Types of Functions in Python

Function Type	Description
Built-in Functions	These functions are pre-defined in Python and can be directly used without the need for any additional code. Examples include print(), len(), and range().
User-defined Functions	These functions are created by the programmer to perform specific tasks. They can be defined using the def keyword followed by a function name and a block of code. User-defined functions can accept arguments and return values.
Recursive Functions	Recursive functions are functions that call themselves within their own code. They are useful for solving problems that can be divided into smaller subproblems. Recursive functions must have a base case to prevent infinite recursion.

Types of Functions in Python

Function Type	Description
Lambda Functions	Lambda functions, also known as anonymous functions, are functions that are defined without a name. They are typically used for simple operations and can be created using the lambda keyword.
Higher-order Functions	Higher-order functions are functions that can accept other functions as arguments or return functions as values. They are useful for creating more flexible and reusable code.

Example of lamda function

- # Regular function
- def square(x):
- return x ** 2

- # Equivalent lambda function
- square_lambda = lambda x: x ** 2

- # Using lambda for a simple operation
- multiply = lambda x, y: x * y

- # Example usage
- print(square(5)) # Output: 25
- print(square_lambda(5)) # Output: 25
- print(multiply(3, 4)) # Output: 12

Example of Highy-order Function

```
# Higher-order function that accepts a function as an argument
def apply_operation(x, y, operation):
    return operation(x, y)

# Example of a function to be passed as an argument
def add(x, y):
    return x + y

# Using the higher-order function
result = apply_operation(3, 4, add)
print(result) # Output: 7

# Higher-order function that returns a function
def get_multiplier(factor):
    return lambda x: x * factor

# Using the returned function
double = get_multiplier(2)
triple = get_multiplier(3)

print(double(5)) # Output: 10
print(triple(5)) # Output: 15
```

Default Argument

- default arguments are used to assign a default value to a parameter if no argument is provided when the function is called.

Here is an example of a function with a default argument:

```
def greet(name='John'):
    print(f'Hello, {name}!')
```

If no argument is provided when calling the greet() function, it will use the default value of 'John'.

Example usage:

```
greet()# Output: Hello, John!
greet('Alice')   # Output: Hello, Alice!
```

Keyword Argument Functions

- keyword argument functions allow us to pass arguments to a function using the argument name, rather than relying on the order of the arguments. This provides more flexibility and makes the code more readable and maintainable.

Argument	Description
Positional Arguments	Arguments that are passed to a function based on their position in the function call.
Keyword Arguments	Arguments that are passed to a function using the argument name, allowing for more flexibility and readability.
Default Values	Assigning default values to keyword arguments, so they are optional to provide.
Example	def greet(name, age): print(f"Hello {name}, you are {age} years old.") greet(name="xyz", age=25) greet(25,sunny)

Variable Arguments Functions

- variable arguments functions allow you to pass a variable number of arguments to a function. This can be useful when you don't know how many arguments will be passed in advance or when you want to provide flexibility in the number of arguments.

Function	Description
args	Allows passing a variable number of non-keyword arguments to a function. The arguments are collected into a tuple.
*args	Similar to 'args', but allows passing a variable number of non-keyword arguments as individual arguments instead of a tuple.
**kwargs	Allows passing a variable number of keyword arguments to a function. The arguments are collected into a dictionary

Variable Arguments Functions

```
# Function using args to collect variable non-keyword arguments into a tuple
def print_args(*args):
    for arg in args:
        print(arg)

# Function using *args to pass a variable number of non-keyword arguments individually
def print_individual_args(first, *args):
    print(f"First argument: {first}")
    for arg in args:
        print(arg)

# Function using kwargs to collect variable keyword arguments into a dictionary
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Example usage
print("Using args:")
print_args(1, 2, "hello", [3, 4])

print("\nUsing *args:")
print_individual_args("apple", "banana", "orange")

print("\nUsing **kwargs:")
print_kwargs(name="Alice", age=25, city="Wonderland")
```

Local and Global Variables

- variables can be classified as either local or global. Understanding the difference between these two types of variables is crucial for writing efficient and maintainable code.

Variable Type	Scope	Access
Local Variable	Limited to the block or function where it is defined	Only accessible within the block or function where it is defined
Global Variable	Accessible throughout the entire program	Accessible from any part of the program

