COMP11124: Object Oriented Programming

Week 2 Practical Lab – Python Basics

1. Comparison and Conditional Statements

1.1. Using Comparison Operators

```
# Demonstrating comparison operators
a = 10
b = 20
print("Is a equal to b?", a == b)
print("Is a not equal to b?", a != b)
print("Is a less than or equal to b?", a <= b)
print("Is a greater than or equal to b?", a >= b)
```

Explanation:
Comparison operators are those that would compare two values and give you a result as True or False. They play a key role in decision making in codes e.g. whether two variables are equal or whether a variable is greater or smaller than the other. Such operators are == (equals), != (not equal), < (less than), > (greater than), <= (les than or equal), and >= (greater than or equal). Through comparisons operators, you can be able to flow your program in different conditions. The loops and the conditional statements are essential to them. The knowledge of these operators is essential in producing well-written and sensible pieces of code.

1.2. Logical Operators in Action

```
# Logical operators: and, or, not
age = 25
print("Is age between 20 and 30?", age > 20 and age < 30)
print("Is age not a teenager?", not (age >= 13 and age <= 19))
print("Is age an adult or a senior?", age >= 18 or age > 65)
```

Explanation:
Logical operators will enable you to have several conditions in one statement. The `and` variable will give true when both conditions are true whereas the `or` will give true when at least one of the conditions is true. The `not` operator prints the opposite of a condition. These operators are very strong in constructing complicated decision making logic in your programs. They are also mainly employed in if condition, looping and input checks. Logical operators are necessary to enable you to write more versatile and strong scripts. They also help to manage various requirements and smarten up your programs.

1.3. Simple if Statement

```
# Using if to categorize age
age = 19
if age > 18:
    group = "adult"
else:
    group = "child"
print("Age group:", group)
```

Explanation:
The `if` statement is an essential Python control statement which enables you to perform some statement only when a given condition is valid. In the given example, the program then asks the question of whether the age exceeds 18 or not in order to reveal whether the individual is an adult or a child. When it fails to meet the condition, that code in the `else` block executes. This design is critical in taking decisions and regulating the flow of your program. It assists you to generate interactive and responsive applications. Creating dynamic and practical programs largely depend on the use of `if` statements.

1.4. if-else Example

```
# Deciding based on wind speed
wind = 30
if wind < 10:
    print("Calm weather")
else:
    print("Windy weather")
```

Explanation:
The if-else structure allows you the option of selecting among two courses of action given an event. When the condition is actualized, the first block of code is executed, whereas the code under the keyword `else` is executed in other cases. This is beneficial when dealing with binary decisions i.e. deciding whether weather is windy or calm. The `if-else` statements will allow you to make your code more universal. It is the fundamental principle of programming logic. Learning about the usage of `if-else` may guide you to program such a way that it can respond to variable inputs and conditions.

1.5. if-elif-else Chain

```
# Grading system
score = 55
if score < 50:
    print("Fail")
elif score < 60:
    print("Pass")
elif score < 70:
    print("Good pass")
```

```
else:
    print("Excellent pass")
```

Explanation:
The chain of `if-elif-else` enables you to test two or more conditions successively. The `elif` statements give another expression to evaluate whether the expressions that precede them are false. This can very well be applied when putting some data together, i.e. determining grades depending on a score. It will look more readable and organized when you are covering multiple possibilities of what could have gone wrong. Taking advantage of `if-elif-else` will prevent you to have too many nested if statements. It is a recommended practice of dealing with multiple decision branches. Levels of competence of this construction will help in writing readable and maintainable code.

1.6. Comparing Two Temperatures

```
# Checking if two temperatures are the same
temp1 = 22.5
temp2 = 25.0
if temp1 == temp2:
    print("Temperatures match")
else:
    print("Temperatures differ")
```

Explanation:
In this exercise, we learn how to compare two values to be equal to each other thus using operator `==`. The topic of determining whether two variables have the same value or not is quite popular in programming: a programmer needs it when comparing the temperatures or comparing the scores of a game or when checking user inputs. Which message printed lies on the outcome of comparison. The concept finds a wide application in data validations and error checks and also in decision making. It is also fundamental to the construction of dependable programs to have an idea of how to make valid comparisons of values. It can also assist in debugging and get your code to act as projected.

