# COMP11124: Object Oriented Programming

## Week 3 Practical Lab – Functions, Scope, Errors, and To-Do List

## Exercise 1: Functions in Python

### 1.1. Greet Friends

```
def greet_friends(friend_names):
    for single_friend in friend_names:
        print(f"Greetings, {single_friend}! Welcome to the club.")
friend_group = ["John", "Jane", "Jack"]
greet_friends(friend_group)
```

**Sample Output:**

```
Greetings, John! Welcome to the club.
Greetings, Jane! Welcome to the club.
Greetings, Jack! Welcome to the club.
```

**Explanation:**
This exercise presents the way to define and use functions in Python. Use of functions makes programs readable and maintainable, since code can be divided into blocks that can be reused. In this instance a function to greet all friends in a list is defined, showing how to pass arguments and how to use loops within functions. The variety of functions will eliminate redundancy of code and enhance modularity. Knowledge of functions is necessary in order to organize bigger programs. Learning of functions is a QRS activity that makes one write productive and scalable code.

---

### 1.2. Tax Calculation

```
def calculate_income_tax(annual_income, rate_of_tax):
    return annual_income * rate_of_tax
tax_result_1 = calculate_income_tax(52000, 0.18)
print(f"Income tax for £52,000 at 18%: £{tax_result_1}")
tax_result_2 = calculate_income_tax(83000, 0.27)
print(f"Income tax for £83,000 at 27%: £{tax_result_2}")
```

**Sample Output:**

```
Income tax for £52,000 at 18%: £9360.0
Income tax for £83,000 at 27%: £22410.0
```

**Explanation:**
This exercise shows the use of functions in the calculation of parameters. The calculate tax function has two parameters: income and tax rate, and outputs the tax charged. This method allows the reusing of the code with other values. Computations with the help of parameters can be dynamic and flexible. This kind of functionality is typical in business and finance. The knowledge of parameterized functions is important to solve complex problems programmatically. It further promotes proper coding techniques where separation of logic and data takes place.

---

### 1.3. Compound Interest Calculator

```python
def calculate_compound_interest(start_amount, years, yearly_rate):
    if yearly_rate < 0 or yearly_rate > 1:
        print("Interest rate must be between 0 and 1 (decimal form)")
        return None
    if years < 0:
        print("Years must be a positive integer")
        return None
    for current_year in range(years + 1):
        total_amount = start_amount * (1 + yearly_rate) ** current_year
        print(f"Year {current_year}: Investment value is £{total_amount:.2f}")
    return int(total_amount)
final_value = calculate_compound_interest(1200, 4, 0.04)
print(f"Total value after 4 years: £{final_value}")
```

**Sample Output:**

```
Year 0: Investment value is £1200.00
Year 1: Investment value is £1248.00
Year 2: Investment value is £1297.92
Year 3: Investment value is £1349.84
Year 4: Investment value is £1403.83
Total value after 4 years: £1403
```

**Explanation:**
This practice demonstrates the process of constructing a more complicated procedure including input verification and iteration. The program displays compound interest after many years; where the number is printed at each year. Input checks are provided so that the error could be avoided because the function is used properly. Repeated calculations are shown by the loop to be inserted in the function. Such tendency is typical in finance modeling and simulations. comprehending the way of combining validation, loops, and calculations in functions is critical to excellence in programming. It also shows the significance of user feedback as well as error control.

## Exercise 2: Variable Scope

```python
global_counter = 10
def scope_demo_function():
    local_counter = 25
    print(f"Inside function: local_counter = {local_counter}")
scope_demo_function()
print(f"Outside function: global_counter = {global_counter}")
```

**Sample Output:**

```
Inside function: local_counter = 25
Outside function: global_counter = 10
```

**Explanation:**
The given exercise studies the Python variable scopes concept. The variables configured within a function belong to that functionality, whereas the global ones belong to all. As displayed in the example, when a variable in a function is changed it does not change the variable of the same name globally. Knowledge of scope avoids bugs and other unwanted side-effects in your code. It plays a major role in data management and control over variables access. Opinionatedness in scope is crucial in the writing of unambiguous and foreseeable programs. It also establishes operating principles of more advanced things such as closures and namespaces.

---

## Exercise 3: Assertions

```python
assert calculate_compound_interest(1200, 4, 0.04) == 1400, "Compound interest calculation did not match expected value"
```

**Sample Output:**

```
Year 0: Investment value is £1200.00
Year 1: Investment value is £1248.00
Year 2: Investment value is £1297.92
Year 3: Investment value is £1349.84
Year 4: Investment value is £1403.83
```

(No output if assertion passes; error message if it fails.)

**Explanation:**
The assertions are applied to validate one of the conditions in your code. In case of the condition being false an error with a given message is raised in the program. It is an easy method of bug-hunting and securing your

code acts the way you want it to. Testing and debugging often use assertions. They assist you with checking assumptions and ironing out errors promptly. Assertion enhances reliability of codes and can be trusted in the yield. Both the beginner and expert programmers can find them useful.

---

## Exercise 4: Fixing Common Errors

### 4.1. Fixed Syntax Error

```
print("Welcome to Python error fixing!")  # Fixed: Added missing quote
```

**Sample Output:**

```
Welcome to Python error fixing!
```

**Explanation:**
Syntax errors are the cases when the code violates the rules of Python, missing quotes, parentheses, etc. The below example indicates how to correct a missing quote in a print statement. Being able to filter and fix the syntax errors is a simple enough needed skill. The Python interpreter normally detects syntax errors before running the program. It is important to learn reading the error messages in order to know how to locate and solve problems quickly. The most basic requirement to writing proper code is first to master syntax. It also instills confidence on troubleshooting and debugging.

