What is Python

Python is a general-purpose, dynamically typed, high-level, compiled and interpreted, garbage-collected, and purely object-oriented programming language that supports procedural, object-oriented, and functional programming.

Features of Python:

- **Easy to use and Read -** Python's syntax is clear and easy to read, making it an ideal language for both beginners and experienced programmers. This simplicity can lead to faster development and reduce the chances of errors.
- **Dynamically Typed** - We do not need to specify the data type of a variable during writing codes. The data types of variables are determined during run-time.
- **High-level** - High-level language means human readable code.
- **Compiled and Interpreted** - Python code first gets compiled into bytecode, and then interpreted line by line.
- **Garbage Collected** - Memory allocation and de-allocation are automatically managed. Programmers do not specifically need to manage the memory.
- **Purely Object-Oriented** - It refers to everything as an object, including numbers and strings.

- **Cross-platform Compatibility** - Python can be easily installed on Windows, macOS, and various Linux distributions, allowing developers to create software that runs across different operating systems.
- **Open Source** - Python is an open-source, cost-free programming language. It is utilized in several sectors and disciplines as a result.

Python Variables

A variable is the name given to a memory location. A value-holding Python variable is also known as an identifier.

Since python is dynamically typed languages we do not need to specify the data type while writing the code , it will automatically assigns valid datatype during run time .

Variable names must begin with a letter or an underscore, but they can be a group of both letters and digits.

Python Is a case senestivite languages which means , Both Rahul and rahul are distinct variables.

Identifier Naming

- The variable's first character must be an underscore or alphabet (_).
- Every one of the characters with the exception of the main person might be a letter set of lower-case(a-z), capitalized (A-Z), highlight, or digit (0-9).
- White space and special characters (!, @, #, %, etc.) are not allowed in the identifier name. ^, &, *).
- Identifier name should not be like any watchword characterized in the language.

- Names of identifiers are case-sensitive; for instance, my name, and MyName isn't something very similar.
- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

Python Variable Types

There are two types of variables in Python - Local variable and Global variable. Let's understand the following variables.

Local Variable

The variables that are declared within the function and have scope within the function are known as local variables. Let's examine the following illustration.

Example -

```python
# Declaring a function
def add():
    # Defining local variables. They has scope only within a function
    a = 20
    b = 30
    c = a + b
    print("The sum is:", c)

# Calling a function
add()
```

Output:

```
The sum is: 50
```

Explanation:

We declared the function add() and assigned a few variables to it in the code above. These factors will be alluded to as the neighborhood factors which have scope just inside the capability. We get the error that follows if we attempt to use them outside of the function.

```python
add()
# Accessing local variable outside the function
print(a)
```

Output:

```
The sum is: 50
    print(a)
NameError: name 'a' is not defined
```

We tried to use local variable outside their scope; it threw the **NameError.**

Global Variables

Global variables can be utilized all through the program, and its extension is in the whole program. Global variables can be used inside or outside the function.

By default, a variable declared outside of the function serves as the global variable. Python gives the worldwide catchphrase to utilize worldwide variable inside the capability. The function treats it as a local variable if we don't use the global keyword. Let's examine the following illustration.

Example -

```python
# Declare a variable and initialize it
x = 101

# Global variable in function
def mainFunction():
    # printing a global variable
    global x
    print(x)
    # modifying a global variable
    x = 'Welcome To Javatpoint'
    print(x)

mainFunction()
print(x)
```

Output:

```
101
Welcome To Javatpoint
Welcome To Javatpoint
```

Explanation:

In the above code, we declare a global variable x and give out a value to it. We then created a function and used the global keyword to access the declared variable within the function. We can now alter its value. After that, we gave the variable x a new string value and then called the function and printed x, which displayed the new value.

=

ADVERTISEMENT

Delete a variable

We can delete the variable using the **del** keyword. The syntax is given below.

Syntax -

```
del <variable_name>
```

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error **"variable x is not defined"**. The variable x will no longer use in future.

