the end of the string. If you don't specify the end index, Python assumes that you want to include all characters until the end. This is known as slicing till the end.
Code: https://pastebin.com/aRtXRRrz

Example:
Code: https://pastebin.com/BWFUeGnc

Explanation:
- slice1 extracts characters from index 0 to index 4 (inclusive), giving us "Hello".
- slice2 starts from index 7 and includes all characters until the end of the string, giving us "World!".
- slice3 starts from the beginning of the string and goes up to index 4 (inclusive), giving us "Hello".
- slice4 uses negative indexing to start from the 6th character from the end and includes all characters until the end, giving us "World!".
- slice5 uses a step of 2, which means it skips every other character from the slice, giving us "Hlo ol".

Negative indexing
In Python, negative indexing allows you to access elements in a sequence (like strings, lists, tuples) from the end of the sequence using negative numbers. The index -1 refers to the last element, -2 refers to the second-to-last element, and so on.

Example
Code: https://pastebin.com/T3ZfSzza

Explanation: In the example above, -1 represents the last character in the string "Hello, World!" which is "!". Similarly, -2 represents the second-to-last character which is "d". When using negative indexing with slicing, the start and end indices are also counted from the end of the string. So, slice1 is "World" since it starts from the sixth-to-last character ("W") and goes up to the second-to-last character ("d").

Modifying Strings
In Python, strings are immutable, which means you cannot modify them directly once they are created. However, you can perform operations on strings to create new strings with the desired modifications. Since you cannot change individual characters in a string directly, you can create new strings by combining or slicing the original string, and then assign the modified string to a new variable.

upper()
upper() is a string method used to convert all the characters in a string to uppercase. It returns a new string with all uppercase letters while leaving any non-alphabetic characters unchanged.
Code: https://pastebin.com/ThKj1JAi

lower()
The lower() is a string method used to convert all the characters in a string to lowercase. Just like the upper() method, the lower() method does not modify the original string (text) but instead creates a new string with all lowercase characters. This behavior is due to the immutability of strings in Python.
Code: https://pastebin.com/JtyiEMyE

strip()
strip() is a string method used to remove leading and trailing whitespace characters from a string. By default, it removes spaces, tabs, and newline characters from the beginning and end of the string. It returns a new string with the leading and trailing whitespace removed.
Code: https://pastebin.com/L4viAxRW

replace()
replace() is a string method used to replace occurrences of a substring with a new substring within a string. It returns a new string with the replacements made, leaving the original string unchanged.

Syntax:
string.replace(old_substring, new_substring, count)

- old_substring: The substring you want to replace in the original string.
- new_substring: The new substring that will replace occurrences of the old_substring.
- count (optional): Specifies the maximum number of occurrences to replace. If not specified, all occurrences will be replaced.

Code: https://pastebin.com/HAanKM0X

split()
split() is a string method used to split a string into a list of substrings based on a specified delimiter. It allows you to break a string into smaller parts whenever the delimiter is encountered. The split() method returns a list of these substrings.
Syntax:
string.split(sep, maxsplit)

- **sep (optional):** The delimiter that specifies where the string should be split. If not provided, the default delimiter is whitespace (spaces, tabs, and newlines).
- **maxsplit (optional):** Specifies the maximum number of splits to be performed. If not provided, there is no limit to the number of splits.

Code: https://pastebin.com/8QJHwwcM

capitalize()
capitalize() is a string method used to capitalize the first character of a string. It converts the first character to uppercase and leaves the rest of the characters unchanged. If the first character is already in uppercase, the method does not alter it.
Code: https://pastebin.com/cpYeB0UU

Concatenation
Concatenation in Python refers to the process of combining two or more strings together to form a single string. This can be achieved using the + operator or by using string formatting methods like str.format() or f-strings (formatted string literals).
Code: https://pastebin.com/WCaBWM04

format()
format() is a string method used for string formatting. It allows you to insert values into a string in a structured and controlled way. The format() method uses curly braces {} as placeholders, which are replaced with the values passed as arguments to the format() method.
Code: https://pastebin.com/dy4JzzES

Question:
quantity = 3
item_number = 567
price = 49.95
Replace the variables in this string with their values and print the output string: I want quantity pieces of item_number item for price dollars.
Code: https://pastebin.com/nixS1Ygq

Escape characters
Escape characters are special characters that are used to represent certain non-printable or reserved characters within a string. They are represented by a backslash (\) followed by a character or a combination of characters. Escape characters enable you to include characters in a string that would otherwise be difficult or impossible to represent directly.

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

Example Code: https://pastebin.com/FxTjQLf2

Question:
Consider a string: 'The unexpected always happens".

1. Assign this string to a variable: text.
2. Print the string.
3. Print the length of the string.
4. Check if the phrase 'pp' is present in the string.
5. Print the substring from 0 till 10th index.
6. Replace 'always' with 'never'.
7. Add "no matter what" to the string.
8. Print the final string.

Code: https://pastebin.com/NTJCDaHE

Question: Write a Python function that checks if the given string is a palindrome or not.
Input: mama
Output: True
Code: https://pastebin.com/iR2xRCmT

Question: Write a Python function that replaces all commas with dots and all dots with commas in the given string.
Input : 14, 625, 498.002
Output : 14.625.498, 002
Code: https://pastebin.com/KgUKXt7M

MCQ Time

1. What will be the output of the following Python code?

```
print("Hello {name1} and {name2}".format(name1='foo', name2='bin'))
```

a. Hello foo and bin
b. Hello {name1} and {name2}
c. Error
d. Hello and

2. What will be the output of the following code snippet?

```
str1="6/4"
print("str1")
```

a. 1
b. 6/4
c. 1.5
d. str1

3. What will be the output of the following code snippet?

```
str1="Information"
print(str1[2:8])
```

a. format
b. formatio
c. orma
d. ormat

4. What will be the output of the following code snippet?

```
str1="Application"
str2=str1.replace('a','A')
print(str2)
```

a. application
b. Application
c. ApplicAtion
d. applicAtion

5. What will be the output of the following code snippet?

```
str1="poWer"
str1.upper()
print(str1)
```

a. POWER
b. Power
c. power
d. poWer

Answers:

1. a) Hello foo and bin
2. d) str1
3. a) format
4. c) ApplicAtion
5. d) poWer

Chapter -6

Introduction to OOPs

What is Object Oriented Programming?