Global Keyword

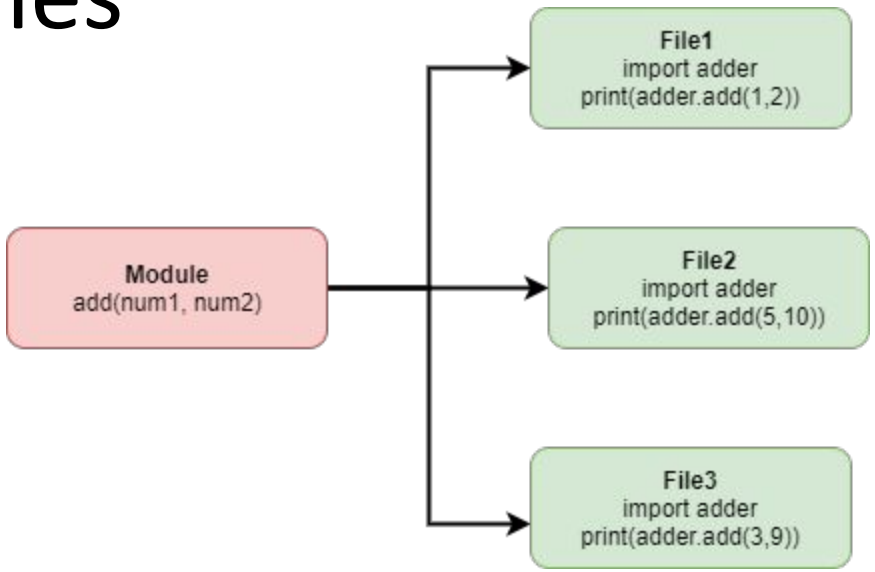
- The global keyword in Python programming is used to indicate that a variable is a global variable, meaning it can be accessed and modified from anywhere in the program.
- When a variable is declared as global, it can be used both inside and outside of functions. By default, variables declared within a function are considered local to that function and cannot be accessed outside of it.
- To declare a variable as global, the global keyword is used followed by the variable name. This tells Python that any references to that variable should be treated as a global variable.

Modules

a module is a file containing Python definitions and statements. The file name is the module name with the suffix .py. Modules are used to organize code into reusable blocks and to avoid naming conflicts. By importing modules, you can access the functions, classes, and variables

Module	Description
math	Provides mathematical functions and constants.
random	Generates random numbers and performs random selections.
datetime	Deals with dates, times, and time intervals.
os	Provides a way to use operating system dependent functionality.
sys	Provides access to some variables used or maintained by the interpreter and functions that interact with the interpreter.
json	Enables encoding and decoding of JSON data.

Modules



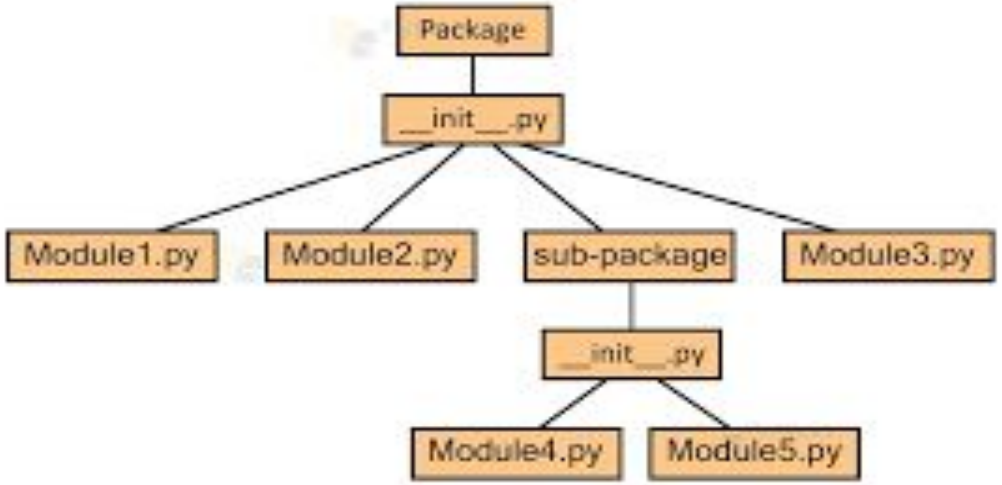
import module_name #you have to call function as module_name.functionname()

from module_name import Function/variable_name

import module_name as mn

from module_name import *

Structure of Packages



A package is a way of organizing related modules. A package is a directory containing Python modules and a special file called __init__.py. This file can be empty, but it indicates to Python that the directory should be treated as a package. Packages help in organizing code into hierarchical structures and facilitate modular programming.

Packages can contain subpackages, which are simply subdirectories with their own __init__.py files, and modules, which are individual Python files containing code.

Python packages are used to organize and distribute reusable code, making it easier to manage and maintain large projects. They provide a convenient way to structure code and to group related functionality together.

Packages

you can import packages using the import keyword. Here's the syntax:

Syntax: import packageName.moduleName

Syntax: import packageName.subPackageName.moduleName

Syntax: from packageName import moduleName

Syntax: from packageName.subPackageName import moduleName

Syntax: from packageName.moduleName import func_name

Syntax: from packageName import

Library

Library: A collection of code (modules or packages) that provide specific functionalities.

libraries	Description
NumPy	A powerful library for numerical computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a large collection of mathematical functions.
Pandas	A library for data manipulation and analysis. Pandas provides data structures like DataFrames and Series, which are efficient for handling and analyzing large datasets.
Matplotlib	A plotting library for creating visualizations in Python. Matplotlib provides a wide range of plots, including line plots, scatter plots, bar plots, histograms, and more.

Libraries

Libraries	Description
Scikit-learn	A machine learning library that provides a variety of supervised and unsupervised learning algorithms. Scikit-learn is widely used for tasks like classification, regression, clustering, and dimensionality reduction.
TensorFlow	An open-source machine learning framework developed by Google. TensorFlow provides a flexible ecosystem of tools, libraries, and community resources for building and deploying machine learning models.
PyTorch	Another popular machine learning framework, PyTorch offers dynamic computational graphs and automatic differentiation, making it easier to build and train deep learning models.