Example -

```
# Assigning a value to x
x = 6
print(x)
# deleting a variable.
del x
print(x)
```

Output:

```
6
Traceback (most recent call last):
  File   "C:/Users/DEVANSH   SHARMA/PycharmProjects/Hello/multiprocessing.py",
line 389, in
    print(x)
NameError: name 'x' is not defined
```

Getting User Input in Python

In Python, the `input()` function is used to get input from the user. This function reads a line from input, converts it into a string (stripping a trailing newline), and returns that string.

Basic Example

Here is a simple example demonstrating how to use the `input()` function to get user input and then use that input in your program:

```
python
Copy code
# Prompt the user for their name
name = input("Enter your name: ")

# Prompt the user for their age
age = input("Enter your age: ")

# Print the input received
print("Hello, " + name + "! You are " + age + " years
old.")
```

Explanation

1. **Prompting for Input**:
   - `input("Enter your name: ")`: This function displays the prompt `Enter your name:` and waits for the user to type something and press Enter. The input is then returned as a string.
   - `name = input("Enter your name: ")`: The returned string is stored in the variable `name`.
   - `age = input("Enter your age: ")`: Similarly, the user's input for age is stored in the variable `age`.
2. **Printing the Input**:

- `print("Hello, " + name + "! You are " + age + " years old.")`: This concatenates the strings and variables to create a message that includes the user's input, which is then displayed.

Handling Different Data Types

By default, the `input()` function returns the input as a string. If you need to work with other data types, such as integers or floats, you must convert the input string to the desired type using functions like `int()` or `float()`.

Example with Integer Conversion

```
python
# Prompt the user for a number and convert it to an
integer
number = int(input("Enter a number: "))

# Perform a calculation (e.g., square the number)
squared = number ** 2

# Print the result
print("The square of " + str(number) + " is " +
str(squared) + ".")
```

Explanation

1. **Converting Input**:
   - `int(input("Enter a number: "))`: This first calls `input("Enter a number: ")` to get the user input as a string, and then `int()` converts that string to an integer.
2. **Performing Calculations**:
   - `squared = number ** 2`: This calculates the square of the number.
3. **Printing the Result**:
   - `print("The square of " + str(number) + " is " + str(squared) + ".")`: This concatenates the strings and variables, converting integers back to strings where necessary, to create a message displaying the result.

By using the `input()` function and the appropriate type conversions, you can effectively gather and use user input in your Python programs.

Python If-else statements

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the bloc the false case of the condition to be checked. If the condition provided in the if statement else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

Indentation in Python

For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

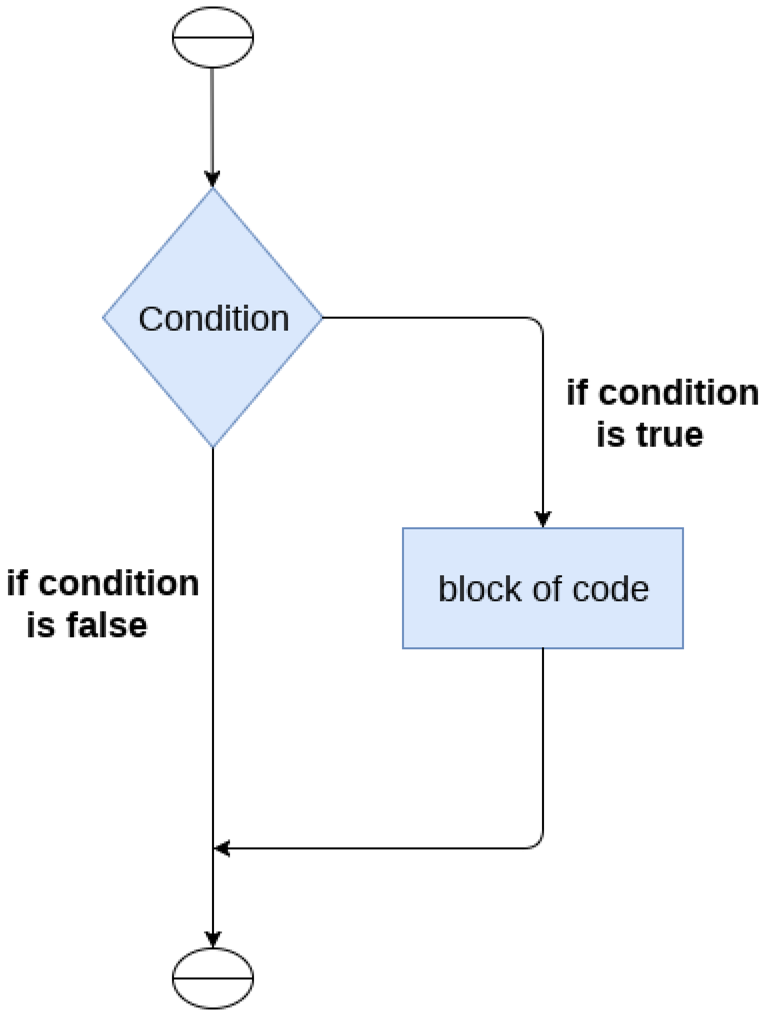
Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Backward Skip 10sPlay VideoForward Skip 10s

