Notebook

March 11, 2019

1 Simple Data Types (draft)

In Python, the data you work with (like the things assigned to variables) have *types*, which specify the kinds of data they are and the things you can do with them.

A good way to understand this is to think about the difference between letters and numbers. While we can write both down, there are different things we can do to them. It wouldn't make sense (except in an algebra context) to multiply and divide letters; it would't make sense to talk about a capital and a lowercase number 3.

Exactly the same idea exists in programming. Numbers and letters are different things, and you can do different stuff to them.

1.1 Strings

In Python, letters are made into *strings*. Usually, we write strings in python programs by enclosing them in quotes—single or double quotes, it doesn't make a difference (but we need to be consistent within a single string, and the other kind of quote just becomes an ordinary letter). Here are some strings:

```
In [1]: "I'm a string."
```

```
Out[1]: "I'm a string."
```

```
In [2]: 'I am a string too, but I have trouble using contractions.'
```

```
Out[2]: 'I am a string too, but I have trouble using contractions.'
```

```
In [5]: "It's easy for me to use contractions, because I've been surrounded by double quotes, so
```

```
Out[5]: "It's easy for me to use contractions, because I've been surrounded by double quotes, so
```

Backslashes have special functions inside strings: they're used as *escape characters* to allow you to include special kinds of letters. For example, the special character `\n` stands for a newline.

```
In [7]: print("Dear Students: \n\nEverybody gets an A!  Congratulations! \n\nJust kidding. \n\n-
```

```
Dear Students:

Everybody gets an A!  Congratulations!
```

```
Just kidding.

-Evil Professor
```

Similarly, you can use backslashes to escape quotes in strings.

```
In [9]: quotestring = "And then she said \"sic 'em, Rover!\" but Rover failed to sic."

        print(quotestring)
```

```
And then she said "sic 'em, Rover!" but Rover failed to sic.
```

You can also create multiline strings by enclosing them in triple quotes (again, of either kind).

```
In [11]: print('''
         Hi there!

         I'm a really long string over lots of lines.

         I can go on for as long as I want.  That makes me the coolest string.  Aren't those oth

         well...

         kinda lame?
         ''')
```

```
Hi there!

I'm a really long string over lots of lines.

I can go on for as long as I want.  That makes me the coolest string.  Aren't those other string

well...

kinda lame?
```

A common feature that you might want is to combine multiple strings, or to substitute variables into strings.

The easy but kind of awkward way to combine strings is just to use the plus string to *concatenate* them. Imagine that you're doing a mail-merge function, for example. Here's how we might do it.

```
In [12]: def concatenated_mail_merge(list_of_names):
             for name in list_of_names:
                 print("Dear " + name + ":\nSend me the money you owe me or I'll sue!!")

         concatenated_mail_merge(["John", "Paul", "George", "Ringo"])
```

```
Dear John:
Send me the money you owe me or I'll sue!!
Dear Paul:
Send me the money you owe me or I'll sue!!
Dear George:
Send me the money you owe me or I'll sue!!
Dear Ringo:
Send me the money you owe me or I'll sue!!
```

But that gets very un-ergonomic. Here's a better way to do it.

```
In [14]: def cleaner_mail_merge(list_of_names):
             for name in list_of_names:
                 print("Dear {}:\nThank you for the money, now I won't sue you.".format(name))

         cleaner_mail_merge(["Peter", "Paul", "Mary"])
```

```
Dear Peter:
Thank you for the money, now I won't sue you.
Dear Paul:
Thank you for the money, now I won't sue you.
Dear Mary:
Thank you for the money, now I won't sue you.
```

`format()` is one example of a *string method*. A string method is a special kind of function that you call only on a string, which you call by putting a period and then the function call after it. Most types in Python have their own special methods (and we'll learn a bit more about why this is the case when we learn what object-oriented programming is).

Here are some more string methods, which should be self-explanatory.

```
In [15]: print("shouting".upper())
```

```
SHOUTING
```

```
In [16]: print("WHISPERING".lower())
```

```
whispering
```

```
In [17]: print("this is a long long string".replace("long", "short"))
```

```
this is a short short string
```

There are lots of others, and you should go look at the documentation to see a list of them.

Another important feature of strings is that you can treat them like a sequence of individual letters. And you can use what's known as subset notation to get at individual letters or groups.

```
In [18]: mystring = "cats are the best"
         print(mystring[0])
```

```
c
```

Everything in Python starts counting at 0, so `mystring[0]` means "get me the first letter in the string at mystring. You can also do ranges:

```
In [19]: "abcdef"[0:3]
```

```
Out[19]: 'abc'
```

Like the `range()` function we looked at before, the subset notation allows us to specify a start and an end, and then it gives us from the start through *one before* the end. So in our example above, we got the 0th, the 1st, and the 2nd letters in the string.

As you noticed, the elements of the subset are separated by colons. We can add a third *step* element, again, just like in range.

```
In [20]: mystring = "abcdefghij"
         print(mystring[3:5])
```

```
de
```

```
In [21]: print(mystring[:5])
```

```
abcde
```

```
In [22]: print(mystring[0:10:2])
```

```
acegi
```

```
In [23]: print(mystring[5:0:-1])
```

```
fedcb
```

```
In [24]: def reverse_string(mystring):
             return mystring[::-1]
```

```
In [25]: reverse_string(mystring)
```

```
Out[25]: 'jihgfedcba'
```

```
In [26]: reverse_string("cat")
```

```
Out[26]: 'tac'
```

Lots of legal data is in string form (think about the texts of cases!) so we'll be spending quite a bit of time thinking about strings through this course. This is just a taste.

