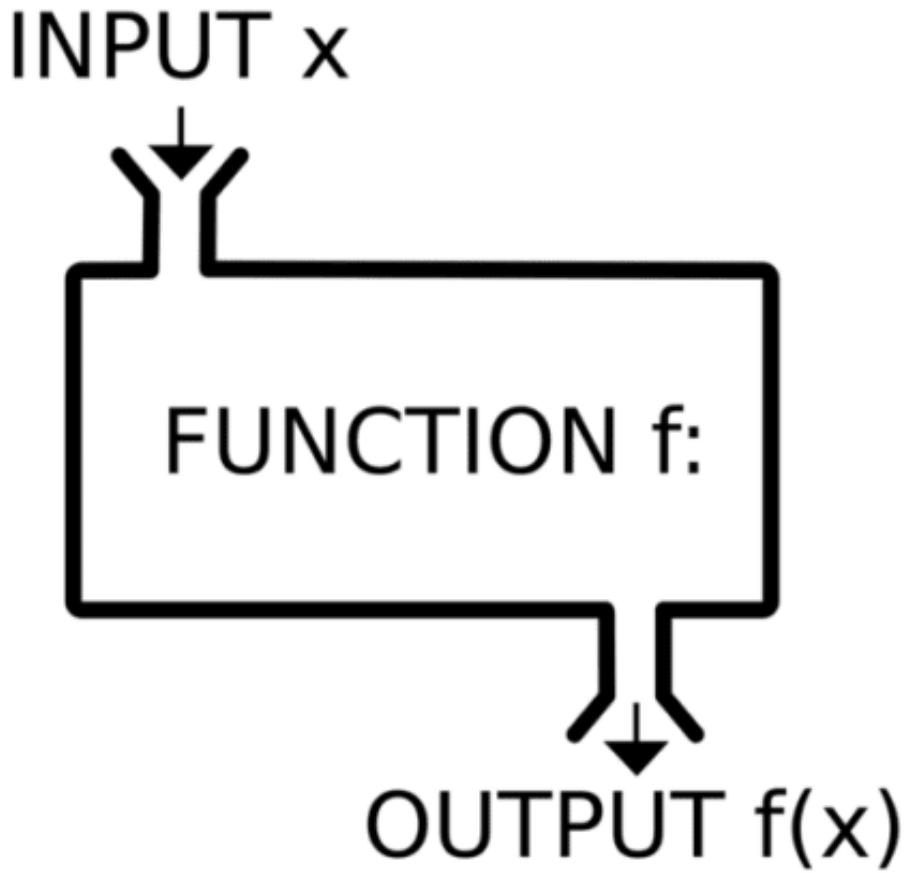

Week 4: Functions

Functions in real life

Baking recipes:

Think of a function as a baking recipe. The recipe (function) takes specific ingredients (arguments) and provides a step-by-step set of instructions (code) to bake a cake (output).

ATM withdrawal:

Imagine a function as an ATM machine. You input the amount you want to withdraw (argument), and it performs the necessary steps to dispense the money (output).

Functions

- Functions are a way of organizing code into reusable blocks.
- Functions can be defined once and called multiple times with different arguments from different parts of a program reducing duplication (**Reusability**).
- Functions help break down a program into smaller, more manageable parts and make it more organized (**Modularization**).
- They are used to perform specific tasks or calculations.
- They help you make your code easier to understand and maintain.
- Functions can take inputs (arguments) and return outputs (return values).

Defining Functions

- To define a function, use the **def** keyword, followed by the **function name** and any **parameters**.
- The syntax for defining a function in Python is as follows:

```
def function_name(parameters):
    # Function body
    # Code that performs a task
    return result
```

Explanation of the syntax

- def: This keyword is used to define a function in Python.

- function_name: This is the name you choose for your function. Make sure it follows Python naming conventions (letters, numbers, and underscores, but cannot start with a number).

- parameters (optional): They act as placeholders for data you want to use inside the function. Parameters are like variables that hold values that you can pass to the function.

- : (colon): The colon indicates the start of the function's code block, which is indented.

Explanation of the syntax:

- Code Block : This is where you write the instructions for what the function should do. You can use the parameters to perform calculations or any other operations.

