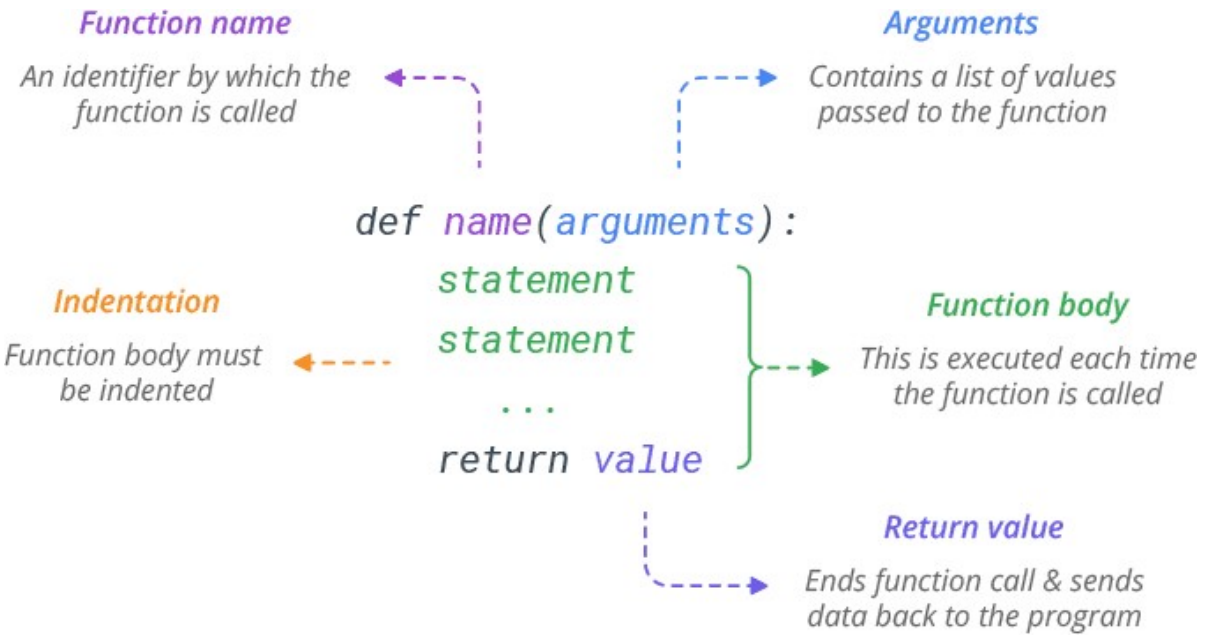
Assignment

Functions in Python

Python function in any programming language is a sequence of statements in a certain order, given a name. When called, those statements are executed. Functions provide a way to modularize and organize code, making it more readable, maintainable, and reusable.

The syntax to declare a function is:



Example:

```
def is_leap_year(year):
    """Check if a year is a leap year."""
    # Leap years are either divisible by 4 but not divisible by 100,or divisible by 400.
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)


is_leap_year(2023)
```

1) Defining a Function: To define a function, you use the **def** keyword, followed by the function name and a pair of parentheses. Any parameters the function takes are listed within the parentheses. The function body, containing the code to be executed, is indented below the def statement.

> **Rules for naming python function (identifier)**

1. It can begin with either of the following: A-Z, a-z, and underscore (_).

2. The rest of it can contain either of the following: A-Z, a-z, digits (0-9), and underscore (_).
3. A reserved keyword may not be chosen as an identifier.

It is good practice to name a Python function according to what it does.

2) Docstring: The first string after the function is called the Document string or Docstring in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

3) Calling a Function: To execute a function and perform the actions defined within it, you call the function by using its name followed by parentheses. If the function requires any arguments, you provide them within the parentheses.

4) Return Statement: Functions can return values using the return statement. The returned value can then be assigned to a variable or used in other parts of the code.

5) Parameters and Arguments: Parameters are variables listed in the function definition, while arguments are the actual values passed to the function when it is called.

6) Default Parameters: You can assign default values to parameters, making them optional when calling the function.

```
def power(base, exponent=2):
    return base ** exponent

power(3)      # Equivalent to power(3, 2)
```

6) **Scope and Lifetime of Variables in Python**

Scope: A variable's scope tells us where in the program it is visible. A variable may have local or global scope.

- **Local Scope-** Variables defined inside a function have local scope by default, meaning they are only accessible within that function.

