Chapter 4: Control Structures and Functions

After reading this chapter, readers will be able to:

- Use iterative structures in Python
- Write conditional statements in Python
- Write nested loops
- Write simple Functions

4.1 Control Structures

"*To be or not to be, that is the question...*": In any programming language, a control structure is a block of instructions that analyzes variables and chooses a direction in which to go based on given conditions. Control structures are decision making process in programming that determines how computer will respond when given certain conditions. Control structures control "flow" of computing and decision making. These control structures include if-else statements, for loops, while loop, and nested versions of these. We will understand each of these with help of various examples.

4.1.1 If-else statements

Let us understand if-else statements with an example of car dealership. A car dealer looks at the inventory of one of their cars, say Toyota Corolla. To ensure that vehicles are always in stock, he looks at number of Corolla in stock. If number of cars is less than 20, he has to call the manufacturer and place order for next dispatch.

We can express the senario in basic programming terminology as:

- **Precondition**: Thee are 16 of vehicles in stock
- **Feeding this information through Control Structures**:

> Is number less than 20? If yes, place an order with manufacturer

> Is number greater than 20? If yes, no action is required

- **Postcondition**: Order is placed

We use **If-else** statements to make such selection in Python. Selection statements allow programmers to ask questions and then various actions can be performed based on the outcome. Let us write the code for our Toyota Corolla example. Make a note of colon and intendations (space before print statements). Python used intendations to structure its programs.

In [1]:

```
num_vehicles = 16   # Pre-condition
if num_vehicles < 20: # Notice colon at the end of if statement.
    print("Inventory low!! Place an order with manufacturer!")  # Press
else:   #Notice that if and else are at same intendation level
    print("No action required. Inventory sufficient")
```

We can also nest multiple ***if-else*** . For example, a grader wants to use conditional statements to compute grade of the students. Rather than doing it manually, he decides to write the Pyhton code. Table below shows the grading slab. Let us help the grader by writing Python codes.

Score	Grades
>90	A
81-90	B
71-80	B-
61-70	C
51-60	C-
41-50	D
<41	F

A student named Alicia secured 75 in test. Let us write code to tell her grade. We will use **elif** (else+if) to test multiple conditions.

In [2]:

```
score = 75
if score > 90:
    print("Grade is A")
elif score >81 and score <90:
    print("Grade is B")
elif score >71 and score <80:
    print("Grade is B-")
elif score >61 and score <=70:
    print("Grade is C")
elif score >51 and score <=60:
    print("Grade is C-")
elif score >41 and score <=50:
    print("Grade is D")
elif score <40:
    print("Grade is F")
else:
    print("student absent")
```

4.1.2 For loops

Programming is of no use to us if we cannot expedite the tasks that we need to repeat. For example, grader would be very upset if he has to write codes again and again to assign grades to entire class. For example, grader wishes that he enters the ***list*** of score of 10 students and python should return a ***list*** of grades.

To accomplish this in Python, we will use ***for*** statements. ***for*** statements in Python iterates over the items of any sequence (be it list or string), in the order they appear (recall from chapter 2 that lists are ordered objects).

Let us say we have grades of 10 students (arranged in ascending ID of their enrollment numbers) as:

scores = [65.0, 98.5, 46.7, 78.5, 83.5, 56.5, 92.5, 78, 63, 88]

To iterate access all **elements in scores**, we will write python script as follows:

In [3]:

```
scores = [65.0, 98.5, 46.7, 78.5, 83.5, 56.5, 92.5, 78, 63, 88]
for score in scores:  # score holds value being accessed
    print(score)
```

We can also print the index of element being accessed during the iteration using ***enumerate*** command:

In [4]:

```
for i,score in enumerate(scores):  #variable i holds value of index; sc
    print([i,score]) # First element has index 0 because Python starts
```

If we do not want to iterate over a list or sequence of numbers, we can also iterate over a ***range***. Let us say, we want to print "Hello! World" 5 times. We can do it using for loop as follows:

In [5]:

```
for i in range(5):
    print([i, "Hello! World"])
```

4.1.3 Combining for statements and if-else statements

Now that we understand the basics of for and if-else conditions, let us combine these two with our grading example and obtain the grades of 10 students without doing manual labor. For loop will iteratively access *score in scores* for all elements in list. Then, we will evaluate grades using ***if-else*** statements.

scores = [65.0, 98.5, 46.7, 78.5, 83.5, 56.5, 92.5, 78, 63, 88]

grades = [] for score in scores: if score > 90: grades.append("Grade is A") elif score >81 and score <90: grades.append("Grade is B") elif score >71 and score <80: grades.append("Grade is B-") elif score >61 and score <=70: grades.append("Grade is C") elif score >51 and score

<=60: grades.append("Grade is C-") elif score >41 and score <=50: grades.append("Grade is D") elif score <40: grades.append("Grade is F") else: grades.append("student absent")

print(grades)

4.1.4 Continue statements and Pass statements

Grader wants to skip all those who score grade D or less.

