Applied Physics Lab -3

Dictionary data type

A **defaultdict** is a dictionary with a default value for keys, so that keys for which no value has been explicitly defined can be accessed without errors. A **defaultdict** is especially useful when the values in the dictionary are collections (lists, dicts, etc) in the sense that it does not need to be initialized every time when a new key is used.
A **defaultdict** will never raise a **KeyError**. Any key that does not exist gets the default value returned.
Dictionary consists of key-value pairs. It is enclosed by curly braces { } and values can be assigned and accessed using square brackets [ ].

```
In [78]: dic={'name':'Redrose','age':10}
         print(dic) # will output all the key-value pairs. {'name':'Redrose','age':10}

            {'name': 'Redrose', 'age': 10}

In [79]: print(dic['name']) # will output only value with 'name' key. 'Redrose'

            Redrose

In [80]: print(dic.values()) #will output list of values in dic. ['Redrose',10]

            dict_values(['Redrose', 10])

In [81]: print(dic.keys()) #will output list of keys. ['name','age']

            dict_keys(['name', 'age'])
```

Condition Statements

The conditional statement checks to see if a statement is **True** or **False**. That's really all it does. However we will also be looking at the following Boolean operations: and, or, and not. These operations can change the behavior of the conditional in simple and complex ways, depending on your project. Python programming language assumes any **non-zero** and **non-null** values as **TRUE**, and any **zero** or **null** values as **FALSE** value.

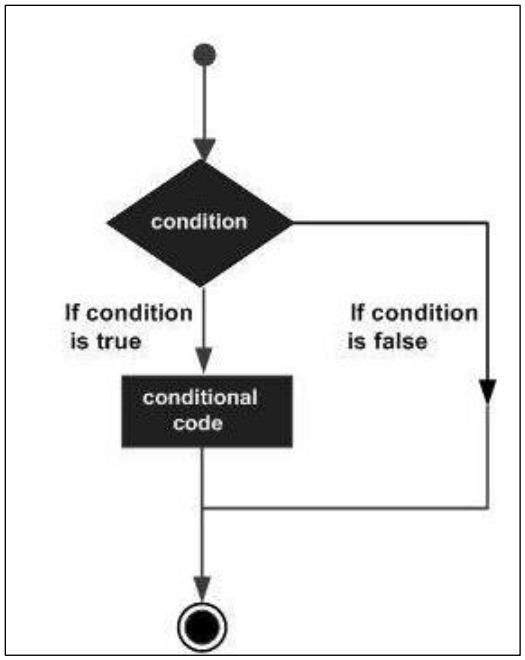

Figure 1: Decision making structure

The IF Statement

The IF statement is similar to that of other languages. The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

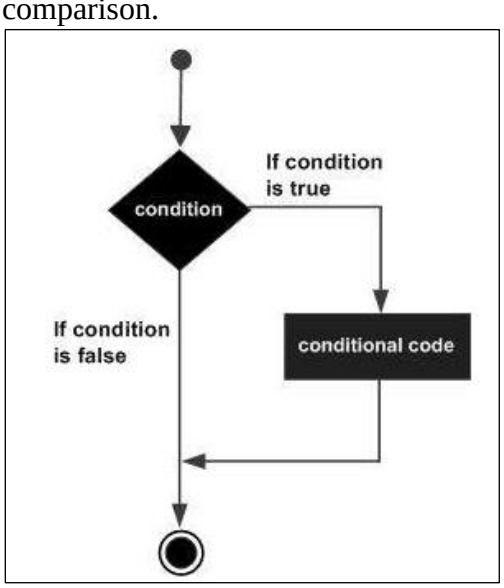

Figure 2: Flow diagram of if statement

```
In [1]: if 2 > 1:
            print("This is a True statement!")

This is a True statement!
```

This conditional tests the "**truthfulness**" of the following statement: 2 > 1. Since this statement evaluates to True, it will cause the last line in the example to print to the screen or standard out.

```
In [2]: var1 = 1
        var2 = 3
        if var1 > var2:
        print("This is also True")

  File "<ipython-input-2-1774bc7236f3>", line 4
    print("This is also True")
        ^
IndentationError: expected an indented block
```

IF / ELSE Statement

An **else** statement can be combined with an **if** statement. An **else** statement contains a block of code that executes **if** the conditional expression in the **if** statement resolves to **0** or a **FALSE** value. The **else** statement is an optional statement and there could be at the most only one **else** statement following **if**.