---

2. Working with Lists

2.1. Creating a List

```
# Making a list of cities
cities = ["Glasgow", "London", "Edinburgh"]
print("Cities:", cities)
```

Explanation:
One of the basic data structures in Python is lists, which enable you to keep several items within one variable. This code demonstrates creation of a list and print statement of the contents of a list. Lists are sortable, alterable, and make it possible to store items of any data type. They can be used to collect together similarly

related data, e.g. names of cities, scores, or objects. It is necessary to know the process of creating and manipulating lists to ensure that the data can be governed in your programs. Lists offer efficient algorithms of addition, removal and access to items. List manipulation is one of the secrets of working with data in Python.

2.2. Accessing List Elements

```
# Accessing items in a list
print("Third city:", cities[2])
print("Last two cities:", cities[-2:])
```

Explanation:
The elements of a list are accessed with the help of indices and the initial one is at index 0. This practice illustrates the manner in which one can extract particular items and slices of a list. The tools indexing and slicing are very useful in manipulating subsets of data. Negative indices enable you to number the list backwards. Knowing the process of access and manipulation of elements of the list is the key to processing the data. It allows you to extract, alter or examine certain portions of your data. A good mastery of list indexing enables you to make more flexible and efficient code.

2.3. Modifying Lists

```
# Adding and changing items
cities.append("Manchester")
cities[1] = "Birmingham"
print("Updated cities:", cities)
```

Explanation:
Python lists are changeable, that is, once created they can be manipulated. This practice demonstrates the inclusion of a new object using `append()` and the change to an existing object using an index to a new value. Dynamic data management uses manipulation of lists, including updating records to construct collections as they go. The concept of list mutability can assist you in making programs that can handle dynamic data. It also provides efficient usage of memory and organization of data. The knowledge of the list operations is crucial to various programs.

2.4. List Summary Task

```
# Practice with lists
colors = ["blue", "green", "yellow"]
print("Original:", colors)
print("Second color:", colors[1])
colors[0] = "red"
print("Changed:", colors)
print("List length:", len(colors))
if "red" in colors:
```

```
    print("Red is present")
print("Selected colors:", colors[1:3])
```

Explanation:
This summary operation is a multiple-list operation, an operation that accesses list, modifies list, checks list membership and slices. It strengthens your knowledge of how to manipulate lists in situations. Verifying that a particular item is or is not there in a list is applied in validating and searching operations. Slicing will enable you to deal with sublists effectively. The `len()` method will give the quantity of objects in a list, and this is valuable in regards to loops and analysis of data. Such operations training will prepare you to more complicated data structures and algorithms. Python programming is based on lists.

---

3. Looping in Python

3.1. While Loop Example

```
# Counting up with while
i = 0
while i < 5:
    print(i)
    i += 1
```

Explanation:
A `while` loop is used when you want to write a block of code repeatedly until a certain condition has been met. This example has a loop that is 0 to 4 with an incrementing variable. Whereas, while loops are used in cases when it is not yet known how many iterations will be performed. They often serve to validate an input, to wait on an event or to parse data until some condition is satisfied. Comprehension of the loop control is a key to cope with the most efficient and reactive program. Use of loops properly will help you avoid infinite loops and mix up of codes, so that they work according to your desires.

3.2. For Loop Example

```
# Looping through a list
for city in cities:
    print(city)
```

Explanation:
The `for` loop is applied to loop through a sequence, like list, tuple, or a string. This is printing all cities concerning the list. The for loops are good when you have the number of items to be processed. They help to make your codes to be compact and easy to read especially when using collections of data. Knowledge in iteration is essential in data analysis, automation and other numerous programming activities. Other structures can also be used together with for loops to have complex logic. The loops are the basic skill of any programmer.