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

```
if expression:
    statement
```

Example 1

```
# Simple Python program to understand the if statement
num = int(input("enter the number:"))
# Here, we are taking an integer num and taking input dynamically
if num%2 == 0:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print("The Given number is an even number")
```

Output:

```
enter the number: 10
The Given number is an even number
```

Example 2 : Program to print the largest of the three numbers.

```
# Simple Python Program to print the largest of the three numbers.
a = int (input("Enter a: "));
b = int (input("Enter b: "));
c = int (input("Enter c: "));
if a>b and a>c:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print ("From the above three numbers given a is largest");
if b>a and b>c:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print ("From the above three numbers given b is largest");
if c>a and c>b:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print ("From the above three numbers given c is largest");
```

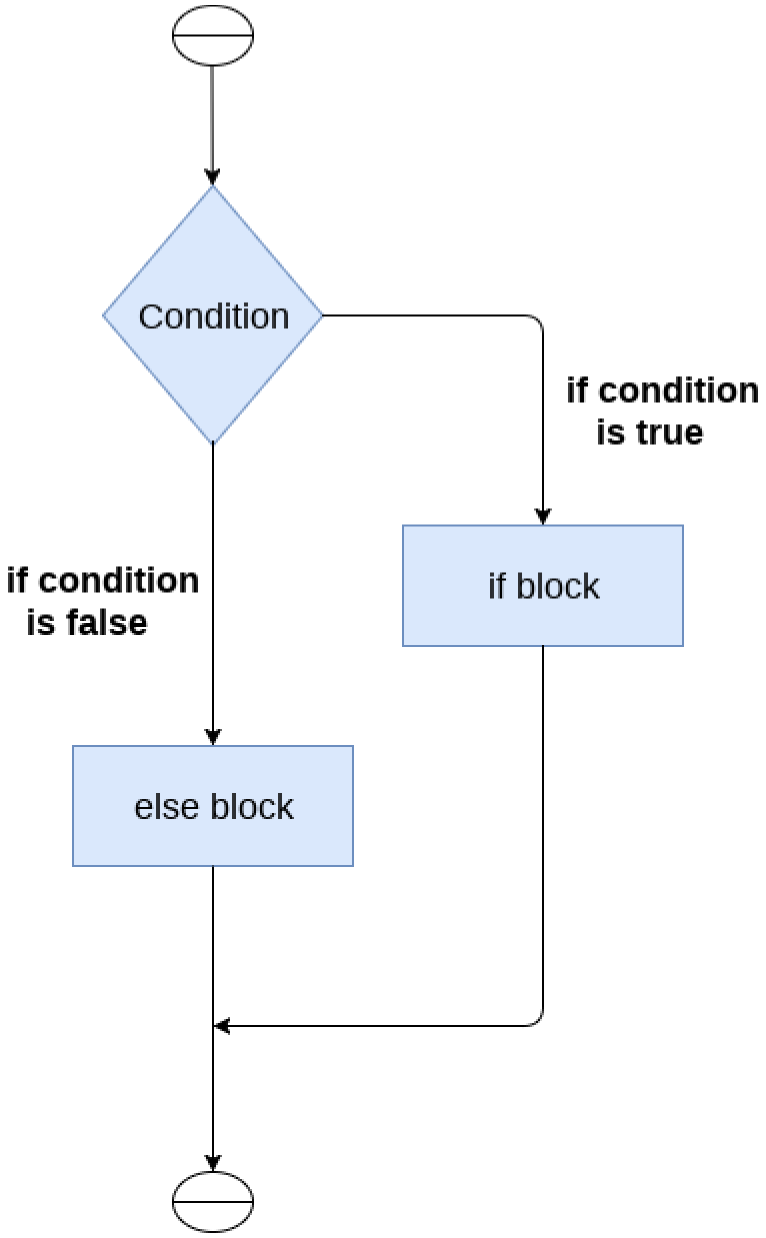
Output:

```
Enter a: 100
Enter b: 120
Enter c: 130
From the above three numbers given c is largest
```

