# COMP11124 Object Oriented Programming

## **Python Functions, Scope and Errors**

### Learning Outcomes

The lab exercises in this class are the last part you need to start building your simple procedural Python programs. You will learn how to create and use functions as well as what variable scope is. Additionally, you will also investigate common errors in Python and how they are fixed.

At the end, you will build a simple To-Do application, which combines the things you have learned from the last three weeks.

It is essential that you finish the labs until next week as then, we will properly start looking at Object Oriented Programming, the main core topic of this module!

### Topics Covered

Functions, Scope, Assertions, Common Python Errors

**Getting Started Task:** To retain all the code that you write and be able to work through the exercises in this lab, ***create a new Python file called: lab\_week\_3.py*** like how we did it last week. You can include the code of this week here and should continuously execute it to see the output. If you want to see the value of a variable, please *print* it using the print() function, as I have omitted that in the sample code.

# Functions and Scope

## Exercise 1: Functions in Python

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Functions are very useful when you need to perform the same task multiple times throughout your code. For example, if we think of our temperature conversion example from the previous few weeks, we could turn this into a function. This would allow us to use the temperature conversion multiple times in our code without having to write the same code repeatedly. This is especially important if the size of our code is growing, and we want to keep it as clean and concise as possible.

### Creating Functions

To create a function in Python, you use the def keyword, followed by the name of the function.

For example, let us create a function called greet\_user that prints a greeting to the user. We can do this by using the following code:

def greet\_user():

    print("Hello!")

Note that the function definition ends with a colon and the function name is followed by parentheses.

The code that the function executes is indented below the function definition. This is similar to how we indent code in if statements and loops.

If you want to call a function, you simply write the name of the function followed by parentheses. For example, to call the greet\_user function, you would use the following code:

greet\_user()

### Function Parameters

Additionally, functions can take parameters, which are variables that are passed into the function when it is called. Before you can use a parameter in a function, you need to define it in the function definition. The parameters you use within a function definition need to be defined between the parentheses and ONLY exist within the function.

If we change our example to take a parameter called name, we can use the parameter in the function definition. We can do this by using the following code:

def greet\_user(name):

    print(f"Hello {name}!")

Now, when we want to call the function, we need to pass a value (an argument) for the name parameter. We can do this as follows:

greet\_user("John")

When working and reading about functions, you will often see the terms "**parameter**" and "**argument**". Although they both mean the same: data passed and used by the function, there is a difference between the two.

* A parameter is the variable listed inside the parentheses in the function definition and that is used within the function's body.
* An argument is the value that we pass to the function when we call it.

So, for our example above, name is a parameter and "John" is an argument.

If we now want to change it to have more than one parameter, we can do this by adding a comma between the parameters in the function definition. Let us use a user's first and last name as parameters.

def greet\_user(first\_name, last\_name):

    print(f"Hello {first\_name} {last\_name}!")

When we call this new function, it is crucial that we pass the parameters in the correct order and that we pass the correct number of parameters.

greet\_user("John", "Smith")

### Keyword Arguments

Functions can not only have positional arguments but also keyword arguments. Keyword arguments are arguments that are passed to a function with a keyword and an equals sign. When we use keyword arguments, the order of the arguments does not matter. For example, we could call our greet\_user function like this:

greet\_user(last\_name="Smith", first\_name="John")

Note that we have switched the order, but the function still works.

Keyword arguments are useful when you have a function with many parameters, and you want to ensure that it is clear when calling the function which argument is which.

### Default Values

Like the keyword arguments syntax, you can also set default values for parameters. This is very helpful when you have a function with many parameters and you want to set a default value for some of them. You need to define this default value in the function definition. For example, if we wanted to set a default value for an additional parameter called age, we could do this as follows:

def greet\_user(first\_name, last\_name, university="UWS"):

    print(f"Hello {first\_name} {last\_name} from {university}!")

If we now call the function without passing a value for the university parameter, the function will use the default value that we have set in the function definition (UWS). For example, if we call the function like this, the output will be as follows:

greet\_user("John", "Smith")

Output: *Hello John Smith from UWS!*

But if we include a value for the university parameter, the function will use the value that we have passed instead of the default value.

greet\_user("John", "Smith", "UWS London")

# or

greet\_user("John", "Smith", university="UWS London")

Output: *Hello John Smith from UWS London!*

Output: *Hello John Smith from UWS London!*

Both, using the positional arguments syntax and the keyword arguments syntax, are valid ways of calling a function. However, it is important to be consistent and use the same syntax throughout your code.

**Task:** Create a function called greet\_friends. The function should take a list of names as a parameter and print a greeting for each name in the list. The greeting should be "Hello " followed by the name. For example, if the list contains the names "John", "Jane" and "Jack", the output should be as follows:

*Hello John!*

*Hello Jane!*

*Hello Jack!*

Note: You can use the for loop syntax to iterate over the list of names and use the following list as a starting point:

friend\_list = ["John", "Jane", "Jack"]

# function code goes here

greet\_friends(friend\_list)

### Return

Another important aspect of functions is that they can return values. This means that the function not only executes some code but returns a value that can be used in the rest of the code. We can for example store the returned value in a variable and then use it later in our code.

