lab02

September 15, 2018

```
In [1]: name = "Emma Prescott"
```

1 Lab 2: Data Types

Welcome to Lab 2!

Last time, we had our first look at Python and Jupyter notebooks. So far, we've only used Python to manipulate numbers. There's a lot more to life than numbers, so Python lets us represent many other types of data in programs.

In this lab, you'll first see how to represent and manipulate another fundamental type of data: text. A piece of text is called a *string* in Python.

You'll also see how to invoke *methods*. A method is very similar to a function. It just looks a little different because it's tied to a particular piece of data (like a piece of text or a number).

Last, you'll see how to work with datasets in Python – *collections* of data, like the numbers 2 through 5 or the words "welcome", "to", and "lab".

Initialize the OK tests to get started.

```
In [2]: from client.api.notebook import Notebook
        ok = Notebook('lab02.ok')
        _ = ok.auth(inline=True)
```

Submission: This should be submitted in PDF format with Homework 2.

2 1. Review: The building blocks of Python code

The two building blocks of Python code are *expressions* and *statements*. An **expression** is a piece of code that

- is self-contained, meaning it would make sense to write it on a line by itself, and
- usually has a value.

Here are two expressions that both evaluate to 3

```
3
5 - 2
```

One important form of an expression is the **call expression**, which first names a function and then describes its arguments. The function returns some value, based on its arguments. Some important mathematical functions are

Function	Description
abs	Returns the absolute value of its argument
max	Returns the maximum of all its arguments
min	Returns the minimum of all its arguments
pow	Raises its first argument to the power of its second argument
round	Round its argument to the nearest integer

Here are two call expressions that both evaluate to 3

```
abs(2 - 5)
max(round(2.8), min(pow(2, 10), -1 * pow(2, 10)))
```

All these expressions but the first are **compound expressions**, meaning that they are actually combinations of several smaller expressions. `2 + 3` combines the expressions `2` and `3` by addition. In this case, `2` and `3` are called **subexpressions** because they're expressions that are part of a larger expression.

A **statement** is a whole line of code. Some statements are just expressions. The expressions listed above are examples.

Other statements *make something happen* rather than *having a value*. After they are run, something in the world has changed. For example, an **assignment statement** assigns a value to a name.

A good way to think about this is that we're **evaluating the right-hand side** of the equals sign and **assigning it to the left-hand side**. Here are some assignment statements:

```
height = 1.3
the_number_five = abs(-5)
absolute_height_difference = abs(height - 1.688)
```

A key idea in programming is that large, interesting things can be built by combining many simple, uninteresting things. The key to understanding a complicated piece of code is breaking it down into its simple components.

For example, a lot is going on in the last statement above, but it's really just a combination of a few things. This picture describes what's going on.

Question 1.1. In the next cell, assign the name `new_year` to the larger number among the following two numbers:

1. the absolute value of $2^5 - 2^{11} - 2^1$, and
2. $5 \times 13 \times 31 + 2$.

Try to use just one statement (one line of code).

```
In [3]: new_year = max(abs(2**5-2**11-2**1), (5*13*31+2))
        new_year
```

```
Out[3]: 2018
```

Check your work by executing the next cell.

```
In [4]: _ = ok.grade('q11')
```

```
<grademaybe.ok.OKTestsResult at 0x7fb7f023a908>
```

3 2. Text

Programming doesn't just concern numbers. Text is one of the most common types of values used in programs.

A snippet of text is represented by a **string value** in Python. The word *"string"* is a programming term for a sequence of characters. A string might contain a single character, a word, a sentence, or a whole book.

To distinguish text data from actual code, we demarcate strings by putting quotation marks around them. Single quotes (`'`) and double quotes (`"`) are both valid, but the types of opening and closing quotation marks must match. The contents can be any sequence of characters, including numbers and symbols.