The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below.

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```

Example 1 : Program to check whether a person is eligible to vote or not.

```
# Simple Python Program to check whether a person is eligible to vote or not.
age = int (input("Enter your age: "))
# Here, we are taking an integer num and taking input dynamically
if age>=18:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

Output:

```
Enter your age: 90
You are eligible to vote !!
```

Example 2: Program to check whether a number is even or not.

```
# Simple Python Program to check whether a number is even or not.
num = int(input("enter the number:"))
# Here, we are taking an integer num and taking input dynamically
if num%2 == 0:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print("The Given number is an even number")
else:
    print("The Given Number is an odd number")
```

Output:

```
enter the number: 10
The Given number is even number
```

The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

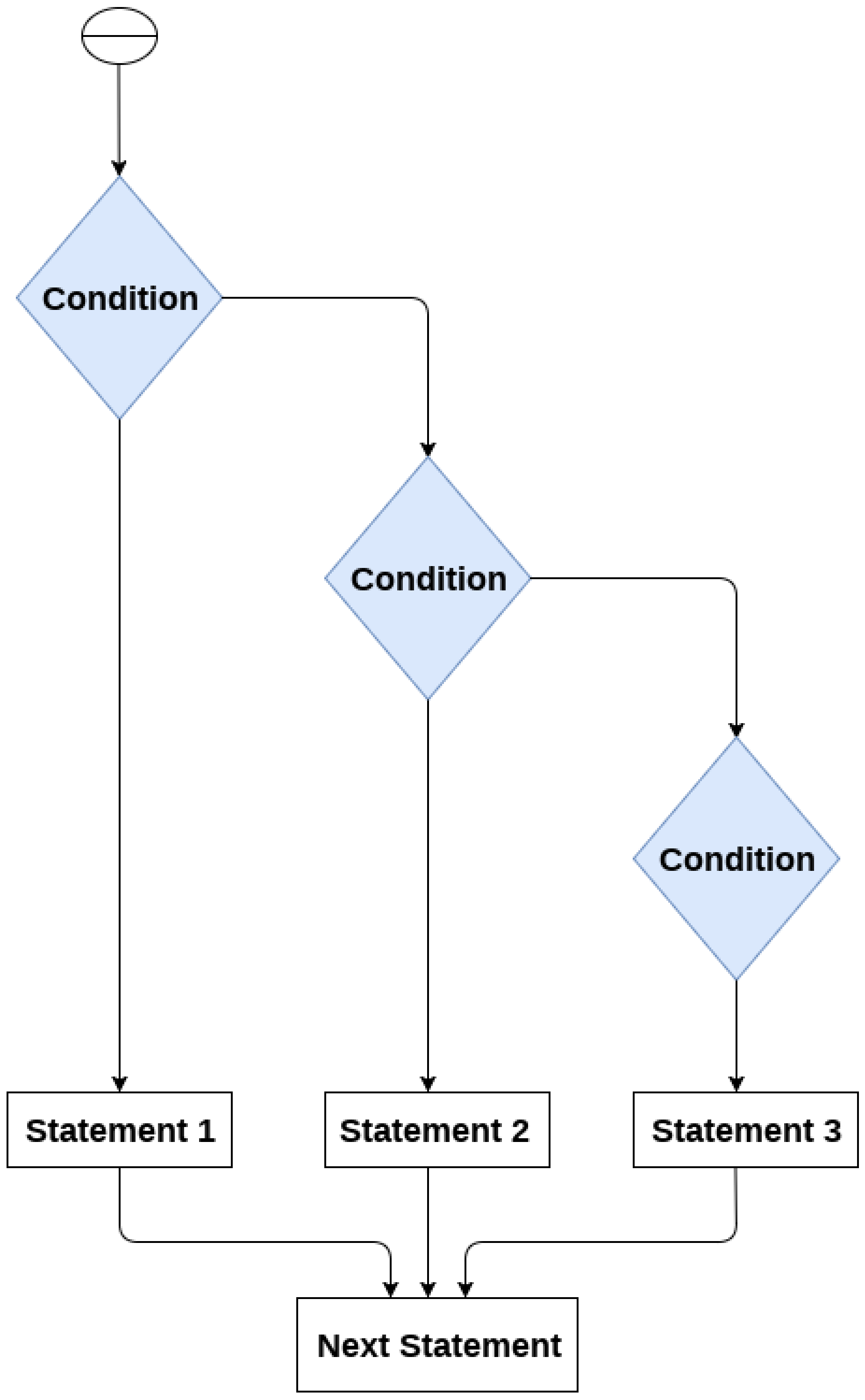
The syntax of the elif statement is given below.

```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```



Example 1

```
# Simple Python program to understand elif statement
number = int(input("Enter the number?"))
# Here, we are taking an integer number and taking input dynamically
```

```python
if number==10:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print("The given number is equals to 10")
elif number==50:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print("The given number is equal to 50");
elif number==100:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print("The given number is equal to 100");
else:
    print("The given number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15