Object-Oriented Programming (OOP) is a way of organizing and designing computer programs to make them more organized, easier to understand, and reusable. At its core, OOP is about modeling real-world objects and their interactions in a computer program.

Why use OOP in Python?

Using Object-Oriented Programming in Python provides numerous advantages, including code reusability, organization, and maintainability. By leveraging OOP principles, you can design robust and flexible software systems that mimic real-world entities and interactions, making it easier to develop and maintain complex projects.

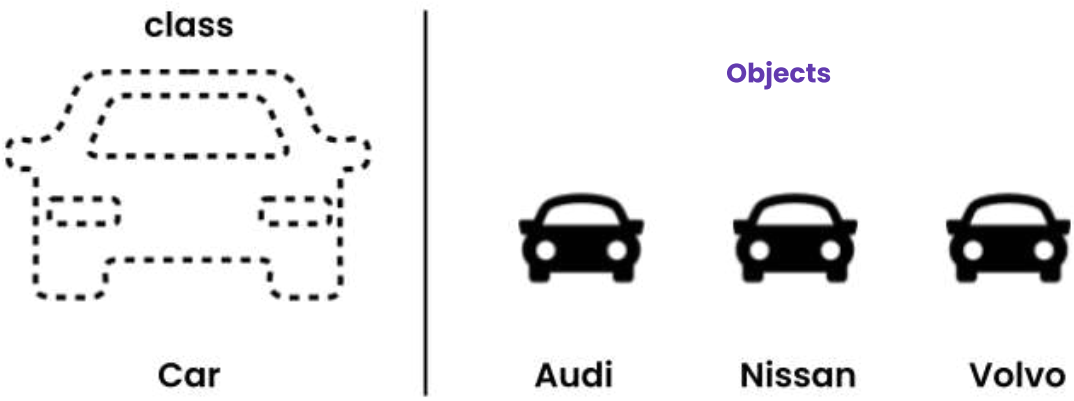
Classes and Objects

Classes:
In OOP, Classes are like user-defined data types, allowing you to encapsulate data (attributes) and operations (methods) that are related to a specific concept or entity. A class is defined using the class keyword in Python.

For example, if you were creating a program for managing cars, you could define a "Car" class. This class would specify attributes like "make," "model," "color," and methods like "accelerate" and "brake."

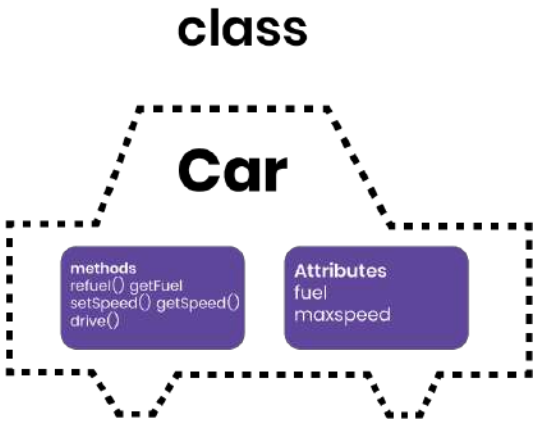
Objects:
An object is an instance of a class. It represents a specific occurrence of the class and contains the actual data defined in the class's attributes. In other words, an object is a concrete representation of the abstract blueprint provided by the class.

Example: An object of the "Car" class could be a particular car, such as an Audi, Nissan, and Volvo. Each car object would have its own unique attribute values, separate from other car objects created from the same class.



Methods and Attributes

In a class, you can have both functions (methods) and variables (attributes). Functions in a class are methods, and they define the behavior or actions that objects of that class can perform. Variables in a class are attributes, and they represent the data associated with the objects of that class.



<<<FOR EDITOR: here after getFuel add () just like others>>>>>>>>>>

Attributes:

Class attributes are shared among all instances (objects) of the class. They are defined within the class, outside any method, and are the same for all objects. Class variables are accessed using the class name (ClassName.class_variable) or through an instance of the class (self.class_variable).

Methods:

Methods are functions defined within the class. They operate on the attributes and can perform various actions. Methods have self as their first parameter, representing the object calling the method.

Using Methods and Accessing Attributes:

Once you have created an object, you can call its methods and access its attributes using dot notation (object_name.method_name() and object_name.attribute_name).

The 4 Pillars of Object-Oriented Programming

The four pillars of Object-Oriented Programming (OOP) are the four fundamental principles that guide the design and implementation of object-oriented systems. They are:

Encapsulation:

Encapsulation is the principle of bundling data (attributes) and methods (functions) that operate on that data within a single unit called a class. It allows the class to control the access to its internal data, ensuring that the data is not directly accessible from outside the class.

Abstraction:

Abstraction is the process of simplifying complex systems by representing the relevant characteristics and behaviors without exposing the underlying implementation details. In OOP, abstraction is achieved by creating abstract classes or interfaces that define the structure and behavior of objects without specifying how those details are implemented.

Abstraction allows developers to focus on what an object does rather than how it does it.

Inheritance:

Inheritance is the ability of a class to inherit attributes and methods from its parent or base class. It allows you to create a hierarchy of classes, where subclasses (child classes) can inherit and extend the functionality of their parent class. Inheritance promotes code reusability.

Polymorphism:

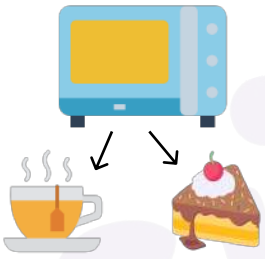
Polymorphism means "many forms" and refers to the ability of objects to take on multiple forms or be treated as instances of their parent class. Polymorphism simplifies code by promoting generic programming and supporting the principle of "write once, use anywhere."
In simpler terms, polymorphism allows you to use a single method or function name to perform different operations based on the context or type of data it operates on.

