Artificial Intelligence Diploma

Python Session 4

NeuroTech

Agenda

1. Function
2. Function Types
3. Scope of variable

Function

NeuroTech

Function

A function in Python is a self-contained block of code that performs a specific task or set of tasks. It allows you to group related code together, making your code more organized, modular, and reusable. Functions are a fundamental concept in programming and play a crucial role in structuring and organizing code.

Functions have the following key characteristics:

1. **Name**: Functions are given a name to identify them in your code.
2. **Parameters**: Functions can take zero or more input parameters (also called arguments) that are used as inputs for the function's logic.
3. **Return Value**: Functions can optionally return a value as the result of their execution.
4. **Code Block**: The body of the function contains the code that is executed when the function is called.

Function

Here's the basic syntax of a function in Python:

```
def function_name(parameters):
    # Code block
    return result
```

Let's break down the components:

- def: The keyword that defines a new function.
- function_name: The name of the function.
- parameters: The input values that the function takes (optional).
- return: The keyword used to specify the value that the function should return (optional).
- result: The value that is returned by the function (optional).

Function Types

NeuroTech

Function

In Python, functions can be categorized into several types based on their behavior, purpose, and usage. Here are some common types of functions:

1. **Built-in Functions:** These are functions that are built into the Python language and are always available for use without requiring an import statement. Examples include **print(), len(), max(), min(), abs(),** etc.

2. **User-Defined Functions**: These are functions that you create yourself using the **def** keyword. They allow you to group code into reusable blocks and give them meaningful names.

Built-in Functions

NeuroTech

Function

Built-in functions in Python are functions that are part of the core Python language and are available for use without needing to import any specific modules. These functions provide a wide range of capabilities and cover various operations, from basic tasks like mathematical calculations and string manipulation to more advanced operations like file handling and object manipulation. Here are some commonly used built-in functions in Python:

1. **Mathematical Functions**:
    - **abs(x):** Returns the absolute value of x.
    - **max(iterable):** Returns the maximum value in the iterable.
    - **min(iterable)**: Returns the minimum value in the iterable.
    - **round(number, ndigits):** Rounds a number to a specified number of decimal places.

Function

2. Type Conversion Functions:

1. **int(x):** Converts x to an integer.
2. **float(x):** Converts x to a floating-point number.
3. **str(x):** Converts x to a string.
4. **list(iterable):** Converts an iterable to a list.

3. Manipulation Functions:

1. **len(s):** Returns the length of the string or container.
2. **str.upper():** Converts the string to uppercase.
3. **str.lower():** Converts the string to lowercase.
4. **str.capitalize():** Capitalizes the first character of the string.
5. **str.split(separator):** Splits the string into a list based on the separator.

User-Defined Functions

NeuroTech

Function

Functions can have various types of parameters that allow you to pass different kinds of values when calling the function. The different types of parameters are:

- **Positional Parameters**: These are the most common type of parameters. They are matched with arguments in the order they are passed. The position of the argument determines which parameter it corresponds to.

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 25)  # Positional arguments
```

Function

- **Keyword (Named) Parameters**: You can specify arguments using parameter names, which allows you to pass them in any order. This can make function calls more readable and less error-prone.

```
greet(age=30, name="Bob")  # Keyword arguments
```

- **Default Parameters**: You can provide default values for parameters. If an argument is not provided, the default value will be used.

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

greet("Alice")  # Uses default greeting "Hello"
greet("Bob", "Hi")  # Overrides default greeting
```

Function

- **Variable-Length (Arbitrary) Parameters**: Functions can accept a variable number of arguments using *args and **kwargs. The *args syntax allows you to pass a variable number of positional arguments, while **kwargs allows you to pass a variable number of keyword arguments.

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args("apple", "banana", "cherry")

def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_kwargs(fruit="apple", color="red")
```

Function

- **Combining Parameter Types**: You can combine different parameter types in a single function.

```
def example(a, b=10, *args, **kwargs):
    print(a, b)
    print(args)
    print(kwargs)

example(5)  # a=5, b=10 (default), no *args, no **kwargs
example(2, 3, 4, 5, foo="bar", hello="world")  # a=2, b=3, *args=(4, 5),
```

Anonymous Function

Function

- In Python, an anonymous function is a function that is defined without a name. It's also known as a lambda function. Lambda functions are often used for short, simple operations where a full function definition using the **def** keyword would be unnecessary or cumbersome.

- Lambda functions are defined using the **lambda** keyword, followed by a list of parameters, a colon, and an expression. The expression is evaluated and returned when the lambda function is called.

- The basic syntax of a lambda function is as follows:

  lambda arguments: expression

- Here's a simple example of a lambda function that calculates the square of a number:

```
square = lambda x: x * x
result = square(5)  # Result is 25
```

Using Lambda Functions:

Lambda functions can be used wherever a function object is required. For example, they can be used with functions like **map(), filter(),** and **sorted().**

- Using **map()** with a lambda function:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
# squared_numbers is [1, 4, 9, 16, 25]
```

- Using **filter()** with a lambda function:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
# even_numbers is [2, 4, 6, 8, 10]
```

Using Lambda Functions:

- Using **sorted()** with a lambda function:

```
data = [(3, 'apple'), (1, 'banana'), (2, 'cherry')]
sorted_data = sorted(data, key=lambda x: x[1])
# sorted_data is [(3, 'apple'), (1, 'banana'), (2, 'cherry')]
```

While lambda functions are concise and useful in certain scenarios, they should not be overused. For more complex operations or functions that will be used multiple times, it's often better to define a regular named function using the **def** keyword for better readability and maintainability.

Recursive Function

NeuroTech

Recursive Function

A recursive function is a function that calls itself in order to solve a problem by breaking it down into smaller instances of the same problem. In other words, a recursive function solves a problem by solving smaller instances of the same problem until it reaches a base case where a solution can be directly determined. Recursive functions are commonly used in programming for tasks that can be divided into simpler, similar sub-problems.

A recursive function typically consists of two parts:

1. **Base Case**: This is the condition under which the function stops calling itself and returns a known result. It's the simplest form of the problem that can be solved directly without further recursion.

2. **Recursive Case**: This is where the function calls itself with smaller or simpler arguments. Each recursive call reduces the problem size until it eventually reaches the base case.

Recursive Function

Here's an example of a simple recursive function that calculates the factorial of a non-negative integer:

```
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n - 1)  # Recursive case
```

In this example, the base case is **n == 0**, where the factorial is known to be 1. The recursive case calculates the factorial of **n** by multiplying it with the factorial of **n - 1.** This process continues until the base case is reached, at which point the recursion stops and the final result is computed.

Recursive Function

Let's see how the function works for calculating the factorial of 5:

```
factorial(5) = 5 * factorial(4)
             = 5 * (4 * factorial(3))
             = 5 * (4 * (3 * factorial(2)))
             = 5 * (4 * (3 * (2 * factorial(1))))
             = 5 * (4 * (3 * (2 * (1 * factorial(0)))))
             = 5 * (4 * (3 * (2 * (1 * 1))))
             = 120
```

Recursive functions can be elegant and concise, but they also require careful design to ensure they reach the base case and don't lead to infinite recursion. When using recursive functions, it's important to consider efficiency, as some problems can be solved more efficiently using iterative approaches or dynamic programming techniques.

High Order Function

NeuroTech

High Order Function

- In programming, a high-level function (also known as a higher-order function) is a function that takes one or more functions as arguments, returns a function as its result, or both. In other words, a high-level function treats functions as first-class citizens, just like any other data type such as integers, strings, or lists.

- The concept of high-level functions is closely related to functional programming, which is a programming paradigm that emphasizes the use of functions as the primary building blocks of software. High-level functions enable you to write more modular and reusable code by abstracting common patterns into function parameters or return values.

High Order Function

Here are some common examples of high-level functions and their usage:

1. Map Function:

The map() function applies a given function to each item in an iterable (such as a list) and returns an iterator of the results.

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)  # Returns an iterator
```

2. Filter Function:

The filter() function creates an iterator from elements of an iterable for which a given function returns True.

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(is_even, numbers)  # Returns an iterator
```

High Order Function

3. Reduce Function:

The functools.reduce() function applies a binary function to the elements of an iterable to reduce them to a single value.

```
from functools import reduce

def multiply(x, y):
    return x * y

numbers = [1, 2, 3, 4, 5]
product = reduce(multiply, numbers)  # Returns a single value
```

4. Function Composition:

You can compose functions together using other functions. This allows you to create more complex functions by chaining together simpler ones.

```
def add(x, y):
    return x + y

def square(x):
    return x * x

result = square(add(3, 5))
```

High Order Function

5. Lambda Functions:

As mentioned earlier, lambda functions (anonymous functions) are often used as arguments to high-level functions.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x * x, numbers)
```

High-level functions promote code reusability, modularity, and maintainability. They allow you to express complex operations concisely by abstracting away lower-level details. By understanding and using high-level functions, you can write more elegant and efficient code that leverages the power of functional programming concepts.

Scope of Variable

Scope of Variable

- The **scope** of a variable refers to the region of your code where the variable can be accessed or modified. Python follows a set of rules to determine the scope of variables. Understanding variable scope is crucial for writing clean and bug-free code.

- Python has two main types of variable scope:

1. **Local Scope (Function Scope)**: Variables defined inside a function are considered to have a local scope. They can only be accessed and used within that function.

```
def my_function():
    x = 10  # Local variable
    print(x)

my_function()
print(x)  # Raises a NameError, x is not defined outside the function
```

Scope of Variable

- **Global Scope**: Variables defined outside of any function or code block have a global scope. They can be accessed and modified from any part of the code.

```
y = 5  # Global variable

def my_function():
    print(y)  # Accessing a global variable

my_function()
print(y)
```

Scope of Variable

Local and Global Variables:

If a variable is defined with the same name in both the local and global scopes, the local variable takes precedence within the local scope. However, the global variable remains unchanged.

```
z = 15  # Global variable

def another_function():
    z = 20  # Local variable with the same name
    print(z)  # Prints the local variable

another_function()
print(z)  # Prints the global variable
```

Scope of Variable

Using Global Variables in a Function:

If you need to modify a global variable within a function, you need to explicitly declare it as global using the **global** keyword.

```
count = 0  # Global variable

def increment_counter():
    global count
    count += 1

increment_counter()
print(count)  # Output: 1
```

Scope of Variable

Nested Functions and Variable Scope:

If a function is defined within another function, it can access variables from both its local scope and the scope of the containing function.

```
def outer_function():
    a = 10

    def inner_function():
        b = 20
        print(a + b)  # Accessing variables from both scopes

    inner_function()

outer_function()
```

Any Question?

NeuroTech

Thanks

NeuroTech