The given number is not equal to 10, 50 or 100
```

Example 2

```python
# Simple Python program to understand elif statement
marks = int(input("Enter the marks? "))
# Here, we are taking an integer marks and taking input dynamically
if marks > 85 and marks <= 100:
# Here, we are checking the condition. If the condition is true, we will enter the block
   print("Congrats ! you scored grade A ...")
elif marks > 60 and marks <= 85:
# Here, we are checking the condition. If the condition is true, we will enter the block
   print("You scored grade B + ...")
elif marks > 40 and marks <= 60:
# Here, we are checking the condition. If the condition is true, we will enter the block
   print("You scored grade B ...")
elif (marks > 30 and marks <= 40):
# Here, we are checking the condition. If the condition is true, we will enter the block
   print("You scored grade C ...")
else:
   print("Sorry you are fail ?")
```

Output:

```
Enter the marks? 89
Congrats ! you scored grade A ...
```

Functions in Python

A function is a reusable block of code that performs a specific task.

Defining a Function

You use the `def` keyword to define a function.

```
python
Copy code
def greet(name):
    print(f"Hello, {name}!")
```

Calling a Function

You call a function by its name followed by parentheses, passing any required arguments.

```
python
Copy code
greet("Alice")  # Output: Hello, Alice!
```

Scoping

Scoping determines the visibility of variables. Python uses the LEGB rule: Local, Enclosing, Global, Built-in.

Scope Levels

- **Local**: Inside the current function.
- **Enclosing**: In the enclosing functions (nested functions).
- **Global**: At the top level of the module.
- **Built-in**: Names in the built-in scope (like `print`, `len`).

