Data-Types

May 29, 2015

1 What's a variable?

Let's start with going over one of the most basic concepts in programming, using a *variable*. You've heard this term used before in math class (remember $y = mx + b$?) and our usage here is more or less the same.

A **variable** is a **symbolic name** that is associated with some **value**.

This is the most basic part of programming, because we want to have some name that we can call and will return a value. We can do this simply, just like:

```
In [2]: number = 2
```

Now we have created `number` as the **variable** and assigned `2` as its **value**. As a hint, the construction of

variable = *value*

will hold in most other programming languages too. Now at any time we can use our variable `number` again, or just look at its value. We can look at it's value by using the `print()` function in Python.

```
In [3]: print( number )
```

```
2
```

We can also use the variable in a mathematical expression too.

```
In [4]: number * 2
```

```
Out[4]: 4
```

An important thing to note though is that a variable can have its associated value **change**. So if we reassign `number` using our variable creation syntax above, we'll see that it's value changes.

```
In [5]: number = 5
        print(number)
```

```
5
```

An important part to remember about variables is that it's up to you to name them well. There are a number of different data types in Python, but there's no distinction on how you have to name them. So this means that it is up to you to give good, descriptive names to your variables.

Why is that important?

Well, the whole point of descriptive variable names is to improve readability and understanding of code by yourself and others. We typically think that we will remember everything that we do, but after a month or two of doing something else it's hard to remember any one piece of code. Good naming practices can make all the difference here.

We can even go through a quick example.

```
In [6]: number = 'Helen'
```

Here I've gone and changed the variable `number` to stand for someone's name. Now that isn't great because when most people see the word `number` they expect that the variable would contain some kind of numeric value. This means that they might want to perform some math with the variable and they would then quickly experience this.

```
In [7]: number + 2
```

```
    ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-7-e96a55dd7add> in <module>()
----> 1 number + 2


    TypeError: Can't convert 'int' object to str implicitly
```

An error!

I know that this is a pretty simplistic example, but just try to keep this point in my mind as we work through these tutorials. Now, onto the meat!

2 The basic data types

Python has eight basic data types for you to use with variables. The first four that we will cover here all allow for a variable to be a single value. These four types are:

- Integers
- Floats
- Strings
- Booleans

The other four we will cover in the next lesson. Those types allow a variable to hold multiple elements as a single value. These are great for keeping multiple related values together. These collection types are:

- Tuples
- Lists
- Sets
- Dictionaries

Now, let's start with one of the most basic data types, the integer.

3 Integers

Integers are the discrete counting numbers that we've been using since you started counting on your fingers. Even without creating any variables we can still do basic arithmetic.

All of the basic arithmetic operations are the same as if we were to write them out on paper.

```
In [8]: 2 + 2
```

```
Out[8]: 4
```

```
In [9]: 4 - 2
```

```
Out[9]: 2
```

```
In [10]: 2 * 2

Out[10]: 4
```

We can also store the result of an operation into a variable. The variable will store the evaluated answer, not the arithmetic expression.

```
In [11]: first_result = 8 / 3
         first_result

Out[11]: 2.6666666666666665
```

We see that the division operator stores the answer that we are used to, which is $2.6\bar{6}$. This behavior for the division operator is actually new in Python 3! Before in Python 2 when we would do the operation
`first_result = 8 / 3`
We would get the result:
`print( first_result ) ==> 2`
This was because it was thought that if we divide one integer by another integer, that the operation should return an integer in order to keep all the variable types the same.

To access this form of truncating division (it's called truncating division, because it just truncates all the numbers after the decimal) we actually use `//` like this:

```
In [12]: second_result = 8 // 3
         second_result

Out[12]: 2
```

What if we wanted to get the remainder, what symbol is that?

```
In [13]: 5 % 2

Out[13]: 1
```

and we see here that it even works with a decimal remainder.

```
In [14]: 4.2 % 2

Out[14]: 0.20000000000000018
```

4 Floats

And that was a perfect introduction to a float, since all a float is a number with a decimal.

```
In [15]: new_float = 4.0
         print(new_float)

4.0
```

Wait, so what if I want to change an integer to a float or vice versa???