3.3. Using break and continue

```
# Using break
for i in range(5):
    if i == 3:
        break
    print(i)

# Using continue
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Explanation:
The `break` and `continue` statements have their control over looping. `break` jumps out of the loop suing a condition is met, whereas `continue` does not operate anything other than a remainder of the current iteration but passes to the next one. These tools may be applied in form of addressing special cases, providing optimizations of performance or in management of errors. When you use break and continue appropriately your loops become flexible and efficient. They become able to deny you extra computations and make the code easier to follow. It is very significant to comprehend such statements in complex loop control.

3.4. Loop Summary Tasks

Even Numbers:

```
nums = [1,2,3,4,5,6,7,8,9,10]
for n in nums:
    if n % 2 == 0:
        print(n)
```

Sum of Squares:

```
total = 0
for i in range(1, 6):
    total += i ** 2
print("Sum of squares:", total)
```

Countdown:

```
count = 10
while count >= 1:
    print(count)
```

```
    count -= 1
print("Liftoff!")
```

Explanation:
These exercises give an opportunity to train with loops and conditionals. The use of modulus operator and usage of conditions also strengthens when printing even numbers. The addition of squares shows the addition in a loop. The example of countdown demonstrates a technique of incrementing a value as well as using a loop to define the control flow in a program. These activities instill your confidence in dealing with loops to achieve various goals. Proper usage of loops and conditionals will allow one to solve a great number of programming problems. These patterns can be practised to get you ready to more advanced algorithms.

---

4. User Input and Conditionals

4.1. Age Group Checker

```
# Ask user for age and categorize
age = int(input("Enter your age: "))
if age < 18:
    print("You are a minor.")
elif age <= 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

Explanation:
The exercise will show how we can get what the user enters and make some conditional statements. The game requests the age of a user and classifies him or her as a minor, adult, or senior citizen. Interaction with the user and input checking are the features of practical programs. The input of `input()` enables you to practise dynamic and responsive programming. Conditional logic is used to make sure that one gets the right message depending on what the user has typed in. The knowledge of conditionals and user input is essential in the construction of interacting applications. It also assists you in the development of programs that are easier and flexible to use.

4.2. Temperature Converter

```
# Convert temperature between units
unit = input("Enter temperature unit (C, F, K): ").upper()
value = float(input("Enter temperature value: "))

if unit == "C":
    print(f"{value}°C = {(value * 9/5) + 32}°F")
    print(f"{value}°C = {value + 273.15}K")
elif unit == "F":
    c = (value - 32) * 5/9
    print(f"{value}°F = {c}°C")
```

```
        print(f"{value}°F = {c + 273.15}K")
    elif unit == "K":
        c = value - 273.15
        print(f"{value}K = {c}°C")
        print(f"{value}K = {(c * 9/5) + 32}°F")
    else:
        print("Invalid unit. Please use C, F, or K.")
```

Explanation:
This drill demonstrates how to intertwine user prompting, conditional statements and mathematics to provide a solution to a real-life problem: temperature conversion. The program prompts the user with two items, a temperature unit and a number and does the corresponding conversion. Dealing with various units would entail appropriate application of the statements if-elif-else. Conversion formulas are implemented by arithmetic operations. Output can also be clear so that the user will have a meaning of the results. These are typical of scientific/ engineer application programs. It will be very important to master input, conditionals and calculations in relation to real programming.

Conclusion:

In this lab, you mastered some of the basic concepts of Python such like comparison and logical operators, conditionals, lists, loops, and user input. You completed an exercise after the other and each one of them represented a structure to the other making you establish a good foundation of programming logic and problem-solving. Exercising the skills you are taught through the above can help jostle you to take on even tougher subjects on object-oriented programming and software development in the real sense. The thing is to remember that the way to learn programming is to practice and experiment. Through more application of such basics, you will become a good and confident Python programmer.