Example:

```
python
Copy code
x = "global"

def outer_function():
    x = "enclosing"

    def inner_function():
        x = "local"
        print(x)  # Output: local
    inner_function()
    print(x)  # Output: enclosing

outer_function()
print(x)  # Output: global
```

Arguments

Functions can take arguments to perform operations on them.

Positional Arguments

Arguments are passed in order.

```
python
Copy code
def add(a, b):
    return a + b

print(add(2, 3))  # Output: 5
```

Keyword Arguments

Arguments are passed with names.

```
python
Copy code
def add(a, b):
    return a + b

print(add(a=2, b=3))  # Output: 5
print(add(b=3, a=2))  # Output: 5
```

Default Arguments

Arguments with default values.

```
python
Copy code
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")  # Output: Hello, Alice!
greet("Bob", "Hi")  # Output: Hi, Bob!
```

Variable-length Arguments

Allows passing a variable number of arguments.

- **Arbitrary Positional Arguments**:

  ```
  python
  Copy code
  def fun(*args):
  ```

```
    for arg in args:
        print(arg)

fun(1, 2, 3)  # Output: 1 2 3
```

- **Arbitrary Keyword Arguments**:

```
python
Copy code
def fun(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

fun(name="Alice", age=30)  # Output: name: Alice age: 30
```

Return Values

Functions can return values using the `return` statement.

```
python
Copy code
def square(x):
    return x ** 2

result = square(4)
print(result)  # Output: 16
```

Advanced Function Calling

Lambda Functions

Small anonymous functions defined with `lambda`.

```
python
Copy code
square = lambda x: x ** 2
print(square(5))  # Output: 25

# Using lambda in higher-order functions
numbers = [1, 2, 3, 4]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16]
```

Higher-order Functions

Functions that accept other functions as arguments or return them.

```
python
Copy code
def apply_function(func, value):
    return func(value)

def increment(x):
    return x + 1

print(apply_function(increment, 5))  # Output: 6
```

Closures

Functions that remember variables from their enclosing scope.

```
python
Copy code
def outer_function(msg):
    def inner_function():
        print(msg)
    return inner_function

closure = outer_function("Hello, world!")
closure()  # Output: Hello, world!
```

Decorators

Functions that extend the behavior of other functions without modifying them.

```
python
Copy code
def decorator_function(original_function):
    def wrapper_function():
        print("Wrapper executed before", original_function.__name__)
        return original_function()
    return wrapper_function

@decorator_function
def display():
    print("Display function ran")

display()
# Output:
# Wrapper executed before display
# Display function ran
```

Summary

- **Functions**: Defined with `def`, take arguments, and return values.
- **Scoping**: Follows the LEGB rule.
- **Arguments**: Positional, keyword, default, and variable-length.
- **Advanced Concepts**: Lambda functions, higher-order functions, closures, and decorators.

Lists

- **Definition**: A list is a mutable, ordered collection of elements. Elements can be of different types, and lists are defined using square brackets.
- **Operations**:
    - **Creating a list**: Defining a list with initial values, e.g., `my_list = [1, 2, 3]`.
    - **Accessing elements**: Retrieving elements using their index, e.g., `my_list[0]` returns `1`.
    - **Adding elements**: Appending new elements to the end of the list with `my_list.append(element)`.

  - **Removing elements**: Deleting elements by value with `my_list.remove(value)` or by index with `del my_list[index]`.
  - **Slicing**: Extracting a portion of the list using a range of indices, e.g., `my_list[1:3]`.

Tuples

- **Definition**: A tuple is an immutable, ordered collection of elements. Tuples are defined using parentheses and cannot be changed once created.
- **Operations**:
  - **Creating a tuple**: Defining a tuple with initial values, e.g., `my_tuple = (1, 2, 3)`.
  - **Accessing elements**: Retrieving elements using their index, e.g., `my_tuple[0]` returns `1`.
  - **Immutability**: Tuples cannot be altered after their creation, meaning you cannot add, remove, or modify elements.
  - **Slicing**: Extracting a portion of the tuple using a range of indices, e.g., `my_tuple[1:3]`.

