01-Comparison Operators File IO and Chained Comparison Operators

October 29, 2020

Coursework delivered by: Alison Mukoma

Copyright: Evelyn Hone College cc DevsBranch.

1 Comparison Operators

In this lecture we will be learning about Comparison Operators in Python. These operators will allow us to compare variables and output a Boolean value (True or False).

If you have any sort of background in Math, these operators should be very straight forward.

First we'll present a table of the comparison operators and then work through some examples:

Table of Comparison Operators

In the table below, a=3 and b=4.

Operator

Description

Example

==

If the values of two operands are equal, then the condition becomes true.

(a == b) is not true.

!=

If values of two operands are not equal, then condition becomes true.

(a != b) is true

>

If the value of left operand is greater than the value of right operand, then condition becomes true.

(a > b) is not true.

<

If the value of left operand is less than the value of right operand, then condition becomes true.

(a < b) is true.

>=

If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.

(a >= b) is not true.

<=

If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

(a <= b) is true.

Let's now work through quick examples of each of these.

Equal

```
[1]: 2 == 2
```

```
[1]: True
```

```
[2]: 1 == 0
```

```
[2]: False
```

Note that == is a comparison operator, while = is an assignment operator.

Not Equal

```
[3]: 2 != 1
```

```
[3]: True
```

```
[4]: 2 != 2
```

```
[4]: False
```

Greater Than

```
[5]: 2 > 1
```

```
[5]: True
```

```
[6]: 2 > 4
```

```
[6]: False
```

Less Than

```
[7]: 2 < 4
```

```
[7]: True
```

```
[8]: 2 < 1
```

```
[8]: False
```

Greater Than or Equal to

```
[9]: 2 >= 2
```

```
[9]: True
```

```
[10]: 2 >= 1
```

```
[10]: True
```

Less than or Equal to

```
[11]: 2 <= 2
```

```
[11]: True
```

```
[12]: 2 <= 4
```

```
[12]: True
```

Great! Go over each comparison operator to make sure you understand what each one is saying. But hopefully this was straightforward for you.

Next we will cover chained comparison operators below

2 Chained Comparison Operators

An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more complex test. You can use these chained comparisons as shorthand for larger Boolean Expressions.

In this lecture we will learn how to chain comparison operators and we will also introduce two other important statements in Python: **and** and **or**.

Let's look at a few examples of using chains:

```
[1]: 1 < 2 < 3
```

```
[1]: True
```

The above statement checks if 1 was less than 2 **and** if 2 was less than 3. We could have written this using an **and** statement in Python:

[2]:
```
1<2 and 2<3
```

[2]:
```
True
```

The **and** is used to make sure two checks have to be true in order for the total check to be true. Let's see another example:

[3]:
```
1 < 3 > 2
```

[3]:
```
True
```

The above checks if 3 is larger than both of the other numbers, so you could use **and** to rewrite it as:

[4]:
```
1<3 and 3>2
```

[4]:
```
True
```

It's important to note that Python is checking both instances of the comparisons. We can also use **or** to write comparisons in Python. For example:

[5]:
```
1==2 or 2<3
```

[5]:
```
True
```

Note how it was true; this is because with the **or** operator, we only need one *or* the other to be true. Let's see one more example to drive this home:

[6]:
```
1==1 or 100==1
```

[6]:
```
True
```

Great! For an overview of this quick lesson: You should have a comfortable understanding of using **and** and **or** statements as well as reading chained comparison code.

Go ahead and go to the quiz for this section to check your understanding!

3 File I/O

Reading and writing files.

3.1 Working with paths

[ ]:
```
import os

current_file = os.path.realpath('file_io.ipynb')
print('current file: {}'.format(current_file))
# Note: in .py files you can get the path of current file by __file__
```

```
current_dir = os.path.dirname(current_file)
print('current directory: {}'.format(current_dir))
# Note: in .py files you can get the dir of current file by os.path.
 ↪dirname(__file__)

data_dir = os.path.join(os.path.dirname(current_dir), 'data')
print('data directory: {}'.format(data_dir))
```

3.1.1 Checking if path exists

[ ]:
```
print('exists: {}'.format(os.path.exists(data_dir)))
print('is file: {}'.format(os.path.isfile(data_dir)))
print('is directory: {}'.format(os.path.isdir(data_dir)))
```

3.2 Reading files

[ ]:
```
file_path = os.path.join(data_dir, 'simple_file.txt')

with open(file_path, 'r') as simple_file:
    for line in simple_file:
        print(line.strip())
```

The `with` statement is for obtaining a context manager that will be used as an execution context for the commands inside the `with`. Context managers guarantee that certain operations are done when exiting the context.

In this case, the context manager guarantees that `simple_file.close()` is implicitly called when exiting the context. This is a way to make developers life easier: you don't have to remember to explicitly close the file you openened nor be worried about an exception occuring while the file is open. Unclosed file maybe a source of a resource leak. Thus, prefer using `with open()` structure always with file I/O.

To have an example, the same as above without the `with`.

[ ]:
```
file_path = os.path.join(data_dir, 'simple_file.txt')

# THIS IS NOT THE PREFERRED WAY
simple_file = open(file_path, 'r')
for line in simple_file:
    print(line.strip())
simple_file.close()  # This has to be called explicitly
```

3.3 Writing files

[ ]:
```python
new_file_path = os.path.join(data_dir, 'new_file.txt')

with open(new_file_path, 'w') as my_file:
    my_file.write('This is my first file that I wrote with Python.')
```

Now go and check that there is a new_file.txt in the data directory. After that you can delete the file by:

[ ]:
```python
if os.path.exists(new_file_path):  # make sure it's there
    os.remove(new_file_path)
```