In Real Life...

Abstraction

Encapsulation

Inhertance



Polymorphism

Creating Classes and Objects

Defining a Class

A class in Python involves creating a blueprint or template for creating objects of that class. It defines the structure, attributes, and behaviors of objects that will be instantiated from the class.

Syntax:

```
class ClassName:
    # Class attributes (optional)
    class_attribute1 = value1
    class_attribute2 = value2

    # Class methods (optional)
    def class_method1(self, parameter1, parameter2, ...):
        # Method logic goes here

    def class_method2(self, parameter1, parameter2, ...):
        # Method logic goes here
```

Explanation of the parts:

class ClassName:

This is the beginning of the class definition, where ClassName is the name of the class. The class body, containing class attributes and methods, is indented below this line.

Class attributes (optional):
Inside the class body, you can define class attributes (variables) if needed. Class attributes are shared among all instances (objects) created from the class and are accessed using the class name (ClassName.attribute).

Class methods (optional):
You can define class methods (functions) within the class to specify the behavior of the class. Class methods are similar to regular functions, but they operate on the class itself and can access class attributes. Class methods have the self parameter as the first parameter to access class attributes and other methods.

Instantiating Object

It involves creating instances of a class using the class name followed by parentheses. When you create an instance of a class, you are creating a new object that is based on the blueprint provided by the class.

Syntax:

```
# Creating an object (instance) of a class
object_name = ClassName()
```

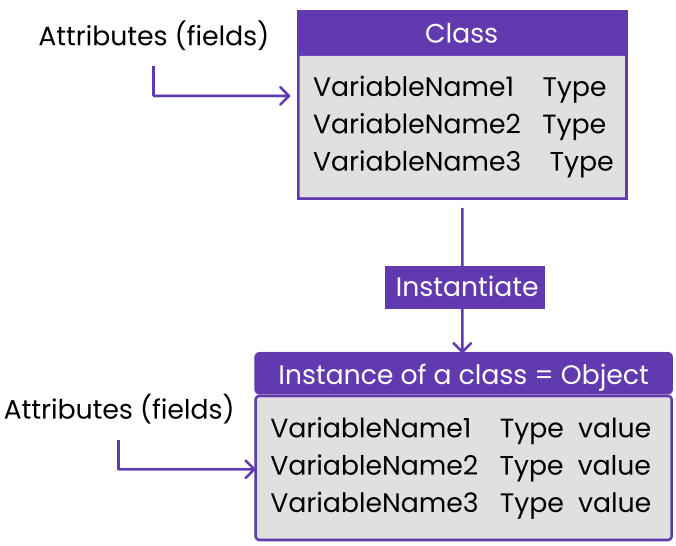
Explanation of the parts:

- object_name: This is the name you choose for the object, which is a reference to the instance of the class you are creating.
- ClassName: This is the name of the class you want to instantiate. It should be a valid Python identifier.

Once you have created an object, you can use it to access the class's attributes and methods.

Instance attributes

Instance attributes are variables or data members that belong to individual instances of a class in object-oriented programming. Each object created from a class has its own set of instance attributes, which can hold unique values or states specific to that particular instance. These attributes define the characteristics and properties of the object and can be accessed and manipulated using the instance of the class. Instance attributes are typically initialized within the class's constructor method and are used to represent the varying characteristics or data associated with each object.



Class Attributes vs. Instance Attributes:

Class attributes and instance attributes are fundamental concepts in object-oriented programming that define the characteristics and behavior of objects within a class. They serve distinct roles in defining the structure and behavior of objects.

Aspect	Class Attributes	Instance Attributes
Definition	Associated with the class itself.	Associated with each individual instance.
Scope	Shared among all instances of the class.	Unique to each instance of the class.
Access	Accessed using the class name.	Accessed using the instance name.
Initialization	Typically defined within the class body.	Initialized within the class constructor.
Modification	Can be modified dynamically for all instances.	Can be modified for individual instances.
Purpose	Used for data shared across instances.	Used for data specific to an instance.
Example Use Case	A 'counter' attribute to count instances.	A 'name' attribute to store a person's name.

Example:
Write a Python class named Rectangle to represent a rectangle shape. The class should have the following functionalities:

- A method named set_dimensions that takes two parameters width and height and sets the attributes of the rectangle object accordingly.
- A method named area that calculates and returns the area of the rectangle
- A method named perimeter that calculates and returns the perimeter of the rectangle

Use this to create objects of the class and print the width, height, area and perimeter.

Code: https://pastebin.com/bQ4WLJkY

Explanation:

1. We define the Rectangle class where we have a method named set_dimensions() that takes width and height as arguments and sets the instance variables' width and height accordingly.
2. The area() and perimeter() methods calculate the area and perimeter of the rectangle based on its attributes.
3. We create an object rectangle1 of the Rectangle class.
4. We set the dimensions of rectangle1 using the set_dimensions() method.
5. We access the attributes of rectangle1 (i.e., width and height) using dot notation and call its methods (i.e., area() and perimeter()) to perform calculations.

Class Constructor

Class constructors in Python are special methods that are automatically called when you create an instance (object) of a class. The constructor method is denoted by __init__, and it allows you to initialize the attributes of the object with values passed as arguments during instantiation. Constructors are essential for setting up the initial state of the object, ensuring that it is in a valid and usable state when it is created.

Syntax:

```
class ClassName:
    def __init__(self, parameter1, parameter2, ...):
        # Initialize instance variables (attributes) here
```

Explanation of the parts:

- def __init__(self, parameter1, parameter2, ...):

This is the special constructor method for the class. The __init__ method is automatically called when you create an instance of the class. The first parameter self refers to the instance of the object being created, and it allows you to access and modify the object's attributes.

- parameter1, parameter2, ...:

These are the parameters of the constructor that you expect to receive when creating an object. They represent the initial values that you want to assign to the object's attributes.

Inside the constructor, you can use the self parameter to set the initial state of the object by assigning values to its attributes. This way, each instance of the class can have its own unique attribute values.