- return (optional): If you want the function to give back a result, you can use the return statement. It's not required, and you can have functions that don't return anything.

Example

Greet function:

- This function greets by saying “Hello World”.
- This function has no parameter and no return value.

```
def greet():
    print("Hello World")
```

Baking cake function

Here is a function defined to bake a cake.

- Instructions needs to be added.

```
def bake_cake(flour, eggs, milk):
    # Instructions to make pancakes using the ingredients
    ...
    return cake
```

When do we need parameters?

- You can have zero, one, or multiple parameters, depending on the function's requirements.

- Whether you define parameters depends on whether your function needs input values to perform its task.

- If your function doesn’t need any input values, you can define it without parameters. If it requires input values, then you define the necessary parameters in the parentheses.

Example

Greet function:

- This function takes the name of a person and greets them by saying their name.
- This function has one parameter and no return value.

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

return statement

- In Python, the return statement is used in a function to specify the value that the function should produce as its result.

- You need to use the return statement in a function when you want the function to return a value to the caller. The presence of a return statement is what allows you to obtain a result from a function.

- Remember that if a function does not contain a return statement, it will return None by default.

When do we need the return statement?

Returning a Value: If your function performs a computation or operation and needs to provide the result to the caller, you use return to send that value back.

Example: This function takes two numbers and creates a variable called results and assigns the sum of those two numbers to it. At the end, it returns the result value.

- This function has two parameters and one return value.

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

When do we need the return statement?

Exiting Early: You might use return to exit a function prematurely if a specific condition is met. This can be useful for error handling or to optimize code by avoiding unnecessary processing.

Example: the divide function returns an error message if the denominator (y) is zero.

```
def divide(x, y):
    if y == 0:
        return "Cannot divide by zero"
    result = x / y
    return result
```

When do we need the return statement?

Returning Multiple Values: You can return multiple values from a function as a tuple, list, or another data structure.

Example: Here, the get_coordinates function returns two values as a tuple.

```
def get_coordinates():
    x = 3
    y = 7
    return x, y
```

When do we need the return statement?

Conditional Returns: You can use return in conditional statements to return different values based on specific conditions.

Example: This function returns a letter grade based on the numeric score provided.

```
def get_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    else:
        return "F"
```

Parameters VS Arguments

- Parameters are the named variables in a function definition that receive values when the function is called.
- Arguments are the values that are passed to a function when it is called.
- Functions can have more than one parameter and argument.
- It's important to ensure that the number and types of arguments you pass match the function's parameter list to avoid errors.
- If you provide too few or too many arguments or if the argument types don't match the parameter types, you'll encounter a **TypeError** or incorrect behavior.

Calling Functions

- To call a function in Python, use the function name followed by parentheses and any arguments:

function_name(argument1, argument2, ...)

- The value returned by a function can be stored in a variable or used directly.

- Here is how to capture and use the return value when calling the function.

```
sum_result = add_numbers(5, 3)
print(sum_result)  # This will print: 8

print(add_numbers(5, 3))  # This will also print: 8
```

Positional and Keyword arguments

- In Python, when you call a function, you can pass arguments to that function in two main ways: positional arguments and keyword arguments.

Positional Arguments:

- Positional arguments are the most common type of arguments in Python.
- They are passed to a function based on their position in the function's parameter list.
- The order and number of positional arguments must match the order and number of parameters in the function's definition.

Example

In this example, "Alice" is passed as the first positional argument, and 25 is passed as the second positional argument.

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

greet("Alice", 25)
```

Positional and Keyword arguments

Keyword Arguments:

- Keyword arguments are passed to a function using parameter names as keywords, and their order does not matter.
- They allow you to specify which value corresponds to which parameter explicitly, making your code more readable.
- You can mix positional and keyword arguments in a function call, but positional arguments must come before keyword arguments.

Example

In this example, age=25 and name="Bob" are passed as keyword arguments.

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

greet(age=25, name="Bob")
```

Default Values

- Parameters in Python functions can have default values. When a parameter has a default value, it becomes optional when calling the function.

- You can provide a value for it if you want to override the default, or you can omit it, and the default value will be used.

Example

In this example, the age parameter has a default value of 30, so if you don't provide an age when calling the function, it will use 30 as the default.

```
def greet(name, age=30):
    print(f"Hello, {name}! You are {age} years old.")

greet("Charlie")  # "age" uses the default value of 30
greet("David", 40)  # "age" is overridden with 40
```

Variable number of arguments

- Functions in Python can accept a varying number of arguments.
- Traditional way: Define a function with a fixed number of parameters.
- Sometimes, it's useful to accept any number of arguments without explicitly defining them.
- In Python, ***args*** and ****kwargs*** are special syntax used in function definitions to allow a function to accept a variable number of positional arguments and keyword arguments, respectively.
- They are often used when you don't know in advance how many arguments will be passed to a function.

Variable Positional Arguments

- ***args** is used to pass a variable number of positional arguments to a function.

- Inside the function, args will be treated as a tuple. You can access the collected arguments using the ***args tuple***.

- The asterisk (*) before the parameter name args allows the function to accept any number of positional arguments.

- The name ***args*** is a convention, but you can choose any valid variable name preceded by *.

Example

Function with arbitrary arguments:

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

my_function('apple', 'banana', 'orange')
# Output:
# apple
# banana
# orange
```

Example

```
def example_function(arg1, *args):
    print("First argument:", arg1)
    print("Additional arguments:", args)

example_function("A", "B", "C", "D")
```

This will be the output:

```
First argument: A
Additional arguments: ('B', 'C', 'D')
```

Variable Keyword Arguments

- ****kwargs** is used to pass a variable number of keyword arguments to a function.

- Inside the function, kwargs will be treated as a dictionary. You can access the collected keyword arguments using the **kwargs dictionary**.

- The double asterisks (**) before the parameter name kwargs allows the function to accept any number of keyword arguments.

- The name ***kwargs*** is a convention, but like *args, you can use any valid variable name preceded by **.

Example

Function with arbitrary arguments:

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

my_function(fruit='apple', color='red', taste='sweet')
# Output:
# fruit: apple
# color: red
# taste: sweet
```

Example

```
def example_function(arg1, **kwargs):
    print("First argument:", arg1)
    print("Additional keyword arguments:", kwargs)

example_function("A", key1="B", key2="C", key3="D")
```

This is the output:

```
First argument: A
Additional keyword arguments: {'key1': 'B', 'key2': 'C', 'key3': 'D'}
```

Combining **args* and ***kwargs*:

- You can use *args and **kwargs together in a function definition.
- When defining the function, ****args*** must come before *****kwargs***.

```
def my_function(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(f"{key}: {value}")

my_function('apple', 'banana', 'orange', fruit='apple', color='red', taste='sweet')
# Output:
# apple
# banana
# orange
# fruit: apple
# color: red
# taste: sweet
```

Example

```
def example_function(arg1, *args, **kwargs):
    print("First argument:", arg1)
    print("Additional arguments:", args)
    print("Additional keyword arguments:", kwargs)

example_function("A", "B", "C", key1="D", key2="E")
```

This will be the output:

```
First argument: A
Additional arguments: ('B', 'C')
Additional keyword arguments: {'key1': 'D', 'key2': 'E'}
```

Scope

- The concept of scope in Python refers to the area of a program where a particular variable or name is accessible and can be used.

- There are two main types of scope in Python: **global scope** and **local scope**, which is closely related to functions.
  - **Local variables** are defined inside a function and can only be accessed inside that function.
  - **Global variables** are defined outside of any function and can be accessed from anywhere in the program.

Local Scope

- Variables defined inside a function have local scope.
- Local variables are only accessible within that function and cannot be accessed or modified outside of the function in which they are defined.

```
def local_scope_example():
    x = 10  # This variable x is in the local scope of the function
    print(x)

local_scope_example()  # Outputs: 10

# Accessing x here would result in a NameError because it's not in the global scope
print(x)  # This would result in an error
```

Global Scope

- Variables defined outside of any function and at the top level of a script or module are in the global scope.
- Global variables can be accessed from anywhere in the module.

```
y = 5  # This is in a global scope

def global_scope_example():
    print(y)  # Accessing the global variable y

global_scope_example()  # Calling the function, outputs: 5

print(y)  # Accessing the global variable outside the function, outputs: 5
```

