Python Conditional Statements, Truthy/Falsy Logic & Functions

Introduction

Welcome to Day 3 of the Python Quick Pass! Today, we'll explore conditional statements, which ties with truthy & falsy logic, and dive into the world of functions. Loops allow us to perform repetitive tasks, and functions help us structure our code and make it more organized and reusable. Let's get started!

Table of Contents

Boolean Values and Expressions

- Python: Boolean Values
- Python: Truth-Value Testing
- Wikipedia: Boolean Data Type

A Boolean can only be one of two values: `True` or `False`. These values can be explicitly defined or defined as the result of an expression. Any Python object can be included in a Boolean expression. For example, if the variable `x` was assigned the value of `35`, then the expression `x > 40` (i.e. is `x` greater than `40`?) would evaluate to `False`.

Comparing and contrasting values agains others is very important in computer programming. It allows programs to execute, or not execute, a task based on a condition. You can think of Booleans as your program's way of responding to yes-or-no questions. Can you think of an instance when you would need to ask your program a question, and execute different commands based on its answer?

In our code cells, you will see words `True` and `False` always show up in special colors and bold because they are reserved words. Remember that each reserved word has a specific purpose in Python code and cannot be used for any other purpose.

```
In [ ]: type(True)

Out[ ]: bool
```

Booleans can be assigned to variables.

```
In [ ]: the_boolean = False
        the_boolean

Out[ ]: False
```

One thing to rememebr is that the `True` and `False` booleans are capitalized. `true` and `false` don't mean anything to the Python interpreter because Python is case-sensitive. In the example below, you will get a `NameError` because `false` hasn't been defined as a variable name yet (and would not be an ideal choice for a variable name in most circumstances).

```
In [ ]: type(false)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 type(false)

NameError: name 'false' is not defined
```

Comparison Operators

Booleam vales can be created with *boolean expressions*. There are many types of boolean expressions and the first type we will present are comparisons.

Comparison operators are used to form a boolean expression.

Operator	Operation
`==`	equal to
`!=`	not equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to

Check out the examples below:

```
In [ ]: 23 == 32

Out[ ]: False
```

```
In [ ]: boolean_1 = 450 < 350
        boolean_1

Out[ ]: False
```

```
In [ ]: num_1 = 8
        num_2 = 6

        num_1 > num_2
```

```
Out[ ]: True
```

Membership Checks

Membership operators are used for a membership check. We check for membership sequences or collections such as Python lists (or strings).

Operator	Operation
`in`	Checks if the left side of expression contained in the right side of expression
`not in`	Checks if the left side of expression is not contained in the right side of expression

```
In [ ]: stl_blizzard_years = [1883, 1889, 1890, 1893, 1897, 1900, 1905, 1907, 1913, 1914, 1921, 1923, 1934, 1940, 1942, 1943, 1944, 1948, 1950, 1952, 1953, 1956, 1957, 1961, 19

        2017 in stl_blizzard_years
```

```
Out[ ]: False
```

Logical Operators

Boolean expressions can also be combined to form other Boolean expressions. The `and` , `or` and `not` keywords are called logical operators and they are used to combine Boolean expressions.

Operator	Operation
`and`	Checks if two statements are *both* True
`or`	Checks if *either* of two statements are True
`not`	Negates the value of a boolean statement

The logical operators work like they do in plain language. For example, the statement "I have a black dog and I have a brown dog" is only true if I have both a black and a brown dog. However, the statement "I have a black dog or I have a brown dog" is true if I have a dog of either color.

```
In [ ]: 10 > 8 and 23 < 25
```

```
Out[ ]: True
```

```
In [ ]: 5 in [2,3,4,5] or 'W' in '123.4545W'
```

```
Out[ ]: True
```

```
In [ ]: boolean_2 = not (4 == 5)
        boolean_2
```

```
Out[ ]: True
```

While it is easy to read a logical operator like we would speak it, there is a big limitation. You need to have a *COMPLETE* boolean expression on either side of the operator for it to evaluate correctly. Consider this example:

```
In [ ]: 2014 and 2018 in stl_blizzard_years
```

```
Out[ ]: True
```

```
In [ ]: print(2014 in stl_blizzard_years)
        print(2018 in stl_blizzard_years)

False
True
```