All we have to do is cast the number using the data type name that we want to transform it to. We can see that below.

```
In [16]: int(4.8)

Out[16]: 4

In [17]: float(2)
```

```
Out[17]: 2.0
```

If you're ever confused or interested about what the type of a variable is you can always just check it with `type`

```
In [18]: type(new_float)

Out[18]: float

In [19]: type(2)

Out[19]: int
```

Now something that you should notice here is that `float` and `int` are colored green (as is `print` and `type`). That's because these are words in Python that are defined already by the language. However, Python will let you overwrite them but it really is best not to ever do that! But you can see that if you ever accidentally do it, like so

```
In [20]: int = 4
         print("What have we don to int ", int)
         int(5.0)

What have we don to int  4


        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-20-3be9f679579b> in <module>()
          1 int = 4
          2 print("What have we don to int ", int)
    ----> 3 int(5.0)


        TypeError: 'int' object is not callable
```

We'll lose the behavior of the function. However, we can get it back if we just delete the assignment that we made.

```
In [21]: del int
         int(5.0)

Out[21]: 5
```

Moving forward with arithmetic, we can make entire mathematical expressions. Just like when we first learned algebra, Python respects the order of operations when it evaluates expressions (PEMDAS - Parentheses, Exponents, Multiplication, Division, Addition, Subtraction).

If we want to use an exponent we just use the `**` symbol

```
In [22]: 2 ** 3

Out[22]: 8

In [23]: eqn1 = 2 * 3 - 2
         print( eqn1 )

4
```

```
In [24]: eqn2 = -2 + 2 * 3
         print( eqn2 )
```

```
4
```

```
In [25]: eqn3 = -2 + (2 % 3)
         print( eqn3 )
```

```
0
```

```
In [26]: eqn4 = (.3 + 5) // 2
         print(eqn4)
```

```
2.0
```

Let's try some equivalencies. These are the ones that we can use
== != ¡= ¿= ¡ ¿
The == let's us check if one side of the operator is equal to the other side.

```
In [27]: 4 == 4
```

```
Out[27]: True
```

```
In [28]: 4 == 5
```

```
Out[28]: False
```

We see here that Python evaluates the expression and if it is correct it tells us that it is `True` or if it is incorrect it tells us that it is `False`.

The ! = operator let's us see if one side does not equal the other side

```
In [29]: 4 != 2
```

```
Out[29]: True
```

```
In [30]: 4 != 4
```

```
Out[30]: False
```

And then the greater than, less than, greater than or equal to, or less than or equal to operators all work as we would expect.

```
In [31]: 4 > 2
```

```
Out[31]: True
```

```
In [32]: 4 > 4
```

```
Out[32]: False
```

```
In [33]: 4 >= 4
```

```
Out[33]: True
```

5 Booleans

Testing equivalencies is a perfect introduction to our next variable type, the Boolean. In its most basic form, a Boolean is just `True` or `False`.

With just these two variables we can implement basic logic and check for truth in a programming language. Let's say that I have just one puppy at home and his name is Frankenstein. So I will say that the variable puppy is True.

```
In [34]: puppy = True

In [35]: print(puppy)

True

In [36]: type(puppy)

Out[36]: bool
```

Here we can see that when we print `puppy` it says `True` and that the type is `bool`.

Since I only have one puppy, I'm going to say that `puppies` is `False`.

```
In [37]: puppies = False
```

In Python we could have just created both of those variables at the same time. I'll show that below, but the important thing is that the number of variables on the left-hand side should be equal to the number of variables on the right-hand side.

```
In [38]: puppy, puppies = True, False
```

and we'll see here that each of those variables has its own value.

```
In [39]: print("Do I have a puppy?", puppy)
         print("Do I have puppies?", puppies)

Do I have a puppy? True
Do I have puppies? False
```

To implement logic we have three basic operations: `and`, `not`, and `or`.