To return a value from a function, we use the return keyword followed by the value we want to return. For example, let us create a function that returns the sum of two numbers. We can do this by using the following code:

def add\_numbers(num1, num2):

    return num1 + num2

We can also simply return a variable that we have defined in the function.

def add\_numbers(num1, num2):

    result = num1 + num2

    return result

This is the same as the previous example, but we have assigned the result of the addition to a variable called result before returning it.

We can now use the function and store the returned value in a variable. For example, if we want to add the numbers 5 and 10, we can do this as follows:

result = add\_numbers(5, 10)

print(result)

You may note that we have used the same variable name for the variable that stores the returned value as the variable that we have used in the function called result. This is not a problem, because the variable that we have defined in the function only exists within the function. The variable that we have defined outside of the function is a different variable and can be used without any problems. This is called variable scope and is an important concept in programming. We will look at this in more detail in the next exercise.

You can also return more than one value from a function. To do this, you simply need to separate the values you want to return with a comma. For example, if we want to return the sum and the product of two numbers, we can do this as follows:

def add\_and\_multiply\_numbers(num1, num2):

    return num1 + num2, num1 \* num2

or

def add\_and\_multiply\_numbers(num1, num2):

    sum = num1 + num2

    product = num1 \* num2

    return sum, product

(which is the same as the previous example, but may be easier to read if you are new to programming)

If you return more than one value from a function, you need to store the returned values in multiple variables. Simply separate the variables that will store the returned values with a comma.

sum, product = add\_and\_multiply\_numbers(5, 10)

print(sum)

print(product)

Now that you understand how to create functions, let us do some exercises to practice what we have learned.

If you are unsure about how to solve the exercises, you can always look back at the previous examples or read through the following online guide (<https://www.w3schools.com/python/python_functions.asp> ) that covers the same topics as this exercise but in somewhat more detail.

**Task Tax Calculation:**

1. Define a function called calculate\_tax that takes two arguments: income and tax\_rate.
2. Inside the function, calculate the tax amount by multiplying income by tax\_rate.
3. Return the tax amount as the result.
4. Call the calculate\_tax function with an income of 50,000£ and a tax rate of 0.2 and print the calculated tax.
5. Try using different incomes and tax rates in this function.

**Task Compound Interest Calculator Function:**

Your goal for this task is to write a function called compound\_interest()that calculates the total amount of money earned by the investment every year. Here are the specifications:

* The function should have three parameters: principal, duration and interest\_rate.
* The function should print the total amount of money earned by the investment every year.
* If the interest rate is smaller than 0 or larger than 1 the function should print out a message that says "Please enter a decimal number between 0 and 1" and return None.
* If the duration is less than 0 the function should print out a message that says "Please enter a positive number of years" and return None.
* The function should use a for loop to calculate the amount of money earned by the investment **every year** (Hint: use the range function and keep in mind that this starts at 0). The format of the output should be: "The total amount of money earned by the investment in year **Y** is **X** £" where **X** is the total amount of money earned by the investment and **Y** the year.
* The function should return the final investment value as an *integer* using the int() function

The formula to calculate the total amount of money earned by the investment for each year is given by:  ***total\_for\_the\_year = principal \* (1 + interest\_rate) \*\* year***

You can test whether this function works by calling it like this: compound\_interest(1000, 5, 0.03) and it should return £1159 for the result in year 5.

## Exercise 2: Variable Scope

Variable scope is an important concept in programming. It refers to the area of the code where a variable can be accessed.

A very simple example of variable scope is the following:

def new\_function():

    my\_new\_variable = 5

new\_function() # call the function. No problems here.

print(my\_new\_variable) # this will cause an error

## 

We define a new variable called my\_new\_variable inside the function new\_function. This variable only exists within the function and can only be accessed within the function. If we try to access the variable outside of the function, we will get an error. This is because the variable only exists within the function and not outside of it. This is called local scope.

Local scope only exists within functions but not within if statements or loops. For example, if we define a variable inside an if statement, we can still access it outside of the if statement. This is called global scope.

Why is this important? Well, you need to be aware of where you define your variables and need to remember that variables are defined within functions ***only!*** exist within the function.

On the other hand, variables defined outside of functions such as in the main body of the code can be accessed within functions. These variables exist within the global scope.

One thing you need to be also aware of is variables that have the same name but exist in different scopes.

my\_new\_variable = 0

def new\_function():

    my\_new\_variable = 5

new\_function()

print(my\_new\_variable)

It may be tempting to think that the variable my\_new\_variable in the function is the same as the variable my\_new\_variable outside of the function. If you run the code, you will see that this is not the case. The variable outside of the function is a global scope variable and the variable inside the function is a local scope variable. They are two different variables that just happen to have the same name.

Summarising this, be very careful when defining variables and make sure that you are aware of the scope of the variable. If you are unsure, you can always use a different variable name to avoid confusion.

# Assertions and Errors

Exercise 6: Assertions

Python assertions are used to validate that certain conditions hold during program execution. Assertions help you catch and handle errors early in your code. Generally, assertions are statements that check whether a given condition is true. If the condition is false, an AssertionError is raised, indicating a problem in the code. Assertions are useful for debugging and ensuring that your code works as expected and can stop problems from occurring before they become severe.

