1. Introduction

Like human beings, computers must be able to repeat a set of actions. They also must be able to select an action to perform in a particular situation. This unit focuses on control statements – statements that allow the computer to select or repeat an action.

2. Selection Statements

In some cases, instead of moving straight ahead to execute the next instruction, the computer might be faced with two alternative courses of action. The computer must pause to examine or test a condition. If the condition is true, the computer executes the first alternative action and skips the second alternative. If the condition is false, the computer skips the first alternative action and executes the second alternative.

Instead of moving blindly ahead, selection statements allow a computer to make choices after examining a test condition.

2.1. The *if* Statement

The simplest form of selection is the **if statement**. This type of control statement is also called a **one-way selection statement**, because it consists of a condition and just a single sequence of statements. If the condition is True, the sequence of statements is run. Otherwise, control proceeds to the next statement following the entire selection statement. The syntax of the if statement is given below.

```
if <condition>:
   <sequence of statements>
```

For example,

```
amount = float(input("Purchased amount:"))
discount = 0
if amount >= 1000:
   discount = amount * 0.05
print(f"Discount = {discount}")
```

2.2. The *if-else* Statement

The **if-else statement** is the most common type of selection statement. It is also called a **two-way selection statement**, because it directs the computer to make a choice between two alternative courses of action. The syntax of the if-else statement is given below.

```
if <condition>:
   <sequence of statements–1>
else:
   <sequence of statements–2>
```

For example,

```
amount = float(input("Purchased amount:"))
discount = 0
if amount >= 1000:
   discount = amount * 0.05
else:
   discount = amount * 0.03
print(f"Discount = {discount}")
```

2.3. The Multiway *if* Statement

Occasionally, a program is faced with testing several conditions that entail more than two alternative courses of action. The process of testing several conditions and responding accordingly can be described in code by a **multi-way selection statement**. The syntax of the if-else statement is given below.

```
if <condition-1>:
   <sequence of statements-1>
elif <condition-n>:
   <sequence of statements-n>
else:
   <default sequence of statements>
```

For example,

```
amount = float(input("Purchased amount:"))
discount = 0
if amount >= 5000:
   discount = amount * 0.1
elif amount >= 4000:
   discount = amount * 0.07
elif amount >= 3000:
   discount = amount * 0.05
elif amount >= 2000:
   discount = amount * 0.03
else:
   discount = amount * 0.02
print(f"Discount = {discount}")
```

Remember: If you have only one statement to execute, you can put it on the same line as the if statement. For example,

```
a = 200
b = 33
if a > b: print("a is greater than b")
```

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line. For example,

```
a = 2
b = 330
print("A") if a > b else print("B")
```

This **single line if-else** statement can also be used to realize the concept of **ternary operator** in python. For example,

```
a, b = 10, 20
min = a if a < b else b
print(min) # 10
```

You can also have multiple else statements on the same line. For example,

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

2.4. The *match-case* Statement

This **match-case** is the switch-case of Python which was introduced in Python 3.10 used for structural pattern matching. Here we have to first pass a parameter then try to check with which case the parameter is getting satisfied. If we find a match, we will do something and if there is no match at all we will do something else. The general syntax of match-case is given blow:

```
match term:
    case pattern-1:
        action-1
    case pattern-2:
        action-2
    case pattern-n:
        action-n
    case _:
        action-default
```

For example,

```
month = input('Enter month:')
match month:
   case 'January':
      print('31 days.')
   case 'February':
      print('28 or 29 days.')
   case 'March':
      print('31 days.')
   case 'April':
      print('30 days.')
   case 'May':
      print('31 days.')
   case 'June':
      print('30 days.')
   case 'July':
      print('31 days.')
   case 'August':
      print('31 days.')
   case 'September':
      print('30 days.')
   case 'October':
      print('31 days.')
   case 'November':
      print('30 days.')
   case 'December':
      print('30 days.')
   case _:
      print('Wrong month.')
```

If you want to achieve the same behavior as fall-through in a traditional switch statement, you can use the | operator to match multiple cases. This operator allows you to specify multiple patterns in a single case, and any of the patterns can be matched to trigger the execution of that branch. For example,

```
month = int(input("Enter month(1-12):"))
match month:
   case 1 | 3| 5 | 7 | 8 | 10 | 12:
      print("31 days")
   case 2:
      print("28 or 29 days")
   case 4 | 6 | 9 | 11:
      print("30 days")
   case _:
      print("Wrong month")
```

Remember: We do not add a break keyword to each of the cases, as it is done in other programming languages. That's the advantage Python's native switch statement has over those of other languages. The break keyword's functionality is done for you behind the scenes.

3. Looping Statements

These statements are also called repetition statements which repeat an action. Each repetition of the action is known as a pass or an iteration. There are two types of loops – those that repeat an action a predefined number of times (definite iteration) and those that perform the action until the program determines that it needs to stop (indefinite iteration).

3.1. The *for* Loop

A **for** loop is used for iterating over a collection (list, tuple, dictionary, set, string etc.). With the for loop we can execute statements, once for each item in the collection. The syntax of the if-else statement is given below.

```
for <variable> in <collection>:
   <do something with variable>
```

For example,

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

To loop through a set of code a specified number of times, we can use the **range()** function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. For example,

```
fruits = ["apple", "banana", "cherry"]
for x in range(3):
   print(fruits[x])
```

The **else** keyword in a for loop specifies a block of code to be executed when the loop is finished. For example,

```
fruits = ["apple", "banana", "cherry"]
for x in range(3):
   print(fruits[x])
else:
   print("Finally finished")
```

Remember: The **range()** function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter, **range(2, 6)**, which means values from 2 to 6 (but not including 6). The *range()* function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter such as **range(2, 30, 3)**. The *range()* function can also be used to count down from an upper bound to a lower bound such as **range(10, 0, -1)**.

3.2. The *while* Loop

The ***for loop*** executes a set of statements a definite number of times specified by the programmer. In many situations, however, the number of iterations in a loop is unpredictable. With the **while loop** we can execute a set of statements as long as a condition is true. The syntax of the if-else statement is given below.

```
while <condition>:
   <sequence of statements>
```

For example,

```
fruits = ["apple", "banana", "cherry"]
x = 0
while x < len(fruits):
   print(fruits[x])
   x = x + 1
```

With the else statement we can run a block of code once when the condition no longer is true. For example,

```
fruits = ["apple", "banana", "cherry"]
x = 0
while x < len(fruits):
   print(fruits[x])
   x = x + 1
else:
   print("Finally finished")
```

3.3. The *break* and *continue* Statements

The use of **break** statement causes the immediate termination of the loop (all type of loops) and the passage program control to the statements following the loop. In case of nested loops if break statement is in inner loop only the inner loop is terminated. For example,

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

The **continue** statement causes the execution of the current iteration of the loop to cease, and then continue at the next iteration of the loop. For example,

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
```

```
print(x)
```

4. Nested Control Statements

We can nest any control statement (selection or looping statement) within another to any depth. When one loop is nested inside another loop, the inner loop is first terminated and again restarted when the first loop starts for the next value. The outer loop is terminated last. For example,

```
for x in ['a', 'b']:
    print(x)
    for y in range(1,4):
        print(y)
```

5. The *pass* Statement

The ***pass*** statement in Python is used as a placeholder for future code. When the pass statement is executed, nothing happens, but you avoid getting an error when empty code is not allowed. Empty code is not allowed in loops, function definitions, class definitions, or in if statements. For example

```
a = int(input("Amount = "))
d = 0
if a >= 1000:
    d = a * 0.05
else:
    pass
print(f"Discount = {d}")
```