1-1_Types

September 15, 2015

1 Basic Python Syntax and Types

1.1 Basic Python Syntax

All programming languages have variables, and python is no different. To create a variable, just name it and set it with the equals sign. One important caveat: variable names can only contain letters, numbers, and the underscore character. Let's set a variable.

```
In [1]: experiment = "current vs. voltage"

In [2]: print experiment

current vs. voltage

In [3]: voltage = 2

In [4]: current = 0.5

In [5]: print voltage, current

2 0.5
```

1.2 Types and Dynamic Typing

Like most programming languages, things in python are typed. The type refers to the type of data (or **data type**). We've already defined three different types of data in experiment, voltage, and current. The types are string, integer, and float. You can inspect the type of a variable by using the type command.

It's OK if you don't fully understand this part, and you shouldn't need to use `type` at all in your current exercises. But types are important stuff if you want to really master Python!

```
In [6]: type("current vs. voltage")

Out[6]: str

In [7]: type(2)

Out[7]: int

In [8]: type(0.5)

Out[8]: float

In [9]: type(experiment)

Out[9]: str
```

```
In [10]: type(voltage)

Out[10]: int

In [11]: type(current)

Out[11]: float
```

Python is a dynamically typed language (unlike, say, C++). If you don't know what dynamic typing means, the next stuff may seem esoteric and pedantic. Its actually important, but its importance may not be clear to you until long after this class is over.

Dynamic typing means that you don't have to declare the type of a variable when you define it; python just figures it out based on how you are setting the variable. Lets say you set a variable. Sometime later you can just change the type of data assigned to a variable and python is perfectly happy about that. Since it won't be obvious until (possibly much) later why that's important, I'll let you marinate on that idea for a second.

Here's an example of dynamic typing. What is the **data type** of voltage?

```
In [12]: type(voltage)

Out[12]: int
```

Lets assign a value of 2.7 (which is clearly a float) to voltage. What happens to the type?

```
In [13]: voltage = 2.7

In [14]: type(voltage)

Out[14]: float
```

You can even now assign a string to the variable voltage and python would be happy to comply.

```
In [15]: voltage = "2.7 volts"

In [16]: type(voltage)

Out[16]: str
```

I'll let you ruminate on the pros and cons of this construction while I change the value of voltage back to an int:

```
In [17]: voltage = 2
```

1.3 Changing types: Coersion

It is possible to coerce (a fancy and slightly menacing way to say "convert") certain types of data to other types. For example, its pretty straightforward to coerce numerical data to strings.

```
In [18]: voltageString = str(voltage)

In [19]: currentString = str(current)

In [20]: voltageString

Out[20]: '2'

In [21]: type(voltageString)

Out[21]: str
```

As you might imagine, you can go the other way in certain cases. Lets say you had numerical data in a string.

```
In [22]: resistanceString = "4.0"

In [23]: resistance = float(resistanceString)

In [24]: resistance

Out[24]: 4.0

In [25]: type(resistance)

Out[25]: float
```

What would happen if you tried to coerce resistanceString to an int? What about coercing resistance to an int? Consider the following:

```
In [26]: resistanceString = "4.0 ohms"
         int(resistanceString)

        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-26-21d2c1428e4a> in <module>()
          1 resistanceString = "4.0 ohms"
    ----> 2 int(resistanceString)


        ValueError: invalid literal for int() with base 10: '4.0 ohms'
```

Do you think you can coerce that string to a numerical type?

```
In [27]: int("4.0")

        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-27-0b5266654709> in <module>()
    ----> 1 int("4.0")


        ValueError: invalid literal for int() with base 10: '4.0'
```

1.4 GOTCHA: On Being Precise with floats and ints

Again, the following may seem esoteric and pedantic, but it is very important. So bear with me.

Let's say you had some voltage data that looks like the following:

```
0
0.5
1
1.5
2
```

Obviously, if you just assigned this data individually to a variable, you'd end up with the following types

```
0   -> int
0.5 -> float
1   -> int
1.5 -> float
2   -> int
```

But what if you wanted all of that data to be floats on its way in? You could assign the variable and then coerce it to type float:

```
In [28]: voltage = float(1)
```

But that's ugly. If you want what is otherwise an integer to be a float, just add a period at the end

```
In [29]: voltage = 1.

In [30]: type(voltage)

Out[30]: float
```

This point becomes important when we start operating on data in the next section.

1.5 Integer and Float Operations

What's the point of data if we aren't going to do something with it? Let's get computing.

```
In [31]: a = 1

In [32]: b = 2

In [33]: c = a + b

In [34]: c

Out[34]: 3

In [35]: type(a), type(b), type(c)

Out[35]: (int, int, int)
```

So we got a value of three for the sum, which also happens to be an integer. Any operation between two integers is another integer. Makes sense.

So what about the case where a is an integer and b is a float?

```
In [36]: a = 1

In [37]: b = 2.

In [38]: c = a + b

In [39]: c

Out[39]: 3.0

In [40]: type(a), type(b), type(c)

Out[40]: (int, float, float)
```

You can do multiplication on numbers as well.

```
In [41]: a = 2

In [42]: b = 3

In [43]: c = a * b

In [44]: c

Out[44]: 6

In [45]: type(a), type(b), type(c)

Out[45]: (int, int, int)
```

Also division.

```
In [46]: a = 1

In [47]: b = 2

In [48]: c = a / b

In [49]: c

Out[49]: 0
```

ZING!

This is why **data type** is important. Divding two integers returnes an integer: this operation calculates the quotient and floors the result to get the answer.

If everything was a float, the division is what you would expect. (Or, if you use Python 3 - this notebook was run in Python 2.7)

```
In [50]: a = 1.

In [51]: b = 2.

In [52]: c = a / b

In [53]: c

Out[53]: 0.5

In [54]: type(a), type(b), type(c)

Out[54]: (float, float, float)
```

This is actually "fixed" in python 3, and you can pull this change in like so. For a beginner, I'd recommend *always* putting the following statement at the beginning of your scripts where you're doing a lot of math! (It'll make more sense later)

```
In [55]: from __future__ import division
```

Now we'll get a different kind of answer from division, even if both numbers are integers

```
In [56]: 1 / 2

Out[56]: 0.5
```

But beware!

Floats are approximate! This can lead to some surprising results

```
In [57]: x = .1 * 3
         y = .3

         x == y

Out[57]: False
```

1.6 Modules

What you wrote `from __future__ import division`, you were importing a module. Python has an extensive standard library that is always included with the python interpreter. You can read about it here:

http://docs.python.org/2/library/

That's a lot! It can seem overwhelming!

But wait there's more: People write other modules that you can download and install in packages. A good scientific python package comes with a lot of packages already installed.

To read more about modules, packages, and how to install them, see the module reference sheet.