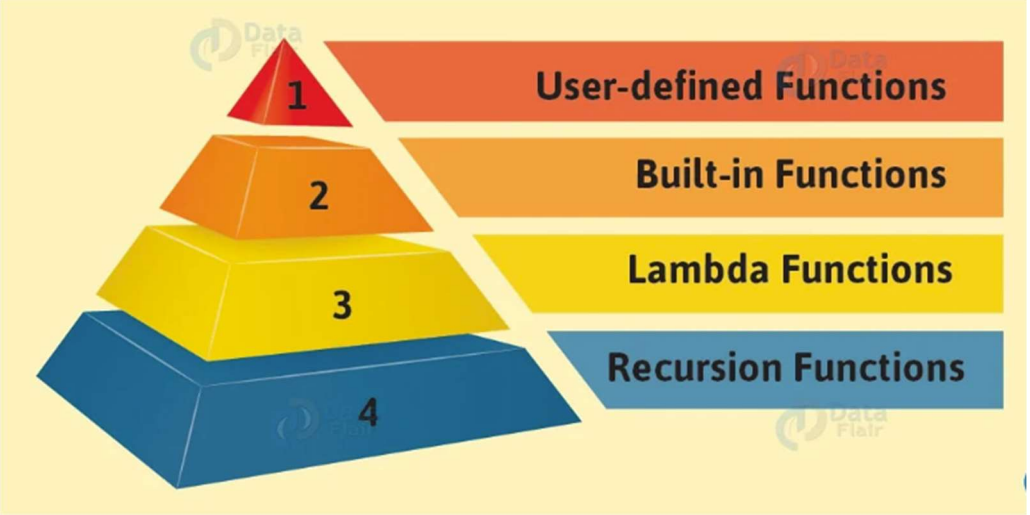
- **Global Scope-** When you declare a variable outside python function, or anything else, it has global scope. It means that it is visible everywhere within the program.

```
global_variable = 10

def example_function():
    local_variable = 5
    print(global_variable)  # Accessing global variable
    print(local_variable)   # Accessing local variable

example_function()
```

Lifetime: A variable's lifetime is the period of time for which it resides in the memory.

A variable that's declared inside python function is destroyed after the function stops executing. So, the next time the function is called, it does not remember the previous value of that variable.

Types of Functions in Python



➢ User-defined Functions

This Python Function help divide a program into modules. This makes the code easier to manage, debug, and scale. It implements code reuse. Every time you need to execute a sequence of statements, all you need to do is to call the function.

This Python Function allow us to change functionality easily, and different programmers can work on different functions. Users can create their own functions using the **def** keyword.

➢ Built-in Functions

Python built-in functions are pre-defined functions that come bundled with Python and can be used directly in a Python program without importing external libraries. They provide essential and commonly used functionality for various tasks, such as data type conversion, mathematical operations, string manipulation, file operations, etc.

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

1. **Type Conversion:**
    - **int()**: Converts a number or a string to an integer.
    - **float()**: Converts a number or a string to a floating-point number.
    - **str()**: Converts an object to a string.
2. **Math Functions:**
    - **abs()**: Returns the absolute value of a number.
    - **round()**: Rounds a floating-point number to the nearest integer.
    - **max()**: Returns the largest item in an iterable or the largest of two or more arguments.
    - **min()**: Returns the smallest item in an iterable or the smallest of two or more arguments.
3. **Sequence and Collection Operations:**
    - **len()**: Returns the length (number of items) of an object.
    - **sum()**: Returns the sum of all items in an iterable.
    - **sorted()**: Returns a new sorted list from the elements of any iterable.
4. **Input/Output:**
    - **print()**: Prints the specified value(s) to the standard output.
    - **input()**: Reads a line from the input and returns it as a string.
5. **Iterating and Generating:**
    - **range()**: Generates a sequence of numbers.
    - **enumerate()**: Returns pairs of index and value while iterating over an iterable.
    - **zip()**: Combines multiple iterables into a single iterable.
6. **String Manipulation:**
    - **len()**: Returns the length of a string.
    - **str()**: Converts an object to a string.
    - **format()**: Formats a string.
7. **Logical and Comparison:**
    - **bool()**: Converts a value to a Boolean.
    - **all()**: Returns **True** if all elements of an iterable are true.
    - **any()**: Returns **True** if any element of an iterable is true.
8. **File Operations:**
    - **open()**: Opens a file and returns a file object.

➢ Lambda Functions

These functions are similar to user-defined functions but without a name. They're commonly referred to as anonymous functions.

Lambda functions are efficient whenever you want to create a function that will only contain simple expressions – that is, expressions that are usually a single line of a statement. They're also useful when you want to use the function once.

The syntax of Lambda function is:

lambda *argument(s): expression*

1. **lambda** is a keyword in Python for defining the anonymous function.
2. **argument(s)** is a placeholder, that is a variable that will be used to hold the value you want to pass into the function expression. A lambda function can have multiple variables depending on what you want to achieve.
3. **expression** is the code you want to execute in the lambda function.

Example:

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

The anonymous function does not have a return keyword. This is because the anonymous function will automatically return the result of the expression in the function once it is executed.

Lambda functions are often used in situations where a small function is needed for a short period and creating a full function using def would be unnecessary. They are commonly employed with functions that accept other functions as arguments, such as map(), filter(), and sorted().

➢ Recursion Functions

In Python, it's also possible for a function to call itself! A function that calls itself is said to be recursive, and the technique of employing a recursive function is called recursion.

A recursive function in Python is a function that calls itself during its execution. Recursive functions are useful when a problem can be broken down into smaller, similar sub-problems.

Example: Factorial of a number

```python
def factorial(n):
    """Calculate the factorial of a number using recursion."""
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# Example usage:
result = factorial(5)
print(result)  # Output: 120
```

In this example:

- The base case (**n == 0 or n == 1**) specifies when the recursion should stop.
- The function calls itself with a smaller argument (**n - 1**) to solve a sub-problem.

It's important to have a base case to prevent infinite recursion. Each recursive call should move closer to the base case to ensure termination.