We've seen strings before in `print` statements. Below, two different strings are passed as arguments to the `print` function.

```
In [5]: print("I <3", 'Data Science')
```

```
I <3 Data Science
```

Just like names can be given to numbers, names can be given to string values. The names and strings aren't required to be similar in any way. Any name can be assigned to any string.

```
In [6]: one = 'two'
        plus = '*'
        print(one, plus, one)
```

```
two * two
```

Question 2.1. Yuri Gagarin was the first person to travel through outer space. When he emerged from his capsule upon landing on Earth, he reportedly had the following conversation with a woman and girl who saw the landing:

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

The cell below contains unfinished code. Fill in the `...`s so that it prints out this conversation *exactly* as it appears above.

```
In [7]: woman_asking = "The woman asked:"
        woman_quote = '"Can it be that you have come from outer space?"'
        gagarin_reply = 'Gagarin replied:'
        gagarin_quote = '"As a matter of fact, I have!"'

        print(woman_asking, woman_quote)
        print(gagarin_reply, gagarin_quote)
```

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

```
In [8]: _ = ok.grade('q21')
```

```
<gradememaybe.ok.OKTestsResult at 0x7fb7f01e86a0>
```

3.1 2.1. String Methods

Strings can be transformed using **methods**, which are functions that involve an existing string and some other arguments. One example is the `replace` method, which replaces all instances of some part of a string with some alternative.

A method is invoked on a string by placing a `.` after the string value, then the name of the method, and finally parentheses containing the arguments. Here's a sketch, where the < and > symbols aren't part of the syntax; they just mark the boundaries of sub-expressions.

```
<expression that evaluates to a string>.<method name>(<argument>, <argument>, ...)
```

Try to predict the output of these examples, then execute them.

```
In [9]: # Replace one letter
        'Hello'.replace('o', 'a')
```

```
Out[9]: 'Hella'
```

```
In [10]: # Replace a sequence of letters, which appears twice
         'hitchhiker'.replace('hi', 'ma')
```

```
Out[10]: 'matchmaker'
```

Once a name is bound to a string value, methods can be invoked on that name as well. The name is still bound to the original string, so a new name is needed to capture the result.

```
In [11]: sharp = 'edged'
         hot = sharp.replace('ed', 'ma')
         print('sharp:', sharp)
         print('hot:', hot)
```

```
sharp: edged
hot: magma
```

You can call functions on the results of other functions. For example,

```
max(abs(-5), abs(3))
```

has value 5. Similarly, you can invoke methods on the results of other method (or function) calls.

```
In [12]: # Calling replace on the output of another call to
         # replace
         'train'.replace('t', 'ing').replace('in', 'de')
```

```
Out[12]: 'degrade'
```

Here's a picture of how Python evaluates a "chained" method call like that:

Question 2.1.1. Assign strings to the names `you` and `this` so that the final expression evaluates to a 10-letter English word with three double letters in a row.

Hint: After you guess at some values for `you` and `this`, it's helpful to see the value of the variable `the`. Try printing the value of `the` by adding a line like this:

```
print(the)
```

Hint 2: Run the tests if you're stuck. They'll often give you help.

```
In [13]: you = 'keep'
         this = 'book'
         a = 'beeper'
         the = a.replace('p', you)
         the.replace('bee', this)

Out[13]: 'bookkeeper'

In [14]: _ = ok.grade('q211')

<gradememaybe.ok.OKTestsResult at 0x7fb7f006aa58>
```

Other string methods do not take any arguments at all, because the original string is all that's needed to compute the result. In this case, parentheses are still needed, but there's nothing in between the parentheses. Here are some methods that work that way:

Method name	Value
`lower`	a lowercased version of the string
`upper`	an uppercased version of the string
`capitalize`	a version with the first letter capitalized
`title`	a version with the first letter of every word capitalized

```
In [15]: 'unIverSITy of caliFORnia'.title()

Out[15]: 'University Of California'
```

All these string methods are useful, but most programmers don't memorize their names or how to use them. In the "real world," people usually just search the internet for documentation and examples. A complete list of string methods appears in the Python language documentation. Stack Overflow has a huge database of answered questions that often demonstrate how to use these methods to achieve various ends.