Example:

Create a Python class named Person to represent a person's information. The class should have the following features:

- A constructor __init__ that takes two parameters name and age and initializes the attributes of the person object accordingly.
- A method named say_hello that prints a greeting message, displaying the person's name and age.

Once you have defined the Person class, create two objects to demonstrate the working.

Code: https://pastebin.com/edS8Tf7B

In this example, the Person class has a constructor that takes name and age as parameters. During object creation (person1 and person2), we pass the corresponding arguments, and the constructor initializes the name and age attributes of the objects with those values. The say_hello() method then uses the attribute values to display a greeting for each person object.

Access Modifiers

In Python, access modifiers are used to control the visibility and accessibility of class attributes and methods. They determine how the attributes and methods can be accessed from outside the class or within the class hierarchy. Python provides three types of access modifiers:

1. Public
2. Protected
3. Private

Public Access Modifier

In Python, there is no strict implementation of public access modifiers. By default, all attributes and methods defined in a class are considered public, and they can be accessed from both inside and outside the class.

Syntax:

```
class MyClass:
    def public_method(self):
        # This is a public method
        pass

    def __init__(self):
        self.public_attribute = 10  # This is a public attribute
```

Both public_method() and public_attribute are accessible from outside the class.

Protected Access Modifier (_)

In Python, an attribute or method prefixed with a single underscore _ is considered protected, but it is still accessible from outside the class. However, it is a convention that indicates the attribute or method is intended for internal use within the class or its subclasses and should not be accessed directly from outside the class.

```
class MyClass:
    def _protected_method(self):
        # This is a protected method
        pass

    def __init__(self):
        self._protected_attribute = 20  # This is a protected attribute
```

Although _protected_method() and _protected_attribute can be accessed from outside the class, it is recommended to use them only within the class or its subclasses.

Private Access Modifier (__)

In Python, an attribute or method prefixed with a double underscore __ is considered private. It is not directly accessible from outside the class, and attempting to access it will raise an AttributeError.

```
class MyClass:
    def __private_method(self):
        # This is a private method
        pass

    def __init__(self):
        self.__private_attribute = 30  # This is a private attribute
```

Both __private_method() and __private_attribute cannot be accessed directly from outside the class.

Getter and Setter Methods

Getter and setter methods, also known as accessor and mutator methods, are used in object-oriented programming to control access to the attributes of a class. They provide a way to get (retrieve) and set (modify) the values of private attributes indirectly, promoting encapsulation and data hiding.

Getter Method:

A getter method is used to retrieve the value of a private attribute. It is decorated with @property, and its name should be the same as the name of the attribute you want to get but without the leading underscore (for convention).

```
class MyClass:
    def __init__(self):
        self._private_attribute = 0

    @property
    def private_attribute(self):
        return self._private_attribute
```

In this example, the private_attribute is a private attribute with a leading underscore _. The private_attribute getter method allows you to access this private attribute's value without exposing it directly.

Setter Method:

A setter method is used to modify the value of a private attribute. It is decorated with @<attribute_name>.setter, and its name should be the same as the name of the attribute you want to set but without the leading underscore (for convention).

```
class MyClass:
    def __init__(self):
        self._private_attribute = 0

    @property
    def private_attribute(self):
        return self._private_attribute

    @private_attribute.setter
    def private_attribute(self, value):
        self._private_attribute = value
```

In this example, we have defined a setter method for private_attribute, which allows you to set a new value to the _private_attribute indirectly.

Now, we can use these getter and setter methods to access and modify the private attribute:

```
obj = MyClass()
print(obj.private_attribute)   # Output: 0

obj.private_attribute = 42
print(obj.private_attribute)   # Output: 42
```

By using getter and setter methods, you can control how the private attributes are accessed and modified, and you can add additional logic or validation before allowing any changes to the attributes. This helps to maintain the integrity of the data and ensures that the class's internal state remains consistent.

Inheritance

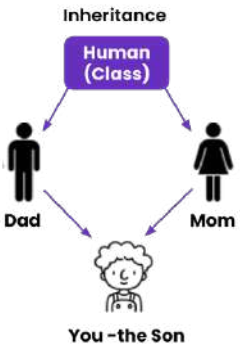
Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (called the subclass or derived class) to acquire the properties and behaviors of an existing class (called the superclass or base class). The subclass can reuse and extend the functionality of the superclass, promoting code reuse and creating a hierarchical relationship between classes.

Syntax:
class SuperClass:
  # Attributes and methods of the superclass go here

class SubClass(SuperClass):
  # Additional attributes and methods of the subclass go here

SuperClass:
This is the existing class (base class) from which you want to inherit attributes and methods.

SubClass:
This is the new class (derived class) that will inherit from the superclass. You define the subclass name, followed by the superclass name inside parentheses, to indicate inheritance.



<<Editor: Remove "The Son" from here>>

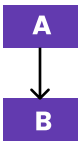
Example:
Code: https://pastebin.com/KNT4HskK

In this example, we have a base class Animal with an __init__ method setting the species attribute and a make_sound method. Two subclasses, Dog and Cat, inherit from Animal. They override the make_sound method to provide their own sounds. Inheritance allows reusing and extending functionality, and instances of Dog and Cat inherit the species attribute and methods from the base class.

Types of Inheritance

Inheritance in object-oriented programming can be classified into several types based on the relationship between the classes involved. The main types of inheritance are:

Single Inheritance:
Single inheritance involves one class (subclass) inheriting from a single superclass. In other words, a subclass can have only one direct parent class. It forms a linear hierarchy. Single inheritance is the simplest and most commonly used form of inheritance.
Syntax:
class SuperClass:
  pass
class SubClass(SuperClass):
  pass



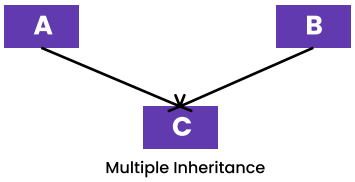
Single Inheritance

Multiple Inheritance:

Multiple inheritance involves one class inheriting from multiple superclasses. The subclass will have features from all the superclasses. Python supports multiple inheritance.

