Previous Next

Get to Know Python's Data Types

Estimated Time: 2-3 hours

Now that you can work through algorithms, it's time to understand exactly what algorithms manipulate: data! At the ground level of any computer program, you will find data. To more accurately describe the thing that data represents, we can break data up into different types. Data *types* are fundamental concepts in any programming language, and to be fluent in Python, it's important to understand what the types are and how they interact.

Grokking Data Types

In math class, maybe you learned about integers – whole numbers such as 3 or -1. In computer science, integers are only one *type of data*. There are also floats, Booleans, characters, strings and so on. Each of these data types simply represents a different concept, which then determines how the data can interact with other data.

This may seem complicated if you haven't worked with data types before, but it's really quite simple. A good metaphor is currency: you can exchange $1 USD to €0.79 EUR, and they mean the same thing. The concept of "US dollar" and "euro" refer to the same actual value in the sense that $1 will buy you the

same amount of coffee as €0.79. But, they are written in different ways, they are used in different countries, the paper money looks different; in short, they have different *attributes*.

Compare this to computer science: the decimal `2` is the same as the binary `10`. If we are being strict about our typing we would not be able to add or subtract them because they are different data types, even though they refer to the same concept of "two". You would first have to convert them both to decimals, or both to binary numbers.

Luckily for you, Python is not so onerously strict, and broadly compatible types can be converted automatically. So, in Python you can add floats (`2.0`) and integers (`2`) and the Python interpreter will automatically figure out that you want `4.0`. Let's take a look at some of the types with which Python works.

Booleans

The Boolean data type is the most simple. (FunFact™: the "Boolean" type is named after George Boole, philosopher and mathematician. He wrote about Booleans in *The Laws of Thought* in 1854.) A Boolean data type can only ever be **True** or **False**. You can check this by opening your Python interpreter:

```
>>> t = True
>>> type(t)
<class 'bool'>
```

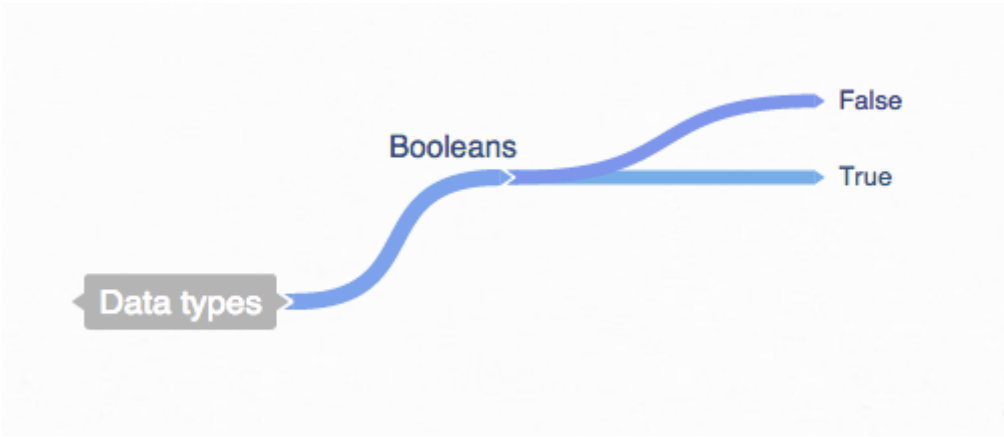
See? `True` is a keyword reserved for Booleans. You can swap `False` into the code above and you'll still get `<class 'bool'>` returned when you run `type(t)`. However, if you try changing `t` to be a number, such as `t = 42`, you'll see that `type(t)` returns something else. Try it out! Note that you can safely depend on `True` and `False` forever being what you expect:

```
>>> True = False
  File "<stdin>", line 1
SyntaxError: can't assign to keyword
```

You can use `bool()` to convert a value to a Boolean. `None`, `0`, and empty strings/lists all evaluate to `False`, everything else is `True`:

```
>>> bool(42)
True
>>> bool("Happy go lucky")
True
>>> bool(0)
False
>>> bool("")
False
```

Before things start getting complicated, let's create a chart to keep track of these. Head over to Coggle.it (or some other mind-mapping service, or pen and paper for that matter) and start a new mind map. Then, put something like the following in it:



Keeping a mind map like this isn't strictly necessary. However when you start working as a coder you will quickly find that taking good notes is very important. Mind maps just happen to be especially conducive to visualizing programming languages.