3.2 2.2. Converting to and from Strings

Strings and numbers are different *types* of values, even when a string contains the digits of a number. For example, evaluating the following cell causes an error because an integer cannot be added to a string.

```
In [16]: 8 + int("8")
```

```
Out[16]: 16
```

However, there are built-in functions to convert numbers to strings and strings to numbers.

```
int:   Converts a string of digits to an integer ("int") value
float: Converts a string of digits, perhaps with a decimal point, to a decimal ("float") value
str:   Converts any value to a string
```

Try to predict what the following cell will evaluate to, then evaluate it.

```
In [17]: 8 + int("8")
```

```
Out[17]: 16
```

Suppose you're writing a program that looks for dates in a text, and you want your program to find the amount of time that elapsed between two years it has identified. It doesn't make sense to subtract two texts, but you can first convert the text containing the years into numbers.

Question 2.2.1. Finish the code below to compute the number of years that elapsed between `one_year` and `another_year`. Don't just write the numbers 1618 and 1648 (or 30); use a conversion function to turn the given text data into numbers.

```
In [18]: # Some text data:
         one_year = "1618"
         another_year = "1648"

         # Complete the next line.  Note that we can't just write:
         #   another_year - one_year
         # If you don't see why, try seeing what happens when you
         # write that here.
         difference = int(another_year) - int(one_year)
         difference
```

```
Out[18]: 30
```

```
In [19]: _ = ok.grade('q221')
```

```
<gradememaybe.ok.OKTestsResult at 0x7fb7f006a0f0>
```

3.3 2.3. Strings as function arguments

String values, like numbers, can be arguments to functions and can be returned by functions. The function `len` takes a single string as its argument and returns the number of characters in the string: its **len**-gth.

Note that it doesn't count *words*. `len("one small step for man")` is 22, not 5.

Question 2.3.1. Use `len` to find out the number of characters in the very long string in the next cell. (It's the first sentence of the English translation of the French Declaration of the Rights of Man.) The length of a string is the total number of characters in it, including things like spaces and punctuation. Assign `sentence_length` to that number.

```
In [20]: a_very_long_sentence = "The representatives of the French people, organized as a Nati
         sentence_length = len(a_very_long_sentence)
         sentence_length

Out[20]: 896

In [21]: _ = ok.grade('q231')

<gradememaybe.ok.OKTestsResult at 0x7fb7f006aeb8>
```

4 3. Importing code

What has been will be again,
what has been done will be done again;
there is nothing new under the sun.

Most programming involves work that is very similar to work that has been done before. Since writing code is time-consuming, it's good to rely on others' published code when you can. Rather than copy-pasting, Python allows us to **import** other code, creating a **module** that contains all of the names created by that code.

Python includes many useful modules that are just an `import` away. We'll look at the `math` module as a first example. The `math` module is extremely useful in computing mathematical expressions in Python.

Suppose we want to very accurately compute the area of a circle with radius 5 meters. For that, we need the constant π, which is roughly 3.14. Conveniently, the `math` module has `pi` defined for us:

```
In [22]: import math
         radius = 5
         area_of_circle = radius**2 * math.pi
         area_of_circle

Out[22]: 78.53981633974483
```

`pi` is defined inside `math`, and the way that we access names that are inside modules is by writing the module's name, then a dot, then the name of the thing we want:

```
<module name>.<name>
```

In order to use a module at all, we must first write the statement `import <module name>`. That statement creates a module object with things like `pi` in it and then assigns the name `math` to that module. Above we have done that for `math`.

Question 3.1. The module `math` also provides the name `e` for the base of the natural logarithm, which is roughly 2.71. Compute $e^{\pi} - \pi$, giving it the name `near_twenty`.

```
In [23]: near_twenty = math.e**math.pi-math.pi
         near_twenty
```