Syntax:

```
class SuperClass1:
    pass
class SuperClass2:
    pass
class SubClass(SuperClass1, SuperClass2):
    pass
```

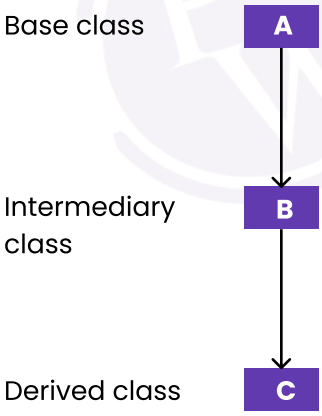


Multilevel Inheritance:

Multilevel inheritance involves one class inheriting from another class, and then another class inheriting from the second class. It creates a chain of inheritance.

Syntax:

```
class Grandparent:
    pass
class Parent(Grandparent):
    pass
class Child(Parent):
    pass
```

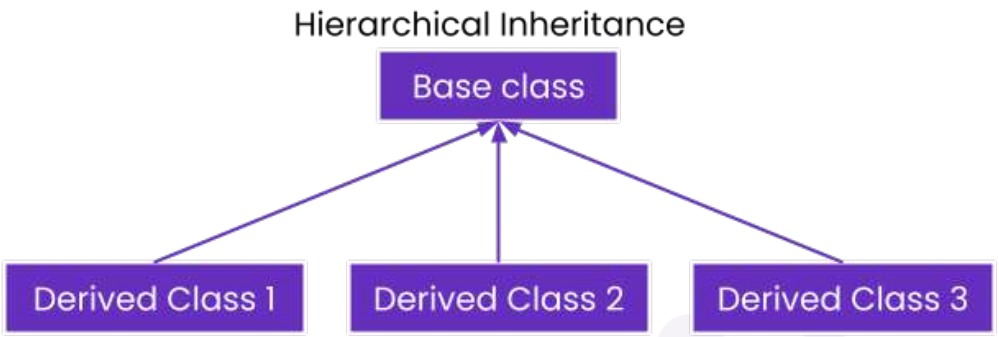


Hierarchical Inheritance:

Hierarchical inheritance involves one class (subclass) inheriting from a single superclass, while there are other subclasses that inherit from the same superclass. It forms a tree-like structure.

Syntax:

```
class SuperClass:
    pass
class SubClass1(SuperClass):
    pass
class SubClass2(SuperClass):
    pass
```

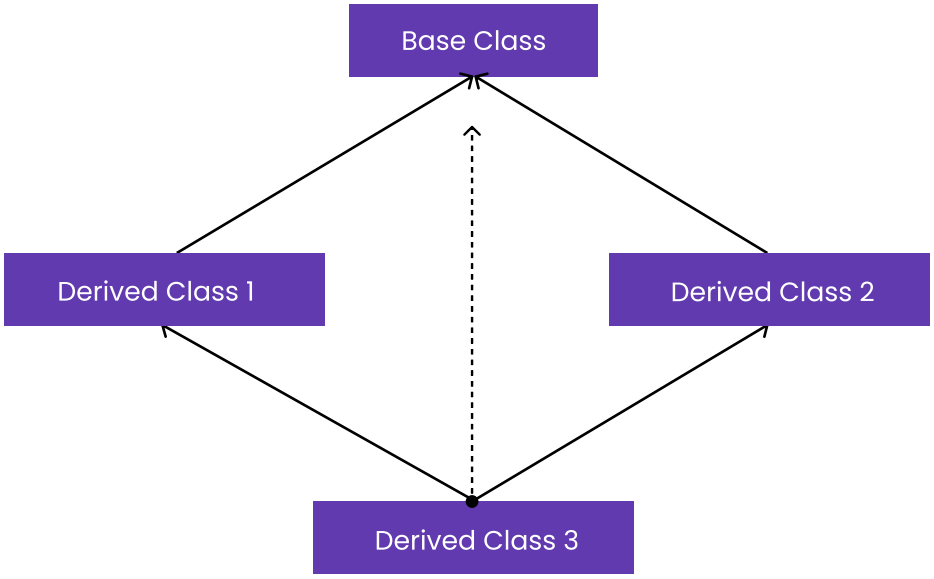


Hybrid (Multiple and Multilevel) Inheritance:

Hybrid inheritance involves a combination of multiple and multilevel inheritance. It can occur when a class inherits from multiple classes and also has subclasses inheriting from it.

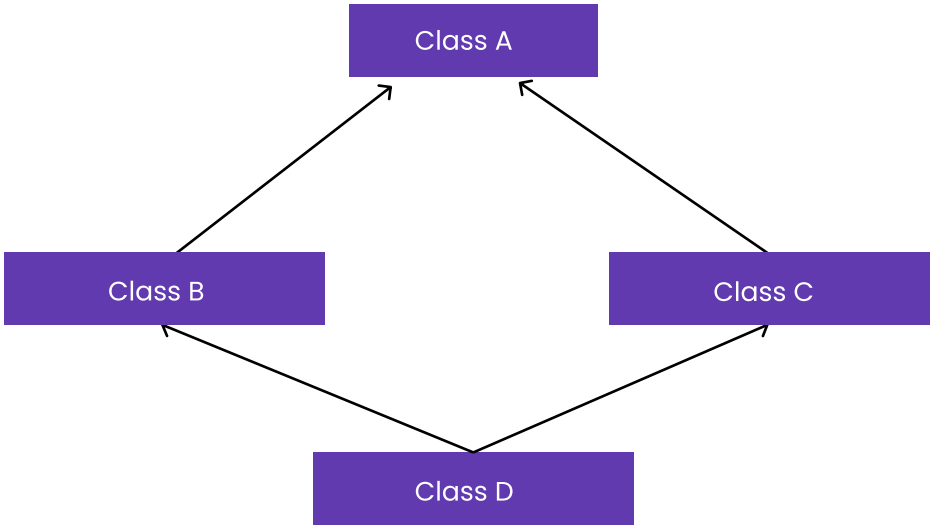
Syntax:

```
class SuperClass1:
    pass
class SuperClass2:
    pass
class SubClass1(SuperClass1, SuperClass2):
    pass
class SubClass2(SubClass1):
    pass
```



Diamond Problem

The "diamond problem" is a term used to describe an ambiguity that can occur in object-oriented programming languages with multiple inheritance and a certain type of class hierarchy structure.