Booleans are most useful in logical operations. You've already learned the arithmetic operators (`+` , `-` , `*` , `/`). Now you get to learn the comparison and logical operators. All of the logic operators that Python offers are below:

- `==` **equal to** | `10 == 12` will evaluate to `False` and `10 == 10` will evaluate to `True`
- `!=` **not equal to** | `10 != 12` will evaluate to `True` and `10 != 10` will evaluate to `False` .
- `>` **greater than** | `10 > 12` will evaluate to `False`
- `<` **less than** | `10 < 12` will evaluate to `True`
- `>=` **greater than or equal to** | `10 >= 10` will evaluate to `True`
- `<=` **less than or equal to** | `10 <= 12` will evaluate to `True`

When you type the above examples into your Python interpreter, you'll see that they evaluate to (or return) `True` and `False` . However, you can also assign their return value to a variable, like so:

```
>>> answer = 10 >= 12
>>> answer
False
```

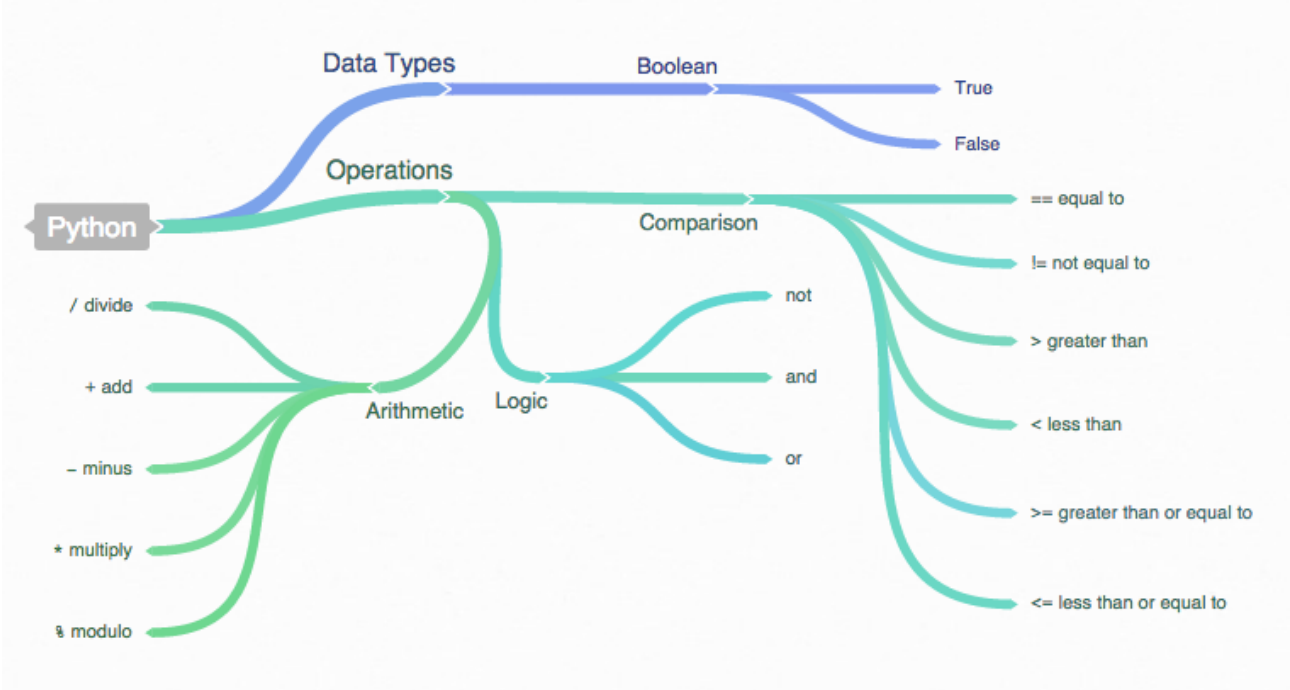
See? The value of `answer` is now set to `False` because `10` is not greater than or equal to `12` ! Very simple.

Often when you're writing logical code, you want to combine multiple operations into one statement. This is possible with the following keywords:

- **and** | `(10 == 10) and (12 != 10)` will evaluate to `True` because the first condition *and* the second condition are both true.