As you can see, we, as people, would evaluate the top statement as false (if we were to scan the list of blizzard years in St. Louis and see that 2014 was NOT included), but Python evaluates the statement as `True` . We didnt get an error so this could be dangerous logic if we were making further programmatic decisions upon the outcome of this evaluation.

```
In [ ]: # Correct logical operator statement:

        2014 in stl_blizzard_years and 2018 in stl_blizzard_years
```

```
Out[ ]: False
```

If Statements

- Python: The if statement
- TutorialsPoint: Python IF...ELIF...ELSE Statements

Sometimes when we're writing code, we want to execute certain lines only under certain conditions. We hinted at this when we introduced Boolean values, saying that that's how your code responds to yes-or-no questions. Now we are going to learn the proper way to evaluate yes-or-no questions in Python.

The `if` statement is used in Python to make decisions. After the `if` keyword, we must always put a Boolean expression and then a colon (`:`). If the Boolean expression after the `if` is true, Python will execute the code indented under the `if` , then skip to the end of the `if` block. If the condition is false, Python will move on to the next statement in the block.

```
In [ ]: temperature = 12

        if temperature < 32:
            print('It is literally freezing outside! Stay inside with jammies and hot cocoa.')
        elif temperature < 55:
            print('Layer up and stay warm.')
        else:
            print('Looks to be a nice day!')

It is literally freezing outside! Stay inside with jammies and hot cocoa.
```

You will typically see logic trees constructed like the one above; starting with `if` , then `elif` (short for 'else if'), and a final catch-all, `else` , for anything not evaluated as `True` with the previous blocks. As you see in the example above, the code will only arrive to a single solution, which is what we want, but consider this code:

```
In [ ]: temperature = 15

        if temperature < 55:
            print('Layer up and stay warm.')
        elif temperature < 32:
            print('It is literally freezing outside! Stay inside with jammies and hot cocoa.')
        else:
            print('Looks to be a nice day!')

Layer up and stay warm.
```

Would this code work? It would, but not as you'd expect. This logic tree would result in the incorrect response (outside of our intent) when the script runs. We would never hit the `elif` block because the `if` block would evaluate `True` to ANY temperature below 55. It is important to utilize programmatic thinking and imagine how the code is evaluated in Python. You would want to start with the most restrictive condition to evaluate `True` higher up in your conditional logic flow.

- Note: `if` statements do not require the other conditionals to follow if not needed. Sometimes you want a conditional to evaluate a special step only if the conditional is true and still run the rest of the code afterwards.

Python Functions

Understanding functions is one of the most important skills to understand with any programming language. Functions provide the basic structure for the operation of a code base. Think about how a program like Microsoft Word works. Everything you do on the application has an associated function. Every single button click, cursor movement, highlight action, font change, etc. has code dedicated to its operation.

The utility of functions is that they are reusable. When you write a function, you are establishing *HOW* the computer takes an input and comes to an output. You are just putting that process together using the foundational toolbox you learn here and packaging that process into a neat, reusable, box. In fact, one of those big tools is the conditional logic. Conditional logic is a big part of the *HOW* of getting a result. Consider this function that was introduced in lesson 1:

```
# Function DEFINED here
def find_name(target_name, input_data):
    present = False
    name_location = input_data.find(target_name)
    if name_location != -1:
        present = True
    return present

long_string_of_data = "Jim Wallace, Bill Smith, Frank Scott, Deborah Randall, Willie Nelson, Kevin Zhang, Reah Nelson, Beth Doran, Aaron Wood, Chance Rooney, Jamal Freeman, Ron Swanson"

# Function CALLED here
find_name("Aaron Wood", long_string_of_data)
```

The inner workings of a function will usually contain the conditional logic to get to a result. Everything indented under the first line of the function is the inner workings

To properly call a function, you must supply the correct number of parameters that the function expects. if you don't (or don't define a function with *default* parameters), you will receive a `TypeError` explaining that you supplied the wrong number of arguments.