These can be used to create the most basic statements and here are how they work.

If I use the `and` operator, then Python both sides of the `and` expression need to be True for the expression to be true.

```
In [40]: True and True

Out[40]: True
```

If one side of the expression is `False`, then the whole expression will be `False`

```
In [41]: True and False

Out[41]: False
```

and as you would expect we can perform these expressions with variables (remember that I only have one puppy).

```
In [42]: puppy and puppies

Out[42]: False
```

The `not` operator expects that the following value should be `False` for the expression to be true. If the following value is `True` or exists then it will say that the expression is `False`

```
In [43]: not puppies
```

```
Out[43]: True
```

```
In [44]: not puppy
```

```
Out[44]: False
```

and we can combine that with the `and` operator to make our entire previous statement about my pets `True`

```
In [45]: puppy and not puppies
```

```
Out[45]: True
```

Finally, the `or` operator just requires that **at least one** side of the expression is `True` for the expression to be `True`

```
In [46]: puppy or puppies
```

```
Out[46]: True
```

But we still need at least one side to be `True`

```
In [47]: False or False
```

```
Out[47]: False
```

6 Strings

Finally, we have our last basic data type: Strings. Text is something that is intuitive to us as humans but dealing with it programmatically can become complicated (especially when there is a lot of it and we don't know it's structure to begin with!).

To start off easily let's just make some variables.

```
In [48]: hello = 'hello'

         print( hello )
```

```
hello
```

Now, it's important to remember that the variable name does not need to be the same as its value.

```
In [49]: falafel = 'gyro'

         print( falafel )
```

```
gyro
```

We can even use basic math operators to add strings together and make a longer string

```
In [50]: print("gyros" + " and " + "falafel")
```

```
gyros and falafel
```

We can even just multiply a string to make it longer. Can you say `gyros` seven times fast?

```
In [51]: "gyros" * 7

Out[51]: 'gyrosgyrosgyrosgyrosgyrosgyrosgyros'
```

Because Python can!

However, we can only use mathematical operations that make sense and are clear what should occur (that means additive operators). We can't divide or subtract though.

```
In [52]: "gyros"/"falafel"


    ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-52-ec2273e01fb3> in <module>()
    ----> 1 "gyros"/"falafel"


        TypeError: unsupported operand type(s) for /: 'str' and 'str'


In [53]: "gyros" - "falafel"


    ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-53-df4656ac4e0f> in <module>()
    ----> 1 "gyros" - "falafel"


        TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Now we can see more easily that we can add strings and variables that have string values together to create a longer string and set that longer string to a variable.

```
In [54]: order = hello + ', I would like a ' + falafel

         print(order)

hello, I would like a gyro
```

Hmmm, well it is correct as a sentence but we forgot to capitalize `hello`! This gets to back to the part of it is a lot easier for us to see and recognize strings after a lifetime of seeing them than it is to tell the computer sometimes. Fortunately, string variables have some built-in methods that we use on the variables to help with these situations.

```
In [55]: order.capitalize()

Out[55]: 'Hello, i would like a gyro'
```

BAM! We can just use the capitalize() method on the order variable and we will get the capitalized `hello`. An important thing to note is that while it is printing the string with `Hello`, it didn't actually change the order variable.

```
In [56]: order
```

```
Out[56]: 'hello, I would like a gyro'
```

If we wanted to change the original `order` variable to the capitalized version, we would need to set `order` equal to order when we use the `capitalize()` function.

```
In [57]: order = order.capitalize()

         order

Out[57]: 'Hello, i would like a gyro'
```

There are three other functions that perform actions like `capitalize()`, and those are:

- `lower()`, makes the entire string lowercase
- `upper()`, makes the entire string uppercase
- `title()`, capitalizes every word in a string

```
In [58]: order.title()

Out[58]: 'Hello, I Would Like A Gyro'
```

But you'll notice that I screwed up a little bit by setting `order` to its capitalized version of itself (the grammar nazis reading along have probably been going crazy all this time!). When we capitalized the string, we lost the capitalized `i` which is necessary since it's the pronoun! Python is pretty smart, but it also does exactly what we told it to do and the `capitalize()` function only capitalizes the first letter in a string.

The simplest thing to do would be to go back and recreate the order variable.