In the above scenario:

- Class A is the base class.
- Classes B and C inherit from class A.
- Class D inherits from both classes B and C.

Now, imagine that classes B and C both have a method with the same name. When a method from class D tries to access that method, there's ambiguity because D inherits from both B and C, which both provide the same method through their common parent A.

This ambiguity can lead to issues like:

- Ambiguity in Method Calls: If class D tries to call the method inherited from A, it's unclear whether it should call the method from class B or class C.

- Diamond Problem: This ambiguity is referred to as the "diamond problem" because the class hierarchy looks like a diamond shape when visualized.

Solution:

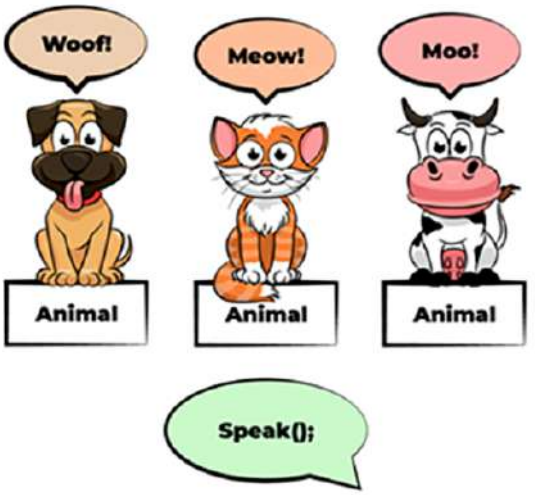
The diamond problem in Python is automatically solved by the Method Resolution Order (MRO) mechanism. Python uses the C3 Linearization algorithm to determine the order in which base classes are checked for method resolution. This mechanism ensures that the diamond problem is resolved without any extra steps required from the programmer.

Code: https://pastebin.com/tgKjFqS6

Polymorphism

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different data types or objects, providing flexibility and reusability in code.

Example:



Code: https://pastebin.com/uK9hjxAp

In this example, the base class Animal has an abstract method speak(), meant to be overridden by its subclasses Dog, Cat, and Cow, which inherit from Animal. When instances of these subclasses call the speak() method, polymorphism allows the same method name to be used for different objects, invoking the appropriate implementation based on the object's type. This uniform treatment of different classes enables code reuse and flexibility in handling diverse behaviors.

There are two main types of polymorphism:

Compile-time Polymorphism (Static Polymorphism):
This type of polymorphism is resolved during compile-time. It is achieved through method overloading or operator overloading. Method overloading allows a class to have multiple methods with the same name but different parameters, while operator overloading allows defining custom behaviors for operators such as +, -, *, etc., for objects of a class.

```
# Method overloading
class MyClass:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

# Operator overloading
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexNumber(self.real + other.real, self.imag + other.imag)
```

Run-time Polymorphism (Dynamic Polymorphism):
This type of polymorphism is resolved during run-time. It is achieved through method overriding. Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in

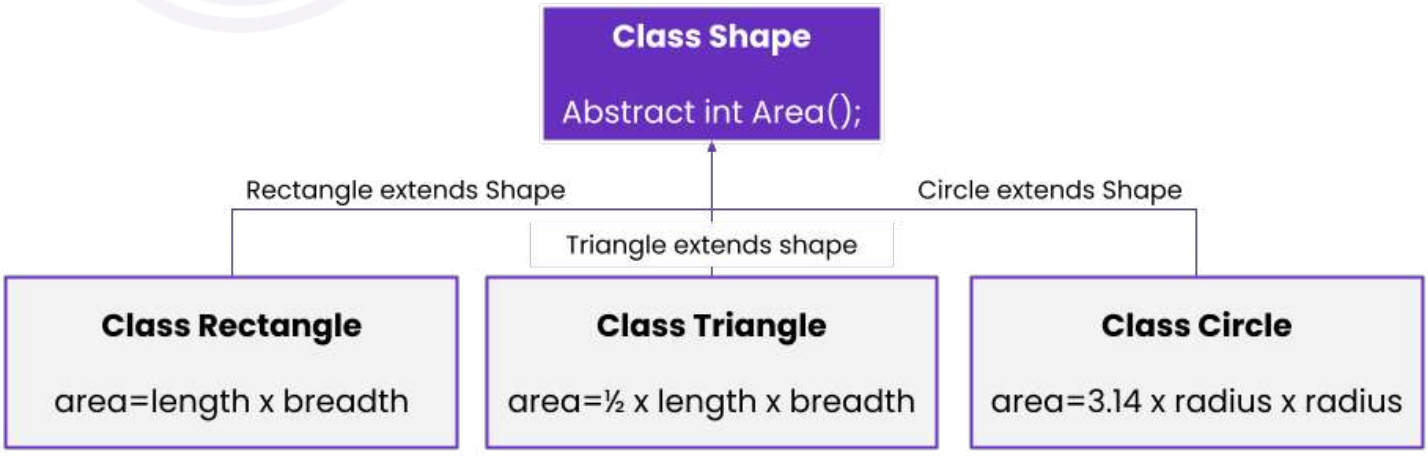
its superclass. The decision of which method to call is determined at run-time based on the actual object type.

```
class Shape:
    def draw(self):
        return "Drawing a generic shape."

class Circle(Shape):
    def draw(self):
        return "Drawing a circle."

class Square(Shape):
    def draw(self):
        return "Drawing a square."

# Creating instances of the subclasses
circle = Circle()
square = Square()

# Polymorphism in action: calling the draw() method on different objects
print(circle.draw())  # Output: Drawing a circle.
print(square.draw())  # Output: Drawing a square.
```

Polymorphism allows code to be more generic, as a single interface can handle different types of objects, providing flexibility and extensibility. It promotes code reuse, simplifies maintenance, and enables the implementation of abstract behaviors through interfaces or base classes.

Abstraction

Abstraction is a fundamental principle in object-oriented programming (OOP) that allows you to create a simplified representation of real-world objects by focusing on their essential characteristics while hiding unnecessary details. It is the process of defining the essential features of an object or class, abstracting away the implementation complexities, and providing a clear and simplified interface for the user.

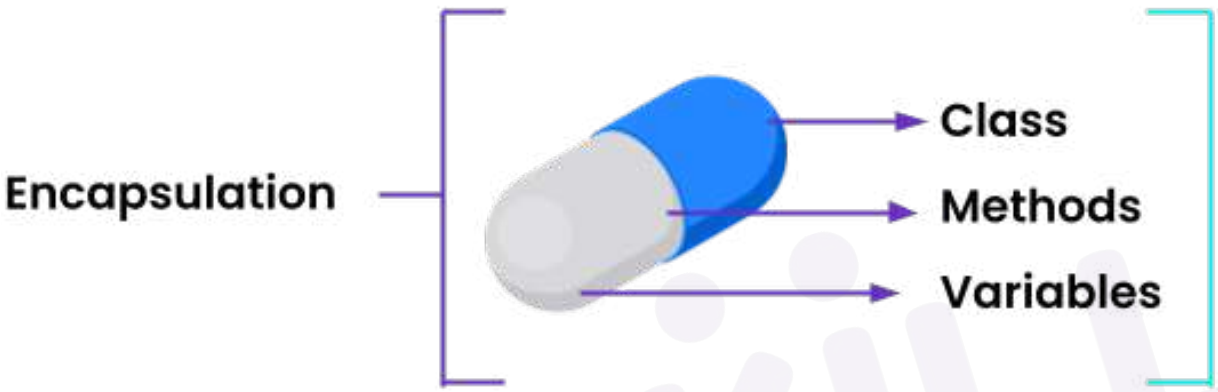


Example:
Code: https://pastebin.com/H6iv9qCT

In this example, the Shape class is an abstract class with abstract methods area() and perimeter(). The Circle and Square classes inherit from Shape and provide concrete implementations for the abstract methods. By using abstraction, we define the common interface for all shapes, allowing users to work with shapes without worrying about the internal details of each specific shape.

Encapsulation

Encapsulation is a fundamental principle in object-oriented programming (OOP) that bundles data (attributes) and the methods (functions) that operate on that data within a single unit called a class. It is the concept of wrapping the data and methods together to hide the internal details of how the class works from the outside world. The encapsulated data can only be accessed and modified through the methods defined in the class, which act as a protective barrier or interface.



The key aspects of encapsulation are:

- Data Hiding: Encapsulation hides internal data, making it private or protected, and only accessible through public methods.
- Access Control: Encapsulation allows the class to define which attributes and methods are public, private, or protected, controlling external access.
- Data Integrity: Encapsulation ensures data consistency and validity by providing controlled access through methods, allowing for validation and error checking.

Example:
Code: https://pastebin.com/Nwefqq1L

The BankAccount class encapsulates account number and balance attributes. deposit() and withdraw() methods provide controlled access to the private __balance attribute, ensuring data integrity. Python uses name mangling (_BankAccount__balance) to make it less accessible but still private. It should not be accessed directly outside the class.

Exception Handling in OOPs

Exception handling in object-oriented programming (OOP) is the process of dealing with unexpected or exceptional situations that may occur during the execution of code within classes and objects. In Python, exception handling is achieved using the try, except, and optionally finally blocks.

In OOP, exception handling can be used to:

- Handle Errors: Catch and handle errors that occur during the execution of methods in classes.
- Ensure Data Integrity: Prevent data corruption or invalid states by handling exceptions gracefully.
- Provide Customized Responses: Respond to specific error scenarios with custom messages or actions.

Syntax:

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
finally:
    # Code that will be executed regardless of whether an exception occurred or not
```

Explanation

try: The "try" block contains the code that you want to execute, which might raise an exception. If an exception occurs within this block, the control will immediately move to the corresponding "except" block. If no exception occurs, the "except" block will be skipped.

ExceptionType: This is the type of exception that you want to catch and handle. You can specify the specific exception type you want to catch, such as "ValueError," "TypeError," "ZeroDivisionError," or use the more general "Exception" to catch any exception.

except: The "except" block contains the code that will be executed if an exception of the specified type occurs in the "try" block. If no exception occurs, the "except" block will be skipped.

finally: The "finally" block contains code that will always be executed, whether an exception occurred or not. It is commonly used for cleanup operations, such as closing files or releasing resources.

Example: Implement exception handling in python by showing division operation. You can show exception - "ZeroDivisionError"

Input:
a = 10
b = 2
Output:
Cleanup: Division operation completed.
5

Input:
a = 5
b = 0
Output:
Error: Cannot divide by zero.
Code: https://pastebin.com/Ruc7UWuH

In this example, the divide_numbers() function attempts to divide a by b. If b is zero, it will raise a ZeroDivisionError, and the control will move to the except block, which prints an error message and assigns None to the result. Regardless of whether an exception occurred or not, the finally block will be executed, printing a cleanup message.

MCQ

1. **What is the purpose of the "init" method in a class in Python?**

a) It is used to define private methods within a class.
b) It is automatically called when an object is created and allows you to initialize its attributes.
c) It is used to define class variables that are accessible by all instances of the class.
d) It is used to define the inheritance hierarchy between classes.

2. What is the concept in OOP that allows a class to inherit attributes and methods from another class?
a) Polymorphism
b) Encapsulation
c) Abstraction
d) Inheritance

3. Which of the following statements about abstract classes in Python is true?
a) An abstract class cannot have any methods.
b) An abstract class can be instantiated to create objects.
c) An abstract class can be inherited but cannot be subclassed.
d) An abstract class may contain abstract methods, which must be implemented by its concrete subclasses.

4. What is the purpose of the "super()" function in Python when working with inheritance?
a) It calls the constructor of the base class.
b) It allows the child class to override the methods of the base class.
c) It initializes the private attributes of the child class.
d) It is used to create a new instance of the child class.

Answers

1. b) It is automatically called when an object is created and allows you to initialize its attributes.
2. d) Inheritance
3. d) An abstract class may contain abstract methods, which must be implemented by its concrete subclasses.
4. a) It calls the constructor of the base class.

Practice Questions

Question: Implementation of Encapsulation

Create a Python class to store and manage patient information in a hospital. Implement the Patient class with encapsulation to ensure that sensitive information, such as the patient's medical history, is hidden from direct access. The class should have methods to set and retrieve the patient's name, age, and contact information while keeping the medical history private.
Example:
Patient Name: John Doe
Patient Age: 35
Patient Contact: john@example.com
Medical History: ['Diagnosed with flu', 'Fractured arm after an accident']

Code: https://pastebin.com/mNGCRJDR

Explanation:

1. Private attributes (__name, __age, __contact, and __medical_history) are marked with double underscores, making them inaccessible from outside the class.
2. Setter methods (set_name, set_age, set_contact) are used to modify the private attributes safely from within the class.

3. Getter methods (get_name, get_age, get_contact) allow controlled access to the private attributes, providing read-only access to external code.
4. The medical history (__medical_history) is kept private and can only be updated using the add_medical_record method.
5. Encapsulation ensures data protection and restricts direct manipulation of sensitive information, promoting better code organization and security.

Time Complexity:
The time complexity of the Patient class methods is generally O(1) since most of the methods involve simple attribute assignments or return operations. The operations performed in the methods, such as setting and retrieving patient details and medical history, do not depend on the size of any input data.

Space Complexity:
The space complexity of the Patient class is O(1) for each instance of the class. The class attributes (__name, __age, __contact, and __medical_history) have fixed memory requirements for each instance, regardless of the number of medical records. Additionally, the methods do not use any auxiliary data structures or recursive calls that would increase the space usage with input size.

Question on Inheritance

Create a Bus child class that inherits from the Vehicle class. The default fare charge of any vehicle is seating capacity * 100. If Vehicle is Bus instance, we need to add an extra 10% on full fare as a maintenance charge. So total fare for bus instance will become the final amount = total fare + 10% of the total fare.
Example:
Vehicle Fare: 5000
Bus Fare: 5500.0

Code: https://pastebin.com/M0UG2019

Explanation:

- **Vehicle Class:** The Vehicle class is the parent class, and it has an instance variable seating_capacity that represents the seating capacity of the vehicle. The class has a method fare that calculates the fare based on the seating capacity, which is simply the seating capacity multiplied by 100.
- **Bus Class:** The Bus class is a child class that inherits from the Vehicle class. It has its constructor __init__ that calls the constructor of the parent class (Vehicle) using super().__init__(seating_capacity) to set the seating capacity of the bus.
- The Bus class overrides the fare method of the parent class to calculate the fare for a bus instance. It first calls the parent class's fare method to get the base fare based on the seating capacity. Then, it calculates a 10% maintenance charge based on the base fare and adds it to the base fare to get the final fare for the bus. This final fare with the maintenance charge is then returned.
- The Bus class extends the functionality of the Vehicle class by adding a maintenance charge to the base fare for bus instances, as required.

1. **Example Usage:** In the example usage section, two instances are created, one of the Vehicle class and one of the Bus class, with a seating capacity of 50.
    a. The first instance vehicle of the Vehicle class has a seating capacity of 50. The fare is calculated as 50 * 100, resulting in a fare of 5000.
    b. The second instance bus of the Bus class also has a seating capacity of 50. The fare calculation involves the fare method of the parent class, which calculates the total fare as 50 * 100 = 5000. Then, a 10% maintenance charge is added (5000 * 0.1 = 500), resulting in a final_fare of 5000 + 500 = 5500.

Time Complexity:

- The time complexity of each method in the Vehicle class is O(1) because they involve simple mathematical operations that don't depend on the size of any input.
- The fare method in the Bus class also has a time complexity of O(1). It calls the fare method of the superclass (i.e., Vehicle class), which has a constant time complexity, and then performs some additional constant-time operations (multiplication and addition). The number of passengers (seating_capacity) is not affecting the complexity.

Space Complexity:

- The space complexity of both classes is O(1) because they only have instance variables (self.seating_capacity) and local variables (total_fare and maintenance_charge), which occupy a constant amount of memory regardless of the input size.

Question:

Define two classes, India and USA. Each class has three methods: capital, language, and type. These methods print out information about the capital, language, and type of the country, respectively to demonstrate polymorphism.

The India class defines the following methods:

- capital() prints out the capital of India, which is New Delhi.
- language() prints out the most widely spoken language in India, which is Hindi.
- type() prints out the type of country India is, which is a developing country.

The USA class defines the following methods:

- capital() prints out the capital of the USA, which is Washington, D.C.
- language() prints out the primary language in the USA, which is English.
- type() prints out the type of country the USA is, which is a developed country.

The code then creates two objects, obj_ind and obj_usa, which are instances of the India and USA classes, respectively. The code then iterates through a tuple of these objects and calls the capital(), language(), and type() methods on each object.

Code: https://pastebin.com/QFXtfS06

Explanation:

1. The India and USA classes both have methods with the same names: capital(), language(), and type(). This is the basis for polymorphism.
2. When the for loop iterates over the objects obj_ind and obj_usa, it treats them as instances of the same base class (in this case, no explicit base class is defined, so they are treated as instances of the implicit base class object).
3. During runtime, the appropriate method is dynamically selected based on the actual type of the object being referred to at each iteration. This process is called method overriding.
4. As a result, when the country.capital(), country.language(), and country.type() calls are made inside the loop, the corresponding method of the specific class (either India or USA) is executed, displaying the appropriate output for each country.

Time Complexity:

Each method in both the India and USA classes involves only print statements, which are basic operations with a time complexity of O(1). The for loop iterates over two objects, one from each class, so the time complexity of the loop is O(2) or simply O(1) since it's a constant number of iterations.

Space Complexity:

The space complexity of the classes is O(1) since they only contain methods with print statements and do not use any additional data structures or variables that scale with the input.