Infrastructure as Code- Walkthrough

Python Loops and Statements

Contents

Last Edited: 29 September 2022

A **walkthrough** is intended to bring you through a technical exercise. A walkthrough shows you how I completed a task in a particular context, on specific systems, at a point in time. The document is a good basic guide, but you should always confirm that the information is current and if a more recent best practice exists, you should make yourself aware of it.

Introduction

This walkthrough is provided to demonstrate how to get started coding loops and conditional statements in Python. The general name for this topic is control flow. In the earliest days of computing, one of the first things we realized, is that we would need to execute different branches of code, depending on Boolean conditions. If it is raining, switch on the windscreen wipers!

We might use the console or Notepad++ to write a quick script, but for deliberate work, we are always going to use a proper editor. For writing commercial code, I use PyCharm. However, for these sessions, we are going to move to Visual Studio Code.

During this walkthrough, I am going to introduce an important concept in Python, colons, and indentation. Look out for this, and make sure you fully understand it. You really cannot proceed without this basic knowledge.

Deliverables

You will be required to keep a list of commands with notes as to why you are using them as well as results of any testing you have done. You may be asked to present these notes as evidence of work completed or as part of a lab book. Check with your lecturer!

Prerequisites

1. I will carry out my exercises using a Windows 11 desktop. You can carry out this work on almost any modern platform.
2. You should have already installed Python as per my notes.
3. You should have already installed Visual Studio Code (VSC) as per my notes.
4. You have created a directory to keep example files in and you are ready to code.
    a. This could be on your OneDrive, in these examples, I am using **OneDrive\Python\Exercises_04**

Flow Control

To control the flow of execution in a program, we normally have a condition associated with a block of code. There are three main types of conditional statements we will deal with.

If, Elif and Else

The first conditional statement we will consider is the if statement. This is also one of the oldest statements used in high level languages.

1. I have opened VSC and created a new directory **C:\Python\Exercise_03**
2. I have created a file, **my_if.py**
3. Over the next few minutes, I am going to build a code template I can use in any project.
4. In Python, control flow is determined by colons (**:**) and indentation. Indentation must be perfect, or you will generate errors when you try to run the program.
5. The syntax is:

```
if condition1:
   # Execute code
elif condition2:
   # Execute code
else:
   # Execute other code
```

If a condition1 is met, code is executed until the end of the if indented block. Indentation can be four spaces, or a tab, but you must use the same indentation throughout.

If the first condition is not met, condition2 is checked, if it is, code is executed until the end of the elif indented block. You may have many elif conditions, but once one is met, its code block will be executed and the programme will move beyond the if, elif, else code block.

If no condition is met, the else code block is executed.

I pulled an example from a program I wrote c. 2015 to decode some Smart Grid data. Note that I put enough comments in that even now, I can figure out what I did and why.

```
# Now test the second byte to identify the frame type
if SecondByte == 0x00:                  # AA0x Data frame
   self.FrameType = 'Data'
elif SecondByte == 0x10:                # AA1x Header frame
   self.FrameType = 'Header'
elif SecondByte == 0x20:                # AA2x Configuration Frame1
   self.FrameType = 'ConfigurationFrame1'
else:                                   # Corrupt data?
   self.FrameType = 'Unknown'
```

Take note of the indentation, the location of colons and the conditions.

Exercise

I saved the following code in **my_if.py**

```
a = True
b = True
c = True

if a:
   print("a was true")
elif b:
   print("b was true")
elif c:
   print("c was true")
else:
   print("None of our boolean variables were true")
```

Run this code, then:

1. Set a = False and run, examine the output
2. Set b = False and run, examine the output
3. Set c = False and run, examine the output

Notice the limitations, once the first test is met, the if/elif/else code block terminates.

I wrote the following code snippet to check more complicated Boolean conditions. Play around with the numbers and conditions so you can understand what I did.

```
if (3<2) and (5<9):
   print("Yup!")
else:
   print("Nope!")
```

Loops

As a general guide, loops keep repeating themselves until the condition to stop the loop is met.

For loops

There is a technical term called iteration. If I have a shopping list, I can walk down the aisles of the supermarket, iterating through the shopping list, until each of the items is in my basket. We can use loops to iterate through items (lists, sequences) and execute a block of code utilizing each item. The loop finishes when a condition is met.

1. I started by creating a new file called **my_for.py**
2. Never use an actual Python keyword as a file name!
3. I entered the following code.
    a. I created a list called **iterable_variable** and populated it.
    b. The next line means, for every **item** in **iterable_variable**, execute the indented code.

```
iterable_variable = [1,2,3,4,5,6]

for item in Iterable_variable:
    # For each item, execute this code block
    print(item)
```

By default, **print** appends a **\n** character, so each item appears on its own line.

```
iterable_variable = [1,2,3,4,5,6]

for item in iterable_variable:
    if item %2 != 0:
        print(item)
```

How about I combine some things we have learned so far? Examine the **if** statement and see if you can understand what it does. Can you make this code print out even number only?

How about I try to add each number in the list.

1. First, I need to define a variable to hold the result.
2. Then I can iterate through the list, adding for each item in iterable_variable
3. Finally, I **print** the total out, at the same ident level as the **for** statement, so it will only run after the **for** loop has completed.
4. What happens if you ident the print statement? Why?

```
iterable_variable = [1,2,3,4,5,6]
total = 0

for item in iterable_variable:
    total = total + item

print(total)
```

Indentation is rather important!