In [ ]:
```
# Function DEFINED here
def find_name(target_name, input_data):
    present = False
    name_location = input_data.find(target_name)
    if name_location != -1:
        present = True
    return present

# Function CALLED here
find_name("Aaron Wood")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 10
      7     return present
      9 # Function CALLED here
---> 10 find_name("Aaron Wood")

TypeError: find_name() missing 1 required positional argument: 'input_data'
```

The `return` statement is used to specify the value that a function should return. It's optional, and a function can return nothing (i.e., `None`) if `return` is omitted.

In [ ]:
```
def multiply(a, b):
    return a * b

result = multiply(4, 3)
print(result)  # Output: 12
```

```
12
```

In this example, the `multiply` function returns the product of `a` and `b`.

Function Scope

Like mentioned before, you can think of a function as a little mini-program that takes something in, does something with it, and gives a result(in most cases). Once you have a function that provides the output you want, you do not have to think about what happens inside of the function. In fact, what happens inside of a function is limited to staying inside.

In [ ]:
```
my_number = 5
my_other_number = 6
my_operation = '-'

unused_variable = 1000

# function DEFINED here
def number_processor(num_1, num_2, operation):
    unused_variable = 999
    print("unused variable inside the function: ", unused_variable)
    num_1 = 9
    num_2 = 25
    if operation == "+":
        result = num_1 + num_2
    elif operation == "-":
        result = num_1 - num_2
    elif operation == '/':
        result = num_1 / num_2
    else:
        result = num_1 * num_2
    return result

print(f"This is the result of number_processor on {my_number} {my_operation} {my_other_number}: ")
# Function CALLED here and result printed out
print(number_processor(my_number, my_other_number, my_operation), "\n")

print(my_number)
print(my_other_number)
print("Unused variable after function has run: ", unused_variable)
```

```
This is the result of number_processor on 5 - 6:
unused variable inside the function:  999
-16

5
6
Unused variable after function has run:  1000
```

Now, this is a useless function as the input parameters are being completely redefined to static values, but it is illustrating the fact that you can change the variable *INSIDE* the function and it is unchanged outside.

Conversely, anything defined *inside* the function is only accessible inside.

In [ ]:
```
def my_function():
    x = 10
    print(x)

my_function()
print(x)  # This will raise an error
```

```
10
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[20], line 6
      3     print(x)
      5 my_function()
----> 6 print(x)  # This will raise an error

NameError: name 'x' is not defined
```

In this case, `x` is a local variable, and attempting to print it outside the function scope will result in an error.

When to Use `print` Statements

While functions are designed to encapsulate logic and produce results, you might need to use `print` statements within functions for debugging purposes or to provide visual feedback. It's common to print intermediate results or diagnostic information while developing and testing code.

In [ ]:
```
def complex_calculation(x, y):
    product = x * y
    print(f"Intermediate result: {product}")
    product = abs(product)
    if product < 55:
        result = "Does not meet threshold."
    else:
        result = "Meets Threshold"
    return result

complex_calculation(-2, 30)
```

```
Intermediate result: -60
```

Out[ ]:
```
'Meets Threshold'
```

In this example, the `print` statement helps monitor the progress of the `complex_calculation` function.

Exercise

Write your own function that determines what letter grade prints out when called with a numeric grade as a parameter.

i.e.

`letter_grade(75)` returns "C"
`letter_grade(96)` returns "A"

- Remember to define the function first, then call it

In [ ]:
```
### Your code goes here
```

Functions in Python are powerful tools for organizing and reusing code. Whether you need to define functions with multiple parameters, nest functions, manage variable scope, or control the output using `return` and `print`, understanding how to use functions effectively is crucial for building complex and maintainable programs.

4. Conclusion

Today, we've delved into loops, an essential tool for performing repetitive tasks, and functions, which help us organize and reuse code. You've learned about different types of loops and how to use them with various data structures. Additionally, you've seen how to create and call functions in Python.

Keep practicing loops and functions as they are fundamental building blocks for creating more complex Python programs. We look forward to seeing you for Day 4, where we'll explore more advanced Python concepts.

Assignments

- Truthy and Falsy
- Conditional Flow