Grader gets super-excited when somebody gets an A. So, he wants to ***break*** out of the code and see who is the first person to get an A grade (obsession,huh!). To accomplish this, we use ***break*** statements. It will terminate the for loop pre-maturely as soon as first condition of A grade is met.

In [6]:

```
scores = [65.0, 98.5, 46.7, 78.5, 83.5, 56.5, 92.5, 78, 63, 88]

grades = []
for i,score in enumerate(scores):
    if score > 90:
        grades.append([i,"Grade is A"])
        break                   # Notice the break statement here
    elif score >81 and score <90:
        grades.append([i,"Grade is B"])
    elif score >71 and score <80:
        grades.append([i,"Grade is B-"])
    elif score >61 and score <=70:
        grades.append([i,"Grade is C"])
    elif score >51 and score <=60:
        grades.append([i,"Grade is C-"])
    elif score >41 and score <=50:
        grades.append([i,"Grade is D"])
    elif score <40:
        grades.append([i,"Grade is F"])
    else:
        grades.append([i,"student absent"])


print(grades)
```

4.1.4 Continue statement

While ***break*** statement terminates and jumps out of the loop, continue statement goes to the beginning of code (without evaluating succeeding statements) and continues execution. For example, if grader does not want to print D grades, he can use ***continue*** statements. note that as soon as the condition in ***continue*** is met,code after this is not executed.

In [7]:

```
scores = [65.0, 98.5, 46.7, 78.5, 83.5, 56.5, 92.5, 78, 63, 88]

grades = []
for i,score in enumerate(scores):
    if score > 90:
        grades.append([i,"Grade is A"])
    elif score >81 and score <=90:
        grades.append([i,"Grade is B"])
    elif score >71 and score <=80:
        grades.append([i,"Grade is B-"])
    elif score >61 and score <=70:
        grades.append([i,"Grade is C"])
    elif score >51 and score <=60:
        grades.append([i,"Grade is C-"])
    elif score >41 and score <=50:
        continue # Notice the continue statement here.A
        grades.append("skipped grading")
    elif score <40:
        grades.append([i,"Grade is F"])
    else:
        grades.append([i,"student absent"])


print(grades)
```

4.1.5 Pass statement

Another way to skip grading is using pass statement. While continue statement continues with the next iteration of the loop when condition is met, pass statement does nothing (no pun intended!). These are used when no action is required. The difference between pass and continue is that pass still evaluates all the statements succeeding it. Carefully note the difference in outputs of 4.1.4 and 4.1.5. 'Skipped grading' was not printed in case of continue statement. (Why?)

In [8]:

```
scores = [65.0, 98.5, 46.7, 78.5, 83.5, 56.5, 92.5, 78, 63, 88]

grades = []
for i,score in enumerate(scores):
    if score > 90:
        grades.append([i,"Grade is A"])
    elif score >81 and score <=90:
        grades.append([i,"Grade is B"])
    elif score >71 and score <=80:
        grades.append([i,"Grade is B-"])
    elif score >61 and score <=70:
        grades.append([i,"Grade is C"])
    elif score >51 and score <=60:
        grades.append([i,"Grade is C-"])
    elif score >41 and score <=50:
        pass # Notice the continue statement here.A
        grades.append("skipped grading")
    elif score <40:
        grades.append([i,"Grade is F"])
    else:
        grades.append([i,"student absent"])


print(grades)
```

4.1.6 While statement

Another structure that is very popular in most of the programming languages (including Python!) is the while statement. The while statement is used for repeated execution as long as statement is true. Let us understand this with help of an example. Our anonymous grader wants to give some extra points to a guy who scored 65 so that his grade can be increased from C to B-. He adds one point each time till score is greater than 71. He decides to use while loop to do this in the Python. Let us see how he can achieve this:

In [9]:

```
score  =  65.0 #Initialization

while score <= 71:
    score  = score +1 #Notice the intendation between while and score
print(score)
```

4.2 Functions in Python

By now you might have realised that how repetive and boring it gets to keep copying same block of code to assign grade to each student separately. How can we make our codes more organized, and reusable? How can be get rid of verbosity of cut, copy, paste? ***Functions*** in Python will help us answer these questions.

Another question: Have we used any function before without knowing? Answer is yes! We used print() function so many times throughout this book. print() is an example of in-built function.

Can we define our own functions? Absolutely yes: That is what we will learn in this section.