- **or** | (10 == 12) or (12 == 12) will evaluate to `True` because one of the conditions is true.
- **not** | `not(True)` will evaluate to `False`, and `not(10 != 10)` will evaluate to `True`, because `not` just takes the opposite of the Boolean that it's acting on.

You can now update your mind map to look something like this:



You can see how your mind map will quickly grow. From this point onward, the mind map is yours to expand as you please. When you learn something which you think is relevant be sure to add it to your map.

Strings

A string is just a series of characters. The sentence you are reading right now is stored as a string somewhere on the Thinkful servers. Strings can be created with single- or double-quotes: `'` or `"`, like so:

```
"This is a string."
'This is also a string.'
```

You can use the addition operator `+` to concatenate (join together) strings:

```
>>> "This is " + "a string."
"This is a string."
```

You can also substitute values into strings with the `format()` method:

```
>>> "{0} don't think it {1} like it is, but it {2}".format("They", "be
"They don't think it be like it is, but it do"
```

You can treat strings like a list of characters, if you want. This selects the first character in the sentence (sentences are 0-indexed):

```
>>> "Yar matey!"[0]
"Y"
```

You'll be using strings *a lot*, so make sure you spend some time getting used to them!

Numbers: Integers, floats, and others

You should already be familiar with the number types, even if you don't know the specific terms.

- Integers are whole numbers: 0, 1, 2, 3, ... 100 ... and so on. They have an unlimited size. (I dare you to test this. Hint: your computer memory will run out if you go too big.)

> **Note**
>
> Python 2 vs Python 3: In Python 2 integers only have a limited range (up to 2^{31}-1 or 2^{63}-1 depending on whether you are using a

32 or 64 bit computer. Numbers larger than this are automatically switched to a separate `Long` type.

- Floats or floating-point numbers have a decimal point in them: 1.0, 2.25, 150.2 ... and so on.
- There are other types such as `complex`, `decimal.Decimal`, and `fractions.Fraction`, which we won't be using in this course, but have important advantages in certain cases.

You *can* add (or multiply, divide, etc) different types of numbers together, but you will only get one type returned to you. This is because, technically speaking, you can't add different types, but Python detects that the types are different and converts the values to the proper data type.

Lists

A list is just a sequence of stuff, usually numbers. Lists are important because they are so flexible; you can construct entire programming languages which only use lists (e.g. Lisp). To understand the basics, take a look at how lists are constructed:

```
>>> li = []              # An empty list!
>>> other_li = [1, 2, 3] # A prefilled list!
>>> li.append(1)
>>> li.append(5)
>>> li.append(3)
>>> li
[1, 5, 3]
>>> li.pop()
3
>>> li
[1, 5]
>>> li[0]
1
```

You can see here that we created two lists, one began as empty, one prefilled. Then, we added some values to the end of the first list (`append()`). Next, we *popped* the end off the list (`pop()`). Finally, we examined the first element in the list with `li[0]` . Remember that lists (like strings) are zero-indexed!

There are many more ways to interact with a list, some of which are shown below. You don't have to memorize them, just be aware of them, and maybe add them to your mindmap for reference.

```
>>> li = [1, 2, 4, 3, 5]
# Select a range between index 1 and 3 (closed/open range, in math ter
>>> li[1:3]
[2, 4]
# Omit the beginning or end
>>> li[2:]
[4, 3, 5]
>>> li[:3]
[1, 2, 4]
# Select every second entry (i.e. step by 2)
>>> li[::2]
[1, 4, 5]
# Reverse the list
>>> li[::-1]
[5, 3, 4, 2, 1]
# Note the syntax for the above is: li[start:end:step]
# Delete the 3rd item
>>> del li[2]
>>> li
[1, 2, 3, 5]
# Search the list li to see if 1 is in it
>>> 1 in li
True
# What's the length of the list li?
>>> len(li)
4
```

An important note: Lists retain order. They don't automatically order their elements. The order in which you add stuff to your list is the order that the list will retain.

Tuples

If you examine the code above closely, you'll notice that `append()` changed the list `li` in place. That is, it actually modified the `li` object. There is no going back to the old `li`, unless you manually remove whatever you appended earlier. This is the same as interacting with objects in real life. If you break the handle off of your coffee mug, there is no going back to the old mug, unless you explicitly glue it back together.

This is called *mutability* and it is a feature of Python lists to make working with them more natural-feeling. However, sometimes you don't want mutable data structures. Sometimes you want to be able to see different versions of a data structure as you modify it. Python provides tuples for this reason.