Scope Hierarchy

- Python follows a hierarchy when looking for a variable's value. It first checks the local scope, then the enclosing (non-local) scopes, and finally the global scope.
- If a variable is not found in the local scope, Python will look for it in any enclosing scopes (such as outer functions). If it's still not found, it will search in the global scope.

```
x = 10  # This variable is in the global scope

def enclosing_scope_example():
    x = 20  # This variable x is in the local scope of the function
    print(x)  # Prints the local x

enclosing_scope_example()  # Outputs: 20 (local x is used)

print(x)  # Outputs: 10 (global x is used)
```

Nested Functions

- Functions can be nested within each other, creating multiple levels of scope.

```
x = 5  # Variable in the global scope

def outer_function():
    y = 10  # Variable in the outer function's scope

    def inner_function():
        z = 15  # Variable in the inner function's scope
        print(x + y + z)  # Accessing the variable from the global and outer function's scope

    inner_function()

outer_function()  # Outputs: 15
```

global Keyword

- If you want to modify a global variable from within a function, you can use the global keyword to indicate that you're working with the global variable.

```
z = 30  # This variable z is in the global scope

def modify_global_variable():
    global z  # Declare that we want to modify the global variable
    z += 5  # Modifying the global variable z

print(z)  # Outputs: 30
modify_global_variable()
print(z)  # Outputs: 35
```

Importance of Understanding Scope

- Understanding scope is essential in Python programming because it determines where variables are valid and helps prevent naming conflicts.

- Local scope allows you to encapsulate data within functions, while global scope provides a broader accessibility.

- When using variables in functions, it's crucial to be aware of scope to avoid unexpected behavior or conflicts in your code.

Writing better functions

- **Documentation**: Add comments or docstrings to describe what each function does, its parameters, and its return values.

- **Debugging**: Debug functions by isolating them and testing them separately from the rest of the code.

- **Parameter choices**: Choosing meaningful parameter names to improve code readability is very important.

Lambda Functions

- Lambda functions, also known as anonymous functions or lambda expressions are a concise way to create small, unnamed functions in Python that can be defined in one line.
- Lambda functions are often used for simple, one-time operations are needed only once.
- Lambda functions can have any number of parameters but can only have one expression.
- The syntax of a lambda function is:

```
lambda parameters: expression
```

Syntax of Lambda Functions

- **lambda**: This keyword is used to define a lambda function.

- **parameters**: These are the input parameters that the lambda function takes. You can specify any number of parameters, separated by commas.

- **expression**: This is a single expression or statement that the lambda function evaluates and returns. The result of this expression becomes the return value of the lambda function.

Note: 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.

When should you use a lambda function?

- ***Simple and Small Functions***: Lambda functions are best suited for small, simple functions that can be defined in a single line. If a function requires multiple lines of code or more complex logic, it's better to use a regular function.
  - For example, if you want to create a function with a for-loop, you should use a user-defined function.

- ***Anonymous Function***: They are often used for short-lived, one-time operations where you don't need to give the function a name.

- ***Functional Programming***: They are commonly used in functional programming constructs like map, filter, and reduce because they allow you to define inline functions for specific operations.

Example

Here is the square function written in regular way and the lambda function.

```
# Regular Function
def square(x):
    return x * x


# Equivalent Lambda Function
square_lambda = lambda x: x * x
```

Summary

- In this lesson, we learned about functions in Python.

- We learned how to define and call functions, use parameters and arguments, return values and statements, and understand scope.

- We also learned about lambda functions and how they can be used in Python.

Exercises

Here are some exercises for Week 4:

- Write a function that takes a positive integer number and returns its factorial.
- Write a function that takes temperature in Fahrenheit and returns it in Celsius.
- First, write a function that takes a letter and checks whether it is a vowel or not. Then use that function in another function to count the number of vowels in a string.
- Write a program that calculates the tax percentage based on gross pay, and then use this function in another function to calculate net pay. Remember that in the net pay, the employee receives 50 Euros as transportation expenses. Here is the percentage scale:

0-240 → 0%, 241-480 → 15% and above 481 → 28%