Logic and Control Flow

Aidan Hunt, University of Washington

Learning Objectives

After this lesson, students will be able to:

- interpret logical operators in Python
- write and interpret logical expressions
- conditionally execute code using if/else

Announcements/ Check-In

- Homework 2 posted
- Homework 1 grading underway

Where we are at

- In our third and final week of Python basics.
- We've discussed basic syntax, functions, lists, and for loops.
- In your homework, practice with these topics, plus strings and dictionaries.
- This week, we'll be discussing logic and control flow.

Where we are going

- Moving beyond built-in Python
- Overview of two useful Python packages: NumPy and Pandas
- Make sure you are comfortable with all of the material from Weeks 1-3 and getting your questions answered.

Motivation - Control Flow

So far in this class we've used Python to execute fairly simple, linear scripts.

- Give computer sequence of commands.
- Each command is executed in order.

Many engineering tasks are more complicated than this!

- Controller: Control strategy changes depending on conditions/state
- Iteration: Want to perform an operation until a particular condition is met (e.g., convergence).

- This sort of decision making is important for solving engineering problems.
  - Allows our program to be more flexible (i.e., respond to different inputs).
  - To implement this, we need to a way of determining whether the information that is flowing through our program satisfies certain criteria.
  - Then we can change the flow of information based on that criteria.

The way that we determine whether a criteria is met: **logical statements**. The way that we act on that criteria: **control flow**.

Logical expressions

- To make a decision, we need a way to determine if something in our program meets a particular criteria
- **Define**: a logical expression is a statement that can be answered with "true" or "false"

For example:

In [21]:
```
x = 10 # Store a value in a variable

x > 5 # Is the value greater than 5?
```

Out[21]:
```
True
```

Here, `x > 5` is a logical expression, and since `x` is greater than `5`, the result is `True`.

Comparison operators

In the example above, `>` is a **comparison operator**: it compares `x` and `y`. There are several comparison operators in Python:

- > : Greater than
- < : Less than
- = : Equal to
- != : Not equal to
- >= : Greater than or equal to
- <= : Less than or equal to

In [22]:
```
# Greater than
x > 5
```

Out[22]:
```
True
```

In [23]:
```
# Less than
x < 5
```

Out[23]:
```
False
```

In [24]:
```
# Equal to
x == 10
```

Out[24]:
```
True
```

In [25]:

```
# Not equal to
x != 10
```

Out[25]:

```
False
```

In [26]:

```
# Greater than or equal to
x >= 10
```

Out[26]:

```
True
```

In [27]:

```
# Less than or equal to
x <= 10
```

Out[27]:

```
True
```

Logical operators

Sometimes we want to make a decision based on the results of several criteria. We can use **logical operators** to combine several expressions into one:

- `and` (True only if both input expressions evaluate to True)
- `or` (True if either input expression evaluates to True, otherwise False)
- `not` (True if the input expression is False, and False if input is True)

In [28]:

```
x = 3
y = 9
```

In [29]:

```
# and operator: full result is true if connected statements are true
x < y and y > 10
```

Out[29]:

```
False
```

In [30]:

```
# or operator: full result is true if either of the connected statements are true
x < y or y > 10
```

Out[30]:

```
True
```

In [31]:

```
# not operator: flips the result of a logical statement
not (x < y or y > 10)
```

Out[31]:

```
False
```

We can also chain comparisons together, which is equivalent to connecting them with `and`:

In [32]:

```
# Example of chaining several logical operators together
1 < 2 < 3 < 4 < 5
```

Out[32]:

```
True
```

In [33]:

```
# Equivalent statement using logical operators
1 < 2 and 2 < 3 and 3 < 4 and 4 < 5
```

Out[33]:

```
True
```

(Note the order of operations in all statements above. Individual statements between `and` s and `or` s are evaluated, then the resulting `True` s and `False` s are combined.)

The `Bool` Data Type

The results of these comparisons isn't just the word "True" or "False", it's data!

In [34]:
```
y = 4 < 8
print(y)
type(y)
```

```
True
```

Out[34]:
```
bool
```

These are "boolean" or "logical" values

- Data type: `bool`
- Can have two possible values: `True` or `False`
- Can be stored in variables just like any other data
- They can be returned by functions just like any other data
- `True` is equivalent to `1`, and `False` is equivalent to `0`.

In [35]:
```
True == 1
```

Out[35]:
```
True
```

In [36]:
```
False == 0
```

Out[36]:
```
True
```

Logical Expression Examples

Discuss with those around you what each of these logical statements is doing:

Example 1)

In [37]:
```
x = 1
z = 3

y = x == z
```

- Remember that *assignment* `=` is different from equality `==`.
- As always, expression on the right of `=` is evaluated and stored in the left-hand side variable
- The logical statement above is evaluating the comparison `x == z` and then storing the result in `y`.