Tuples are *immutable* lists. In fact, you can do most of the same operations on tuples that you can do on lists. However, when you add something to a tuple, you don't actually modify the tuple. Instead, a *new* tuple is created with the added value. You can't even `append()` on a tuple. You can add two tuples, but that gives you a *new* tuple:

```
>>> tup = (1, 2, 3)
>>> tup.append(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> tup + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> tup
(1, 2, 3)
>>> new_tup = tup + (4, 5, 6)
>>> new_tup
(1, 2, 3, 4, 5, 6)
>>>
```

You'll notice that `tup + (4, 5, 6)` does return a tuple, but it doesn't save the tuple as `tup`. You *could* do `tup = tup + (4, 5, 6)`, or more concisely `tup += (4, 5, 6)` and it would set `tup` to the new value of `(1, 2, 3, 4, 5, 6)`. However, this is basically mimicking mutability, and that defeats the purpose. So don't do that. If you need mutability, just use a list.

Another important fact is that tuples *maintain structure*. For this reason, tuples can also be thought of like coordinates. For example, if you want to store reference locations in a book, you would write:

```
bookmark = (42, 5) # page number, line number
```

Now, you could use `bookmark` as a key to look up the reference, for example. Or, you could keep a list of bookmarks as notes:

```
bookmark1 = (35, 5)
bookmark2 = (86, 15)
bookmark3 = (106, 11)
notes = [bookmark1, bookmark2, bookmark3]
```

Each bookmark refers to a singular location, and `notes` is a list of these locations. This is just one contrived example, but hopefully you get the idea that tuples and lists are each better suited to different roles.

Sets

Remember how we made a note that lists retain order? Well, sets don't. They are like lists that only allow unique entries. If you are familiar with mathematical sets, then you will find Python sets very familiar.

Sets are created with either `set()` or `{}`. Because they work in the same way as mathematical sets, intersection, union, and difference operations work as you would expect:

```
>>> my_set = {1, 2, 2, 3, 4, 5, 6, 7}
>>> my_set
{1, 2, 3, 4, 5, 6, 7}
>>> your_set = {6, 7, 8, 9, 10}
>>> your_set
{8, 9, 10, 6, 7}
# Set intersection - finds the common elements
>>> my_set & your_set
{6, 7}
# Set union - combines the two sets into one
>>> my_set | your_set
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
# Set difference - finds the elements in one set but not in the other
>>> my_set - your_set
{1, 2, 3, 4, 5}
# Quickly check if 1 is a member of my_set
>>> 1 in my_set
True
```

Now that you understand the major Python data structures, you can see that lists have order, tuples have structure, and sets are mathematical.

> **Note**
>
> Python 2 vs Python 3: In Python 2 the `{}` format for declaring sets isn't well supported, so it is more common to use the `set` function.

Dictionaries

The dictionary data structure simply associates one value with another, so that `k => v`, where `k` is a key and `v` is the value associated with that key.

The definitions are stored *inside* the dictionary, and you can only access them by calling on the dictionary itself first. So, let's say you make a dictionary like so:

```
>>> stooges = {"Larry": "balding, with frazzled hair",
               "Curly": "short buzz-cut",
               "Moe"  : "bowl cut"}
>>> stooges
{'Larry': 'balding, with frazzled hair', 'Moe': 'bowl cut', 'Curly': '
```

To access just one element in the dictionary, you have to call that dictionary with the name of the key that you want to look up:

```
>>> stooges['Larry']
'balding, with frazzled hair'
```

You can do fun things with dictionaries, such as get all of the keys in a dictionary, get all of the values, or check for existence:

```
>>> stooges.keys()
dict_keys(['Larry', 'Curly', 'Moe'])
>>> stooges.values()
dict_values(['balding, with frazzled hair', 'short buzz-cut', 'bowl cu
>>> "Larry" in stooges
True
```

If you lookup a non-existing key, then you get a KeyError. You can avoid this by using the `get()` method:

```
>>> stooges["Shemp"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Shemp'
>>> stooges.get("Shemp")
>>> s = stooges.get("Shemp")
# Doesn't return anything...
```

As you continue to work through the lessons you will pick up more ways to work with these core data types. But you should now have a basic

understanding of what each of the data types represent and how you can work with them.

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Completed

Previous Next