4.2.1 Defining functions

Functions block begin with the keyword ***def*** followed by ***functions_name*** followed by ***parenthesis*** in which arguments or inputs are specified. In grading example, let us name our function grading with inputs or arguments as score. Hence, function will look like:

In [10]:

```
def grading(score):
    print("we will calculate grades")
```

Now, to call function, simply use the word grading and specify score in parenthesis:

In [11]:

```
grading(65)
```

Let us make our function more meaningful by adding actual grade definitions. Adding comments while defining functions always help. In our grading example, we want that user must specify scores as list of floats or integers.

Very very important:

- Notice that all the variable names that we use to write the functions are local in scope. For instance, variable student_grades is recognized only within the function and has no relevance outside the function.
- Python works by intendation! All the statements within function are at one level of indentation. Recall that you can intend the line by pressing tab key before it.
- Last print statement is outside the for loop but within the function. The way to identify it is indentation. print(student_grades) is at same level of indentation as for loop, implying that it is outside the for loop. However, print(student_grades) is at first intendation level with respect to **def**
- By now, you may have noticed that use of spacebar is prohibited while declaring variables. Same rule applies to the function names.
- return command returns the output of function and is usually specified at the end of function definiton.

By running the ***def*** block, we have just defined the function. No output is expected as we have not executed the function yet.

In [12]:

```
# Function to compute grades.
# User must specify scores as list
def grading(scores):
    student_grades = []
    for i,score in enumerate(scores):
        if score > 90:
            student_grades.append([i,"Grade is A"])
        elif score >81 and score <=90:
            student_grades.append([i,"Grade is B"])
        elif score >71 and score <=80:
            student_grades.append([i,"Grade is B-"])
        elif score >61 and score <=70:
            student_grades.append([i,"Grade is C"])
        elif score >51 and score <=60:
            student_grades.append([i,"Grade is C-"])
        elif score >41 and score <=50:
            student_grades.append([i,"Grade is D"])
        elif score <40:
            student_grades.append([i,"Grade is F"])
        else:
            student_grades.append([i,"student absent"])
    return student_grades
```

4.2.2 Executing functions

Let us create a variable named scores_highschool and pass it as an argument to our grading function.

In [13]:

```
scores_highschool = [65.0, 98.5, 46.7, 78.5, 83.5, 56.5, 92.5, 78, 63, 
#Executing the function: Grades are expected.
a = grading(scores_highschool)
print(a)
```

```
[[0, 'Grade is C'], [1, 'Grade is A'], [2, 'Grade is D'],
[3, 'Grade is B-'], [4, 'Grade is B'], [5, 'Grade is C-'],
[6, 'Grade is A'], [7, 'Grade is B-'], [8, 'Grade is C'],
[9, 'Grade is B']]
```

Now, we can reuse grading function as many times as we want without copying, pasting, again and again!

4.3 Conclusions and next steps

Congratulations for making this far into basics of Python. Now that we have enough toy examples (thanks to anonymous dealer and anonymous grader!), it is right time to use this knowledge to perform analysis on real data. In next chapter, we will discuss basics of Dataframe, Pandas, and plotting figures in Python.

Exercise

Consider the data of three Largest cities (by population). These cities are Shanghai, Beijing, and Delhi. Population of these cities are 24.2 Million, 21.5 Million, and 16.78 Million, respectively. Write a function to create dictionary with following information: Name of the city (as string) , population (as float), and location (as tuple). Your function should return the dictionary and text "Megapolis found!" if population of city is greater than 20 Million. If population is less than 20 Million, return the text "Not a Megapolis!"

Input arguments of your function should be: city_name (string), population [in Million] (as float), latitude (as float), longitude (as float).

Output argument should be list containing following two elements :

- a dictionary with following keys: name, population, location;
-

CITY	LATITUDE	LONGITUDE
Shanghai	31.23	121.47
Beijing	39.904	116.40
Delhi	28.7	77.1

Solution

Let us define the function

In [14]:

```
city_dict(name, population, lat, lon):    #Defining function
req_dict = {}                             #Creating empty dictionary
req_dict['name'] = name
req_dict['population'] = population
req_dict['location'] = (lat,lon)

if population > 20:
    a = ("Megapolis found!")
else:
    a = ("Not a Megapolis")
return [req_dict,a]           #Note that function can return output as
```

Let us run the function

In [15]:

```
shanghai = city_dict("Shanghai",  population = 24.2, lat = 31.23, lon =
```

Let us check what the function returned

In [16]:

```
print(shanghai)
```

```
[{'name': 'Shanghai', 'population': 24.2, 'location': (31.
23, 121.47)}, 'Megapolis found!']
```

Now, You can produce the outputs for Beijing and Delhi in similar fashion!