1.4 Representing Data II: Booleans and Strings

In the previous section, we studied how we could represent numeric data in Python using two concrete data types: `int` and `float`. In this section, we'll introduce two new data types that represent common pieces of individual data, before seeing how to combine multiple pieces of data together in the next section.

Boolean data

A **boolean** is a value from the set {True, False}. Think of a boolean value as an answer to a Yes/No question, for example, "Is this person old enough to vote?", "Is this country land-locked?", and "Is this service free?".[1]

We also saw booleans in the last section as the result of comparison operations. When we write $3 > 2$, the value of this expression isn't a number, but instead a boolean (in this case, True).

In Python, boolean data is represented using the data type `bool`. Unlike the broad range of numbers we just saw, there are only two literal values of type `bool`: `True` and `False`.

Operations on boolean data

Booleans can be combined using different kinds of operations. The three most common ones are:

- **not**: reverses the value of a boolean. "not True" is False, and "not False" is True. In symbolic logic, this operator is represented by the symbol $\neg$.
- **and**: takes two boolean values and produces True when both of the values are True, and False otherwise. For example, "True and False" is False, while "True and True" is True. In symbolic logic, this operator is represented by the symbol $\wedge$.
- **or**: takes two boolean values and produces True when at least one of the values is True, and False otherwise. For example, "True or False" is True, while "False or False" is False. In symbolic logic, this operator is represented by the symbol $\vee$.

  One note about the logical *or* operation is that it is an **inclusive or**, meaning it produces True when both of the given boolean values are True.[2]

In Python, we have three *logical operators* that correspond to the above operators. Unlike the arithmetic and comparison operators we learned about in the last section, Python's logical operators are written as common English words rather than symbols: `not`, `and`, and `or`.[3]

```
>>> not True
False
>>> True and True
True
>>> True and False
False
>>> False or True
True
>>> False or False
False
```

Just as we saw how arithmetic operator expressions can be nested within each other, we can combine logical operator expressions, and even include arithmetic comparison operators:

```
>>> True and (False or True)
True
>>> (3 == 4) or (5 > 10)  # Both (3 == 4) and (5 > 10) evaluate to False
False
```

Next week, we'll discuss these logical operators in more detail and introduce a few others.

Textual data (strings)

Our second new data type in this section is used to represent text: everything from names to chat logs to the text of of Shakespeare's *Romeo and Juliet*. We represent text using the **string** data type, which is a sequence of characters.[4] We use the word *character* and not *letter* because the set of all characters extends far beyond any one alphabet (a, b, c, etc.). Here are examples of other types of characters that you have encountered before:

- numeric digits (0, 1, 2, etc.)
- letters and glyphs from languages other than English (e.g., ق, ξ, 你)
- punctuation marks and symbols (such as the period (.) and at symbol (@)
- spaces
- emojis (e.g., 😠, 🚗, 📠)[5]

All Python code is text that we type into the computer, so how do we distinguish between text that's code and text that's data, like a person's name? In Python, we represent strings using the `str` data type.[6] A `str` literal is a sequence of characters surrounded by single-quotes (`'`).[7] For example, we could write this course's name in Python as the string literal `'Foundations of Computer Science I'`. Here are some examples of writing string literals in the Python console.

```
>>> 'Foundations of Computer Science I'
'Foundations of Computer Science I'
>>> 'I ❤️ ➡️ 👤'
'I ❤️ ➡️ 👤'
>>> ''
''
```

That last example is interesting: it is an *empty string*, a string with no characters. This is perfectly allowed in Python, and is a surprisingly common value to see when processing text data—for example, working with survey data in which a participant left a question blank.

Operations on strings

While there are many different kinds of operations we can perform on strings, we'll only look at a few common ones right now.