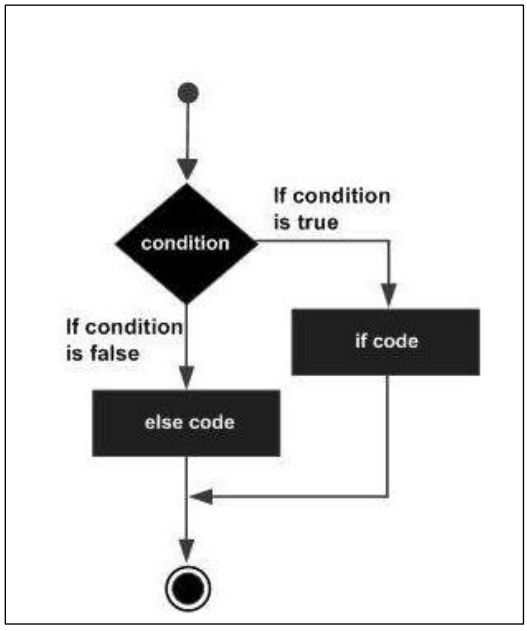

Figure 3 : Flow Diagram of if./ else statement

In this one, we compare two variables that translate to the question: Is 1 > 3? Obviously one is not greater than three, so it doesn't print anything. But what is we wanted it to print something? That's where the else statement comes in. Let's modify the conditional to add that piece:

```
if var1 > var2:
    print("This is also True")
else:
    print("That was False!")
```

```
That was False!
```

The ELIF Statement

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.
Similar to the else, the elif statement is optional. However, unlike else, for which there can be at the most one statement, there can be an arbitrary number of elif statements following an if.

Syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

```
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
elif amount<5000:
   discount = amount*0.10
   print ("Discount",discount)
else:
   discount = amount*0.15
   print ("Discount",discount)

print ("Net payable:",amount-discount)
```

In [2]:
```
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
elif amount<5000:
   discount = amount*0.10
   print ("Discount",discount)
else:
   discount = amount*0.15
   print ("Discount",discount)
print ("Net payable:",amount-discount)
```

```
Enter amount: 600
Discount 30.0
Net payable: 570.0
```

```
In [3]: amount = int(input("Enter amount: "))

        if amount<1000:
            discount = amount*0.05
            print ("Discount",discount)
        elif amount<5000:
            discount = amount*0.10
            print ("Discount",discount)
        else:
            discount = amount*0.15
            print ("Discount",discount)
        print ("Net payable:",amount-discount)

        Enter amount: 3000
        Discount 300.0
        Net payable: 2700.0
```

```
Enter amount: 6000
Discount 900.0
Net payable: 5100.0
```

Nested IF Statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax:

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

```
In [5]: num = int(input("enter number"))
        if num%2 == 0:
           if num%3 == 0:
              print ("Divisible by 3 and 2")
           else:
              print ("divisible by 2 not divisible by 3")
        else:
           if num%3 == 0:
              print ("divisible by 3 not divisible by 2")
           else:
              print  ("not Divisible by 2 not divisible by 3")

           enter number8
           divisible by 2 not divisible by 3
```

```
enter number15
divisible by 3 not divisible by 2

enter number12
Divisible by 3 and 2

enter number5
not Divisible by 2 not divisible by 3
```

If you run this code, it will print the string that follows the else statement. Let's change gears here and get some information from the user to make this more interesting. In Python 3.x, you can get information using a built-in called "Input".

```
In [*]: value = input("How much is that doggy in the window? ")
        value = int(value)

        if value < 10:
            print("That's a great deal!")
        elif 10 <= value <= 20:
            print("I'd still pay that...")
        else:
            print("Wow! That's too much!")

        How much is that doggy in the window? 25
```

```
How much is that doggy in the window? 25
Wow! That's too much!
```

Boolean Operation

Now we're ready to learn about Boolean operations (and, or, not). According to the Python documentation, their order of priority is first or, then and, then not. Here's how they work:

OR: means that if any conditional that is "ORed" together is True, then the following statement runs
AND : means that all statements must be True for the following statement to run
NOT : means that if the conditional evaluates to False, it is True. This is the most confusing, in my opinion.

```
In [15]: x = 10
         y = 30

         if x < 10 or y > 15:
             print("This statement was True!")

         This statement was True!
```

```
In [16]: x = 10
         y = 10
         if x == 10 and y == 15:
             print("This statement was True")
         else:
             print("The statement was False!")

         The statement was False!
```

In [18]:
```
x = 10
if x != 11:
    print("x is not equal to 11!")
```

```
x is not equal to 11!
```

In [17]:
```
my_list = [1, 2, 3, 4]
x = 10
if x not in my_list:
    print("'x' is not in the list, so this is True!")
```

```
'x' is not in the list, so this is True!
```

In [19]:
```
my_list = [1, 2, 3, 4]
x = 10
z = 11
if x not in my_list and z != 10:
    print("This is True!")
```

```
This is True!
```

In [22]:
```
empty_list = []
empty_tuple = ()
empty_string = ""
nothing = None

if empty_list == []:
    print("It's an empty list!")

if empty_tuple:
    print("It's not an empty tuple!")

if not empty_string:
    print("This is an empty string!")

if not nothing:
    print("Then it's nothing!")
```

```
It's an empty list!
This is an empty string!
Then it's nothing!
```

1.1 Loop

The Python world has two types of loops:

- The for loop and
- The while loop

You will find that the for loop is by far the most popular of the two. Loops are used when you want to do something many times. Usually you will find that you need to do some operation or a set of operations on a piece of data over and over. This is where loops come in. They make it really easy to apply this sort of logic to your data.

1.5.1 The FOR loop

A for loop is used when you want to iterate over something n number of times. It's a little easier to understand if we see an example. Let's use Python's built-in range function. The range function will create a list that is n in length.

```
In [31]: range(5)
Out[31]: range(0, 5)

In [32]: list(range(1, 10, 2))
Out[32]: [1, 3, 5, 7, 9]
```

range(start, stop [, step])

```
In [47]: for i in range (0,10,2):
             print(i)

0
2
4
6
8
```

```
In [33]: for number in range(5):
                print(number)

0
1
2
3
4
```

The for loop above would be the equivalent of the following:

```
In [34]: for number in [0, 1, 2, 3, 4]:
             print(number)

0
1
2
3
4
```

```
In [39]: a_dict = {1:"one", 2:"two", 3:"three"}
keys = a_dict.keys()
keys = sorted(keys)
for key in keys:
    print(key)

1
2
3
```

```
In [40]: for number in range(10):
             if number % 2 == 0:
                 print(number)

0
2
4
6
8
```

1.5.2 The WHILE loop

The while loop is also used to repeat sections of code, but instead of looping n number of times, it will only loop until a specific condition is met. The while loop is kind of like a conditional statement. Let's look at a very simple example:

```
In [44]: i = 0
         while i < 10:
             print(i)
             i = i + 1

         0
         1
         2
         3
         4
         5
         6
         7
         8
         9
```

Here's what this code means: while the variable i is less than ten, print it out. Then at the end, we increase i's value by one. If you run this code, it should print out 0-9, each on its own line and then stop. If you remove the piece where we increment i's value, then you'll end up with an infinite loop. This is usually a bad thing. Infinite loops are to be avoided and are known as logic errors.
There is another way to break out of a loop. It is by using the **break** built-in. Let's see how that works:

```
In [58]: j=0
         while j < 30:
             print(j)
             if j == 5:
                 break

             j += 1

         0
         1
         2
         3
         4
         5
```

You will also note that we changed how we increment the value by using + =. This is a handy shortcut that you can also use with other math operations, like subtraction (- =) and multiplication (* =).

1.5.3 What else is for in loops

The **else** statement in loops only executes if the loop completes successfully. The primary use of the else statement is for searching for items:

```
In [94]: my_list = [1, 2, 3, 4, 5]
         for i in my_list:
             if i == 3:
                 print("Item found!")
                 break
             print(i)
         else:
             print("Item not found!")
```

```
1
2
Item found!
```