We can use any legal variable name and we do not have to create a variable to iterate over. Try this code below using your own name.

```
# Define a string to iterate over
for this_letter in "John ORaw":
    # Specify which letter to test for
    if this_letter == "J":
        # Found the test letter
        print(f"Woo hoo, found a {this_letter}!")
        # Exit the current loop
        break
    else:
        # Didn't find the test letter
        print(f"Aww man, I didn't want a {this_letter}!")
```

Now change the test letter to a lower case “**j**” and rerun.

Note how I used the keyword **break** above. We have some special keywords for all our Python loops.

break exists from the current loop and ends it immediately

continue goes to the top of the current loop, skipping to the next iteration

pass does absolutely nothing, I use it as a placeholder when I am writing code, and I will do the code block later.

We can end an entire program using **sys.exit()**, but we will talk about that later.

Try the code below, make sure you understand what is happening.

```
my_list = [1,2,3,0]

for my_number in my_list:
    if my_number == 1:
        pass
    if my_number == 2:
        continue
    if my_number == 3:
        print(f"Found the number {my_number}")
    if my_number == 0:
        break
```

Note that we can iterate through any sequence, for example a tuple. We can also nest tuples in lists, etc. Try the code below.

```
list_of_tuples = [(1,2), (3,4), ("A", "B")]
for item in list_of_tuples:
    print(item)
```

How about this code? Read a little about tuple unpacking to see what the term means.

```
# Tuple unpacking
list_of_tuples = [(1,2), (3,4), ("A", "B")]
for a,b in list_of_tuples:
    print(a)
    print(b)
```

Sometimes in my networking code, I get a tuple returned with multiple properties, but always in the same order. I can just extract the property I want based on its order.

When I started programming using languages like C and Basic, we used for loops very differently, with start, stop and step parameters. In Python, the **range()** function returns a range object (we will look at objects in a later session) which is like a sequence. In Python, I can use a **for** loop with a **range(start, stop, step)** to emulate this.

Try the code below.

```
for index in range(1, 100, 5):
    print(index)
```

This can be a useful way to step through data with fixed fields, or with information in fixed positions.

Exercise

1. Create a dictionary as shown, do not worry about what it means!

   scan = {"192.168.3.10": "80", "192.168.3.11": "443", "192.168.3.55": "22"}

2. Iterate through this dictionary as you have been shown. What is the result? What are you missing?
3. Instead of iterating through **scan**, try iterating through **scan.items()**, What is the difference?
4. Finally, we can address each item in the tuple. Here is an extract from some real code written for a network security scan. Try it!

```
scan = {"192.168.3.10": "80", "192.168.3.11": "443", "192.168.3.55": "22"}
for ipv4, port in scan.items():
    print(f"Found a service on {ipv4} at {port}")
```

While loops

1. I started by creating a new file called **my_while.py**
2. While loops will continue to execute a block of code so long as the condition is true.

The syntax is:

```
while condition:
   # Execute code block
else:
   # Execute something else
```

3. I give a very simple example below. Run this and understand the result.

```
x = 0
while x < 10:
   print(f"X is = {x}")
   x = x + 1
else:
   print(f"As x is now = {x}, we are all finished")
```

4. When we want to increment a variable, we used: **x = x + 1**
5. In Python, it is more common to do this using: **x += 1**
6. If you make a mistake in your code and create a never-ending loop, you can exit it by pressing **[ctrl][c]**.
7. Sometimes I want a program to run forever. I can do this by enclosing the code in a never-ending while block, using **while True:** or **while 1:**

```
while 1:
   pass
```

List Comprehensions

In previous notes we looked at data structures such as lists. Sometimes you may find yourself using a loop to populate a list. For example

```
my_list = []
my_string = "Morning Folks!"
for letter in my_string:
   my_list.append(letter)
print(my_list)
```

Give this a go, the result I got was as expected **['M', 'o', 'r', 'n', 'i', 'n', 'g', ' ', 'F', 'o', 'l', 'k', 's', '!']**

In a list comprehension, we collapse the block of code to

```
my_string = "Morning Folks!"
my_list = [letter for letter in my_string]
print(my_list)
```

And we get the same result! The comprehension takes an element, for an element, from a string or other iterable object.

Try

```
my_list = [number for number in range(0,20)]
print(my_list)
```

We can perform operations on variables, for example

```
my_list = [number * 10 for number in range(0,20)]
print(my_list)
```

How about adding an if statement to further filter the output

```
my_list = [number * 10 for number in range(0,20) if number < 10]
print(my_list)
```

Final scenario. I have a list of depths in feet (!) and I want to convert them to meters.

```
conversion = 0.3048
my_depth_in_feet = [12.3, 13.8, 15.3, 12.1, 8.8]
my_depth_in_meters = [(depth * conversion) for depth in my_depth_in_feet]
print(my_depth_in_meters)
```

We can create complicated comprehensions with if, else and nested loops. I never do this! If a list comprehension makes your code more readable, use it. If it makes it hard to understand, don't!

Exercise

I have an American air conditioner system returning temperatures, but I need the temperatures in standard units. Create a list of 10 values in degrees Kelvin. Write a code block to print these values in Celsius and Fahrenheit.