The first is **string equality**. Two strings are equal when they have the same characters, in the same order. In Python, we can compare strings using the `==` operator, just like we could for `int`s and `float`s.

```
>>> 'David' == 'David'
True
>>> 'hi' == 'bye'
False
```

Because uppercase and lowercase letters are represented in Python as different characters, they are *not* considered equal when comparing strings. We say that string equality is *case-sensitive*.

```
>>> 'Mario' == 'mario'
False
```

Finally, string equality requires an exact match of all characters in both strings, so there can't be any "leftover" characters:

```
>>> 'David' == 'David Liu'
False
```

This brings us to our second string operation, **substring search**. This operation takes two strings s and t, and returns whether the string s appears somewhere inside of t. If this is true, we say that s is a **substring** of t. In Python, we can write a substring search expression using a new operator called `in`.

```
>>> 'm' in 'computer'  # single character search
True
>>> 'x' in 'computer'
False
>>> 'put' in 'computer'  # multiple character search
True
>>> 'pur'  in 'computer'  # False, because characters must appear consecutively
False
>>> 'computer' in 'computer'  # every string is a substring of itself
True
>>> '' in 'computer'  # the empty string is a substring of every string
True
```

Our next string operation gives us ways of building up larger strings from smaller strings. The **string concatenation** operation takes two strings s_1 and s_2 and returns a new string consisting of the characters of s_1 followed by the characters of s_2. In Python, this is represented using the same `+` operator as for arithmetic addition.[8]

And Python supports **concatenation** using the familiar `+` operator:

```
>>> 'David' + 'Mario'
'DavidMario'
>>> 'David ' + 'Mario'  # Extra space after David
'David Mario'
>>> 'Mario' + 'David '  # Order matters!
'MarioDavid '
```

The final operation we'll cover in this section is **string indexing**. This operation takes a string s and a natural number i, and produces the i-th character of s, where the counting starts at 0.[9] This operation is denoted $s[i]$, so $s[0]$ returns the first character of s, $s[1]$ returns the second, etc. We use the same syntax with square brackets in Python as well, as in the following examples.

```
>>> 'David'[0]  # Remember, indexing starts at 0
'D'
>>> 'David'[1]
'a'
>>> 'David'[2]
'v'
>>> 'David'[3]
'i'
>>> 'David'[4]
'd'
```

References

- CSC108 videos: Type bool (Part 1, Part 2, Part 3, Part 4)
- CSC108 videos: Type str (Part 1, Part 2)
- Appendix A.2 Python Built-In Data Types Reference

[1] The word "boolean" might sound a bit strange! Even though the concepts of "True" and "False" have been around since humans have existed, the term boolean is named after the logician George Boole (1815–1864), who was an early pioneer of symbolic logic.

[2] This contrasts with *exclusive or*, which produces True when exactly one of the given values is True. The *exclusive or* operation produces False when both of the given values are True.

[3] The choice to use English words rather than symbols was made to improve the *readability* of Python code. We hope you appreciate this aspect of the language when going through the following examples!

[4] One of the English definitions of the word "string" is a sequence of items.

[5] In case you're wondering, we copied these emoji characters from https://emojipedia.org/. 💯

[6] Just as `int` is a short form of "integer", `str` is a short form of "string".

[7] Python allows string literals to be written using either single-quotes or double-qutoes (`"`). You're free to use either in your own code, but these notes and other course materials will use single-quotes to match how the Python console displays strings.

[8] So just as we saw how `==` can be used on values of different types in Python, so too can `+`. The designers of the Python programming language tried to reuse operators when possible to reduce the amount of different operators programmers would have to learn. This is great for beginners, but also means that you'll need to pay attention to what the *types* of the values in an operation are, because the same operator will do different things depending on those types!

[9] Typically we're used to counting starting at 1 ("first", "second", "third", etc.). But due to how most programming languages, including Python, are designed, it is a very strong convention to have indexing of all sequences start at 0. We'll keep reminding you of this in the first few chapters because it can be a bit strange, but eventually you will get used to it!