1.2 Numbers (integers and floats)

There are actually two kinds of numbers in Python. The simplest are integers (or ints), or whole numbers, without a decimal. Like 1, 2, and 3. You can do what you would normally expect with them.

In earlier versions of Python, you couldn't divide two integers and get a decimal, you'd get another integer, even if they didn't divide evenly. So you'd divide 5 by 2 and get 2. This isn't a problem in Python 3, but be careful if you end up using Python 2 somehow (not a good idea)

```
In [27]: 5 / 2

Out[27]: 2.5

In [32]: 1 + 1

Out[32]: 2

In [34]: 5*2

Out[34]: 10

In [35]: 5 ** 2

Out[35]: 25

In [36]: 2 ** 3

Out[36]: 8

In [37]: -2 + 2

Out[37]: 0

In [38]: 2 + -2

Out[38]: 0
```

All numbers in Python are represented in binary "under the hood." What that means is that decimal numbers, called *floats*, aren't actually exact. They're just approximations, because decimal floats can't be exactly represented in binary. The Python Documentation has a nice explanation of the fact, and here's an example:

```
In [39]: 0.1 + 0.1 + 0.1 == 0.3

Out[39]: False

In [41]: 1.2-1.0

Out[41]: 0.19999999999999996
```

This can actually be a problem when you're doing math, especially with small numbers, and especially repeatedly, as floating point inaccuracy can compound on itself and give freakishly wrong results.

Programmers who fail to take this problem into account can kill people. Seriously: during the first Iraq war, a Patriot missile failed to intercept an Iraqi Scud because of floating point math errors, causing the death of 28 American soldiers.

The easiest fix for this is to not do anything important with floating point arithmetic.

- For some uses (like calculations with money) you can just use integers. For example, instead of doing math with dollars and cents, just use cents and multiply everything by 100.
- Most languages have *numerics libraries* to mitigate floating point problems for applications where you need accurate decimals (like stats!). In Python, we use Numpy for this, and later on in the course we'll give it some work to do. Most of the stats packages you'll ever see in Python are built on Numpy, and the implementers are (obviously) aware of the problems with floating point calculations, so they work around them.

Ints and floats can be converted to one another, though obviously going from float to int loses data.

```
In [42]: float(1)

Out[42]: 1.0

In [43]: int(1.5)

Out[43]: 1

In [45]: int(1.9)

Out[45]: 1
```

As you can see, converting a float to an int just truncates it.

You can also read numbers from strings.

```
In [46]: int("1") + 1

Out[46]: 2

In [47]: float("1.5") * 3

Out[47]: 4.5
```

And you can convert floats and ints to strings.

```
In [48]: str(1.5)

Out[48]: '1.5'
```

1.3 Other Basic Types

We've already seen the special *boolean* values `True` and `False`. We've also seen the special value `None`, which represents nothing, and usually is seen when you assign the result of a function that doesn't return anything to a variable.

As a follow-up on our conditionals lesson, this is a good time to introduce the concept of compound conditionals that evaluate to a boolean. You can sort of represent basic logic this way, using the operators `and`, `or` and `not`. They work about how you'd expect (and, just like in math, you can use parantheses to organize compound expressions and make your intention clear.)

```
In [50]: if "cat" and "dog":
             print("pet store")
```

```
pet store
```

Remember how I said that values in Python can be Truthy or Falsey? Well, it turns out that the way these boolean operators work is that they return the stuff that they work with, and then that's evaluated for truthiness or falsiness. They don't actually evaluate to a boolean (I lied to you a moment ago.) So, if we look at what `"cat" and "dog"` actually returns:

```
In [51]: "cat" and "dog"
```

```
Out[51]: 'dog'
```

What happened there is that `and` tries to return the first falsey value it sees, or the last value if it doesn't see a falsey value.

```
In [52]: 0 and "dog"
```

```
Out[52]: 0
```

```
In [53]: "dog" and 0
```

```
Out[53]: 0
```

```
In [54]: 0 and False
```

```
Out[54]: 0
```

Similarly, `or` returns the first truthey value it sees, or the last value if it doesn't see any.

```
In [55]: "cat" or "dog"
```

```
Out[55]: 'cat'
```

```
In [56]: "cat" or 0
```

```
Out[56]: 'cat'
```

```
In [57]: 0 or "cat"
```

```
Out[57]: 'cat'

In [58]: None or False

Out[58]: False
```

`not` is actually a bit more honest: it figures out whether the value it's applied to is truthy or falsey and then returns the opposite boolean.

```
In [59]: not 0

Out[59]: True

In [60]: not "cat"

Out[60]: False

In [61]: not ("cat" and 0)

Out[61]: True

In [62]: False or (not (0 or False))

Out[62]: True
```

See if you can figure out why that last expression evaluated to True. Then figure out what this will evaluate to:

`"cat" and (not ("dog" or (not False)))`

Then give it a look and see if you're right! This is looking all very LSAT-ey, isn't it?