Sequences

- **Definition**: Sequences are an abstract data type that includes lists, tuples, and strings. They are ordered collections that support indexing and iteration.
- **Operations**:
  - **Iteration**: Looping through each element of the sequence, e.g., `for item in sequence: print(item)`.
  - **Length**: Getting the number of elements in the sequence with `len(sequence)`.
  - **Concatenation**: Combining two sequences into one using `sequence1 + sequence2`.
  - **Repetition**: Repeating the sequence multiple times with `sequence * n`.

Dictionaries

- **Definition**: A dictionary is a mutable, unordered collection of key-value pairs. Each key is unique and maps to a specific value. Dictionaries are defined using curly braces.
- **Operations**:
  - **Creating a dictionary**: Defining a dictionary with initial key-value pairs, e.g., `my_dict = {'key1': 'value1'}`.
  - **Accessing values**: Retrieving values using their keys, e.g., `my_dict['key1']` returns `'value1'`.
  - **Adding/Updating**: Adding new key-value pairs or updating existing ones with `my_dict[key] = value`.
  - **Removing items**: Deleting key-value pairs with `del my_dict[key]`.
  - **Iterating**: Looping through keys and values with `for key, value in my_dict.items():`.

Files

- **Definition**: Files are used to store data on disk. They can be read from or written to, and are handled using file objects.
- **Operations**:

    - **Opening a file**: Creating a file object with `open('filename.txt', 'mode')`, where `mode` can be `'r'` for reading, `'w'` for writing, `'a'` for appending, etc.
    - **Reading a file**: Extracting data from the file using methods like `file.read()`, which reads the entire content.
    - **Writing to a file**: Saving data to the file using methods like `file.write('text')`.
    - **Closing a file**: Ending the file operation with `file.close()`.

Using context manager: Automatically managing file opening and closing with `with open('filename.txt', 'mode') as file:`. `for` Loop

- **Definition**: Iterates over a sequence (such as a list, tuple, string, or range) and executes a block of code for each item in the sequence.
- **Syntax**:

```
python
Copy code
for item in sequence:
    # Code to execute
```

- **Examples**:
    - **Iterating over a list**:

    ```
    python
    Copy code
    fruits = ['apple', 'banana', 'cherry']
    for fruit in fruits:
        print(fruit)
    ```

    - **Iterating over a range**:

    ```
    python
    Copy code
    for i in range(5):  # Iterates from 0 to 4
        print(i)
    ```

    - **Iterating over a string**:

    ```
    python
    Copy code
    for char in 'hello':
        print(char)
    ```

`while` Loop

- **Definition**: Repeats a block of code as long as a specified condition is `True`.
- **Syntax**:

```
python
Copy code
while condition:
    # Code to execute
```

- **Examples**:
    - **Basic `while` loop**:

      ```
      python
      Copy code
      count = 0
      while count < 5:
          print(count)
          count += 1
      ```

    - **Using `while` with `break`**:

      ```
      python
      Copy code
      count = 0
      while True:
          if count >= 5:
              break
          print(count)
          count += 1
      ```

Loop Control Statements

- **`break`**: Exits the loop prematurely.

  ```
  python
  Copy code
  for i in range(10):
      if i == 5:
          break
      print(i)
  ```

- **`continue`**: Skips the current iteration and continues with the next iteration of the loop.

  ```
  python
  Copy code
  for i in range(10):
      if i % 2 == 0:
          continue
      print(i)  # Prints only odd numbers
  ```

- **`else`**: An optional clause that executes after the loop completes normally (i.e., not terminated by `break`).

  ```
  python
  Copy code
  for i in range(5):
      print(i)
  else:
      print('Loop completed successfully')
  ```

These loops and control statements help in efficiently handling repetitive tasks and iterating through collections of data in Python.

○