In [38]:
```
print(y)  # y stores the boolean value False
```

```
False
```

Example 2

In [39]:
```
x = 11
```

```
y = 6

y == x < 10
```

Out[39]:

```
False
```

- This is an example of "chaining" comparison operators together.
- This is equivalent to `y == x and x < 10`

In [40]:

```
y == x and x < 10
```

Out[40]:

```
False
```

Example 3

In [41]:

```
x = 11
y = 6

(y == x) < 10
```

Out[41]:

```
True
```

- Just like for mathematical expressions, we can use parentheses to indicate order of operations. So, `y == x` is evaluated first, and then the result is compared to `10`.
- Remember, when comparing booleans to numbers, `True` has a value of `1` and `False` has a value of `0`.
- So `y == x` evaluates to `False`. Then, the expression `False < 10` is evaluated, which is equivalent to `0 < 10`. Thus the total result of the expression is `True`!

Other types of logical statements

Equality vs Identity

Another way to compare objects is based on *identity*:

- `is`: True if two objects have the same identity.
- `is not`: False if two objects do not have the same identity.

Consider a simple comparison between two lists with identical contents.

Question: Why does `==` return `True` but `is` returns `False`?

In [42]:

```
list1 = [1, 2, 3, 4]
list2 = [1, 2, 3, 4]
```

In [43]:

```
list1 == list2
```

Out[43]:

```
True
```

In [44]:

```
list1 is list2
```

Out[44]:

```
False
```

Answer: `==` tests *equality*, whereas `is` tests *identity*.

- `==` is asking "do these lists have the same values"?
- `is` is asking "are these lists the same object?" (i.e., are `list1` and `list2` references to the same data in memory?)

If we change our code such that `list1` and `list2` point to the same object, the `is` comparison now returns true.

In [45]:

```
list1 = [1, 2, 3, 4]
list2 = list1
list1 is list2
```

Out[45]:

```
True
```

Note that you can also test the negative (that is, that two objects do not share the same identity).

In [46]:

```
list1 is not list2
```

Out[46]:

```
False
```

Membership

We can also compare data on the basis of membership using `in` and `not in`. This is easiest to understand in the context of sequences.

In [47]:

```
exampleList = [10, 20, 30, 40, 50]
```

In [48]:

```
# Is 30 in the list?
30 in exampleList
```

Out[48]:

```
True
```

In [49]:

```
# Is the sequence [10 20 30] in the list?
[10, 20, 30] in exampleList # False since [10, 20, 30] is not an element of [10, 20, 30]
```

Out[49]:

```
False
```

In [50]:

```
# Is 55 not in the list?
55 not in exampleList
```

Out[50]:

```
True
```

This is helpful for determining whether certain characters or substrings are in a given string!

In [51]:

```
# String example
testStr = 'Quark'

# Is the str 'ark' in testStr?

'ark' in 'Quark'
```

Out[51]:

```
True
```

If Statements

We can use the results of logical statements to execute certain lines of code if certain conditions are met.

Use `if` statement!

In [52]:

```
# Quick function for showcasing if statements
def checkNumber(num):

    if num > 0:
        print(num, "is a positive number.")


checkNumber(5)
checkNumber(-5)
checkNumber(0)
```

```
5 is a positive number.
```

Key points on syntax:

```
- The `if` keyword is followed by a logical expression.
- Code contained in the if statement is indented one level.
- The logical statement following the `if` keyword determines whether the code
inside the block is executed.
```

We can test several conditions by adding `elif` (short for else-if) blocks:

In [53]:

```
# Quick function for showcasing if statements
def checkNumber(num):

    if num > 0:
        print(num, "is a positive number.")
    elif num < 0:
        print(num, "is a negative number.")
    elif num == 0:
        print(num, "is neither positive or negative: it's zero!")

checkNumber(5)
checkNumber(-5)
checkNumber(0)
```

```
5 is a positive number.
-5 is a negative number.
0 is neither positive or negative: it's zero!
```

Each `elif` condition is checked only if the previous condition was not met.

Note that the last condition is met by default:

- if a number is positive or negative, it must be 0!
- this is essentially a "catch-all" case
- Instead of specifying an expression for this last case, use `else` keyword to catch anything that doesn't meet the previous criteria

In [54]:

```
def checkNumber(num):
    if num > 0:
        print(num, "is a positive number.")
    elif num < 0:
        print(num, "is a negative number.")
    else:
        print(num, "is neither positive or negative: it's zero!")

checkNumber(5)
checkNumber(-5)
checkNumber(0)
```

```
5 is a positive number.
-5 is a negative number.
0 is neither positive or negative: it's zero!
```

Other cool things we can do with logic

`If` in List Comprehensions

- List comprehensions are shorthand for creating a list based on the contents of an existing list.
- We can combine this with conditional statements to create a list based on *certain* values of the existing list.

In [55]:

```
# Define a list
oldList = [1, 2, 3, 4, 5, 6]

# Create a new list whose values are the squares
# of the even elements of the old list
newList = [num**2 for num in oldList if num % 2 == 0]

# Print the new list
print(newList)
```

```
[4, 16, 36]
```