If you built the function in exercise 5 correctly, the following code should run without problems:

assert compound\_interest(1000, 5, 0.03) == 1159

More information on assertions can be found here: <https://www.tutorialspoint.com/python/assertions_in_python.htm>

Exercise 7: Identifying and Fixing Common Errors

Errors in Python tell you when something is wrong. There are two main kinds of errors which you will encounter when programming: Logical Errors, and Semantic Errors.

Logical errors occur when the code is semantically correct, but you have built your program incorrectly. For example, you have a function that performs addition, but instead of using the + operator to add numbers together, you use the – one.

Semantic errors are the kind of error that will lead to error messages either before, or during program execution. They are very common, and therefore you must know how to understand and fix them.  In this exercise, you will practice identifying and fixing common types of errors in Python code. These errors include Syntax, Name, Value, Index, and Indentation Errors.

Below are examples of those errors in Python code. (**Note:** do not copy and paste the code as it will not work due to the errors.)

**Syntax Error:** Occurs when code violates the rules of Python syntax.   
**Example:** We forgot to close the String quote.

print("Hello World!)

**Name Error:** This happens when a variable or name is not defined or misspelled.   
**Example**: We spell the variable we want to print incorrectly.

my\_name = "Alice"

print("Hello, " + myname)

**Value Error:** Occurs when a function receives an argument of the correct data type but an inappropriate value.   
**Example:** We want to turn a sequence of characters into an integer.

my\_string = "Hello World"

print(int(my\_string))

**Index Error:** This happens when trying to access an element of a sequence that is out of range.   
**Example:** We want to access the fourth element (index is 3) but the list only has three items.

fruits = ["apple", "banana", "orange"]

print(fruits[3])

**Indentation Error:** This occurs when the code's indentation is incorrect.   
**Example**: We have a conditional, but have not indented the code correctly.

if 5 > 2:

print("Five is greater than two!")

Often enough, errors are highlighted in your code editor by either red or yellow squiggly lines as in the following example:

A black background with white text

Description automatically generated

Pay close attention to those lines to pick up errors and fix them quickly.

**Task - Fixing Errors:** Copy and paste the following code snippets into your code one by one. Run each and then find and fix the error.

* Fix the **syntax error** in the code below so it prints "Hello, World!"

pritn("Hello, World!")

**Name Error:**

* Correct the name error by defining the missing variable to print "My favorite color is Blue."

print("My favorite color is", favorite\_color)

**Value Error:**

* Fix the value error by changing the string to an integer so the sum is correctly calculated and printed.

number1 = "5"

number2 = 3

result = number1 + number2

print("The sum is:", result)

**Index Error:**

* Correct the index error by accessing the second element (index 1) of the list and printing it.

fruits = ["apple", "banana", "cherry"]

print(fruits[3])

**Indentation Error:**

* Fix the indentation error so the code correctly prints "Good morning!" when the time is before 12:00.

time = 11

if time < 12:

print("Good morning!")

3. Your first larger-scale Python programme

In this exercise, you will create a simple to-do list program using Python. You will use variables, lists, input, loops, functions, and conditionals to build a basic but functional to-do list manager.

**Task To-Do list manager:**

You need to create a to-do list manager with the following functionalities:

1. Initialize an empty list to store tasks.
2. Implement a menu that allows the user to perform the following actions:

* Add a new task to the list.
* View the current tasks in the list.
* Remove a task from the list.
* Quit and exit the program.

1. Use a while loop to repeatedly display the menu and handle user input.
2. Create functions for adding, viewing, and removing tasks.
3. Use conditionals to execute the appropriate function based on the user's choice.
4. Display a message if the user tries to remove a task that doesn't exist.
5. Exit the program when the user chooses to quit.

Create a new file called to\_do\_week\_3.pyand use the following (*incomplete*) template to get started.

When you work on the script it is beneficial to run it in intervals to see if the code you have implemented works.

# Initialize an empty list to store tasks

tasks = # TODO: Add code here to initialize an empty list

# Function to add a task to the list

def add\_task():

    pass  # TODO: Add code here to add a task to the list

    # Don't forget to add arguments to the function, if needed

# Function to view current tasks in the list

def view\_tasks():

    pass  # TODO: Add code here to view all tasks in the list

# Function to remove a task from the list

def remove\_task():

    pass  # TODO: Add code here to remove a task from the list

# Main program loop

while True:

    print("To-Do List Manager")

    print("1. Add a task")

    print("2. View tasks")

    print("3. Remove a task")

    print("4. Quit")

    choice = input("Enter your choice: ")

    if choice == "1":

        pass # TODO: Add code for this option here and replace the pass statement

    # TODO: Add code for other menu options here

    else:

        print("Invalid choice. Please try again.")

**Hints:**

* You can (and should) add function arguments as necessary.
* You can use the append() method to add tasks to the list.
* To view tasks, you can loop through the list and print each task with a numerical index.
* To remove a task, consider using the remove() method or handle it with a loop and conditional statements.