---

### 4.2. Fixed Name Error

```
favorite_food = "Pizza"
print("My favorite food is", favorite_food)  # Fixed: Defined favorite_food
```

**Sample Output:**

```
My favorite food is Pizza
```

**Explanation:**
Name errors occur when you attempt to utilise a variable that is not defined. This example corrects this problem of defining the variable before its use. acquaintance with variable definition and range avoids the name errors. The name errors may be easily correctable by the new users especially when a lot of care is taken. It is important to ensure that variables are spelled correctly and defined before it could be used. Correcting name bugs is one of the practices of good coding habits. It also stimulates clean and tidy code.

---

### 4.3. Fixed Value Error

```
first_number = 12  # Fixed: Converted string to integer
second_number = 8
sum_result = first_number + second_number
print("The total sum is:", sum_result)
```

**Sample Output:**

```
The total sum is: 20
```

**Explanation:**
Problems in value errors can arise when a given function has a suitable type inserted in it, but the unsuitable value. In this case the mistake is corrected by in addition of variables being integer first. Data type and conversion are very important areas where one might encounter some value mistakes. Never just take the input of a user; this should be verified and converted accordingly. With a wise input handling and a testing, value errors can be avoided. Correction of these errors will enhance robustness of programs. It also assists you to write a code that is remedial in unexpected cases.

---

### 4.4. Fixed Index Error

```
fruit_basket = ["apple", "banana", "cherry"]
print(fruit_basket[2])  # Fixed: Accessed index 2 instead of 3
```

**Sample Output:**

```
cherry
```

**Explanation:**
Index errors occur when you attempt to access an element of a list which is out of range. This instance corrects the mistake with the help of an effective index. The indexing of Python Lists is zero-based, and therefore viable indexes are of the form 0 to length-1. Always initiate a time of checking the size of a list before the elements are accessed. Index errors usually happen but are easy to correct by careful coding. A basic concept needed when dealing with collections is the understanding of list indexing. Index errors are also prevented thus producing more dependable bug-free programs.

---

### 4.5. Fixed Indentation Error

```
current_hour = 15
if current_hour < 18:
    print("Good afternoon!")  # Fixed: Proper indentation
```

**Sample Output:**

```
Good afternoon!
```

**Explanation:**
Indentation mistakes happen when there is no proper alignment of the handlings of the code blocks. Python employs indentation in determining the layout in the form of code, including loop and conditionals. The mistake is corrected in this example by indention of the print statement. The regular indentation determines the code clarity and functionality. Indentation bugs are typical of new comers but simple to prevent by good habits. Indentation is the crucial aspect in writing a valid Python code. It is easier to read and maintain your code too.

---

## Exercise 5: To-Do List Manager

```python
# Initialize an empty list to store tasks
user_tasks = []

def add_new_task(task_description):
    user_tasks.append(task_description)
    print(f"Added new task: '{task_description}' to your to-do list!")

def display_all_tasks():
    if not user_tasks:
        print("Your to-do list is currently empty.")
    else:
        print("Here are your current tasks:")
        for idx, task_item in enumerate(user_tasks, 1):
            print(f"Task {idx}: {task_item}")

def delete_task_by_index(task_number):
    if 1 <= task_number <= len(user_tasks):
        removed = user_tasks.pop(task_number - 1)
        print(f"Successfully removed: '{removed}' from your to-do list.")
    else:
        print("That task number is not valid. Please try again.")

# Main program loop
while True:
    print("\nPersonal To-Do List Manager")
    print("1. Add a new task")
    print("2. Show all tasks")
    print("3. Delete a task")
    print("4. Exit")

    user_choice = input("Select an option (1-4): ")

    if user_choice == "1":
        new_task = input("Describe the new task: ")
        add_new_task(new_task)
    elif user_choice == "2":
        display_all_tasks()
    elif user_choice == "3":
        display_all_tasks()
        try:
            task_to_remove = int(input("Enter the number of the task to delete:
"))
            delete_task_by_index(task_to_remove)
        except ValueError:
            print("Please enter a valid task number.")
    elif user_choice == "4":
        print("Thank you for using your personal to-do list. Goodbye!")
        break
    else:
        print("Invalid selection. Please choose a valid option.")
```

**Sample Output:**

```
Personal To-Do List Manager
1. Add a new task
2. Show all tasks
3. Delete a task
4. Exit
Select an option (1-4): 1
Describe the new task: Complete Python lab
Added new task: 'Complete Python lab' to your to-do list!

Personal To-Do List Manager
1. Add a new task
2. Show all tasks
3. Delete a task
4. Exit
Select an option (1-4): 2
Here are your current tasks:
Task 1: Complete Python lab

Personal To-Do List Manager
1. Add a new task
2. Show all tasks
3. Delete a task
4. Exit
Select an option (1-4): 4
Thank you for using your personal to-do list. Goodbye!
```

**Explanation:**
This support is an easy to-do list manager that uses functions and lists. Through a menu based interface, the users can add tasks, view them and delete them. In the program, one can find an example of writing functions to achieve modularity and lists to store dynamic data. Input validation will make the program tolerant to what the user inputs. This arrangement of applications is widespread in productivity and personal organizers.

Construction of such projects assists in cementing the fundamental concepts of programming. It also gives a working experience in designing interactive programs.

---

**Conclusion:**

During the lab, you learned more complex concepts in Python such as functions, variable scope, assertion, error handling, and making a to-do list manager. Every exercise aimed at making you better comprehend the way to organize, test and debug your code. When you practice these skills, you increase your readiness to handle sophisticated programming tasks in future and work on software in the real world. Note that, practice and attention to details will be vital in learning how to program. Continue with experimenting and developing your projects and become a competent confident Python developer.