```
In [59]: order = hello.capitalize() + ', I would like a ' + falafel

         order

Out[59]: 'Hello, I would like a gyro'
```

We could do this programmatically though by using some the other built-in functions.

One way would be to `strip` away the `Hello,` at the start. Python has `strip` which strips away characters from the right side and `lstrip` which strips away characters starting from the left.

```
In [60]: order.lstrip('Helo,')

Out[60]: ' I would like a gyro'
```

Notice that I didn't need to put in `l` twice, that's because I just put in all of the individual characters I want stripped and Python goes and removes **any and all** instances of those characters until it encounters a character that I did not tell it to strip. We can test that by adding an `I` the next character that we see, but not a space which comes before it.

```
In [61]: order.lstrip('Helo,I')

Out[61]: ' I would like a gyro'
```

Same result! This is a handy way of thinking, we just want to strip away the parts we don't want until we get to what we do want.

We can also check the contents of a string using built-in methods. For example, we can make sure that all of the characters are alphabetical.

```
In [62]: hello.isalpha()

Out[62]: True
```

This is handy because we can have numbers that are a string

```
In [63]: '4'.isnumeric()
```

```
Out[63]: True
```

This is a way to test the contents of the string without knowing what's inside it. This is important because sometimes we will read in text that has numbers, but we'll want those numbers to become an integer or float so we can mathematically manipulate them.

We can convert a string of numbers into an integer just by casting it with the `int()` function.

```
In [64]: real_number = int('4')

         print( real_number )
         print( type(real_number) )
```

```
4
<class 'int'>
```

We can do the same thing for floats also.

```
In [65]: float('4.2') * 2
```

```
Out[65]: 8.4
```

However, we cannot do that with anything that has alphabetical characters.

```
In [66]: float('I would like 4.5 gyros')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-66-b7a074fdfbca> in <module>()
----> 1 float('I would like 4.5 gyros')


ValueError: could not convert string to float: 'I would like 4.5 gyros'
```

7 The more complicated parts of Strings

We already saw that we can strip out a part of a string and that we can add them together. There is a bit of an additional part to strings though, which is that we can access any individual character of a larger string. This is the part of strings that breaks the whole single element notion, strings can be cut up and individual parts can be accessed.

This is something that is unique to strings, because if we have a number we can't access a single part of it and have it be the same value. We know intuitively that `40` is not the same as just `4` so we always need the entire value.

When we access just a single element of the string, that is called **indexing**. To index a single element we just add `[ ]` after the variable name and tell it the numeric index of the element we want to access.

```
In [67]: falafel
```

```
Out[67]: 'gyro'
```

```
In [68]: falafel[1]
```

```
Out[68]: 'y'
```

Huh? I said that I wanted the first element but Python returned `y` which is the second character in the `falafel` variable.

Why is that???

In Python, like in most other programming languages, all sequences are actually **zero-indexed**. That means that the numerical index for the first element is actually `0`

```
In [69]: falafel[0]
```

```
Out[69]: 'g'
```

However, the counting after that position is normal. So if want the letter `r` that is the **third** letter in the word `gyro`, then the index will be `2`

```
In [70]: falafel[2]
```

```
Out[70]: 'r'
```

We can also access elements starting from the end of the string too, we just need to use a negative number. To get the very last letter we use the index `-1`. The end starts from `-1` because `0` always means the first entry and there is no such thing as `-0`

```
In [71]: falafel[-1]
```

```
Out[71]: 'o'
```

From the end, the counting works just the same as from the start

```
In [72]: falafel[-2]
```

```
Out[72]: 'r'
```

As you can see here, there is always more than one way to skin a cat with programming. No one way to solve the problem is more *correct* than any other way, it just comes down to what makes sense for your problem, your code, and the way that you think about it.

Something to be aware of though, is that if you try to access an element **it must exist**. That means that since `gyro` is four letters long, I cannot give it an index that is greater than `3`

```
In [73]: falafel[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-73-9c7825952384> in <module>()
----> 1 falafel[5]


IndexError: string index out of range
```

That gives us an error! So always make sure that when you access an element that the index is within the range of how long the string is.

8 Slicing a string

What if we wanted to get out more than one element from a string? We can do that too, it's called slicing!

The syntax for slicing is deceptively simple, the full syntax is:

`variable[start_index : stop_index : step]`

You'll see that all of the inputs go within the `[]` and the `:` separates each input.

The `start_index` tells python what index we want to start getting elements from.

The `stop_index` tells python what index that we want elements **up to but not including**

The `step` tells python how many steps to take between elements within the range. This means that we don't need to take every element. We could instead take **every other** element if we specified a `step` of `2`.

So if we wanted to get the `gy` from `gyros` we would do

```
In [74]: falafel

Out[74]: 'gyro'

In [75]: falafel[0 : 2 : 1]

Out[75]: 'gy'
```

Remember that the index of `2` means the third element, so we specified that we wanted every letter from the first index **up until** the third index, and we want every letter.

We could get every other letter from the first four letters like so:

```
In [76]: falafel[0 : 4 : 2]

Out[76]: 'gr'
```

However, you'll rarely see someone specify all of those inputs when they slice. If you don't give all of the inputs Python just assumes the defaults. Those are:

- `start_index` is the first index `0`
- `stop_index` is the last index `-1` (notice that this will always be the last character no matter how long the string is)
- `step` of `1`

```
In [77]: falafel[0 : 2]

Out[77]: 'gy'

In [78]: falafel[: 2]

Out[78]: 'gy'

In [79]: falafel[: 4 : 2]

Out[79]: 'gr'
```

And like you noticed from the default listings, we can mix and match positive and negative indices like so

```
In [80]: falafel[ : -1 : 2]

Out[80]: 'gr'
```

One difference from accessing a single element though, is that we can give a `stop_index` that is bigger than the length of the string. Python will just give us all of the possible characters that exist happily.

```
In [81]: falafel[: 10]
```

```
Out[81]: 'gyro'
```

However, in most situations that is poor practice and you should just not give a `stop_index` so that it returns all of the characters.

```
In [82]: falafel[:]
```

```
Out[82]: 'gyro'
```

An important note though is that the `start_index` always has to come before the `stop_index`, otherwise we will get an empty string

```
In [83]: falafel[3 : 1]
```

```
Out[83]: ''
```

That's because there is no valid sequence moving from left to right while reading the string that exists with those limits.

If we want it to return the slice but reversed, we actually control that with the `step` input and tell it that we want the reverse slice

```
In [84]: falafel[3 : 1 : -1]
```

```
Out[84]: 'or'
```

9 Exercises

Use five mathematical operators (`+ - * / **`) to produce the number `4`

```
In [84]:
```

Convert the output of one of those expressions to a `float`

```
In [84]:
```

I have a string called pet_shop that has all of the different pet varieties in a store.

```
In [85]: pet_shop = 'dog cat hedgehog fish bird'
```

Capitalize all of the different pet types in a single line

```
In [85]:
```

Print out a single `g` from `pet_shop`

```
In [85]:
```

Print out just `hedgehog`

```
In [85]:
```

Print out `gohegdeh`

```
In [85]:
```

I have two variables

```
In [86]: dogs, cats = '8', '4'
```

that tell me how many `dogs` and `cat` I have at the store. Using those two variables, calculate how many more dogs I have than cats

In [86]:

Exercises completed!

In [87]:
```
from IPython.core.display import HTML


        def css_styling():
            styles = open("../styles/custom.css", "r").read()
            return HTML(styles)
        css_styling()
```

Out[87]:
```
<IPython.core.display.HTML at 0x1038db940>
```

In []: