Primitive types

You can make a program for almost anything. There are programs that read books, rank movies, look for interesting numbers, or even play games. In order for programming to be so versatile, it needs to understand a few things — numbers and words, for instance. We call those things primitive types.

Python's primitive types

Here's the full list of the primitive types in Python.

- `int`: Integers are whole numbers. You can do math stuff with them. We'll talk about more details later.

```
>>> 1 + 2 * 3
7
```

- `float`: Floats are just numbers that have decimal points. Notice in the example below that dividing two integers produces a float. We'll revisit this point later.

```
>>> 8 / 2
4.0
>>> 3.14
3.14
```

- `str`: Strings represent words or sentences. They're called strings because they make up a sequence, or string, of characters. There is no difference between using single quotes or double quotes, as long as you're consistent. Looking at the last example below, note it's possible to include any character, even quote marks, within a string.

```
>>> 'robots'
'robots'
>>> "tyrannosaurs"
'tyrannosaurs'
>>> "problem'
Error
```

- `bool`: Booleans represent whether something is `True` or `False`. Those are the only two booleans in Python. We'll talk about them in a separate chapter.

```
>>> True
True
>>> 1 < 0
False
```

- `None`: This is a special value in Python that literally represents nothing. Take a look at the first example below. Notice how when you evaluate `None`, the result doesn't get displayed on the screen. This is unlike all the examples above, where evaluating `8 / 2` displayed the result `4.0`, evaluating `1 < 0` returned `False`, and so on. If you try to do things with `None`, you'll usually encounter an error.

```
>>> None
>>> 0 > None
Error
```

Make sure you understand these primitives before moving on to the next section.

Converting between types

You can also convert between primitive types. You probably won't do this very often, but sometimes it can be helpful. For example, you might have a float and want an integer, or have a string and want a boolean. Here's how to change between them.

- `int`: You can convert to an integer using the function `int`. Note that converting a float to an integer always rounds down. Also, `int(True)` gives you 1 while `int(False)` gives you 0.

```
>>> int(4.9)
4
>>> int('1')
1
```

- `float`: You can convert to a float using the function `float`. This works pretty much the same as `int`, except you get a decimal number.

- `str`: You can convert to a string using the function `str`.

```
>>> str(10)
'10'
>>> str(True)
'True'
>>> str(None)
'None'
```

- `bool`: You can convert to a boolean using the function `bool`. Everything in Python evaluates to `True`, except for `0`, `0.0`, `False`, `None`, and the empty string (`''` or `""`). Empty lists, tuples, dictionaries, and sets also evaluate to `False`, but we won't learn about those for a while.

```
>>> bool(4)
True
>>> bool(None)
False
```

I've only gone over the not-too-obvious conversions above. The rest are pretty intuitive, but don't worry because I'll make sure to go over all the weird edge cases in the practice. For now, just check that the conversions above make sense to you. Also, in case you're wondering, there's no way to convert to `None` because that wouldn't be very useful. (If you ever wanted to get `None`, you could just write `None`.)

Variable Assignment

Now that we have a handle on the different types available to us, let's talk about how we can use them.

Fundamentals of variable assignment

Lots of the time there's a particular value we want to keep track of, by giving it a name. That's where variables come in. You've probably encountered variables in math class, named obscure things like x, y, or z. In computer science we prefer more meaningful names, like this:

Now, the variable `magic_level` is assigned to the value 10. I can ask Python to evaluate an expression involving `magic_level`, and it will use 10 where it should.

```
>>> magic_level
10
>>> 2 * magic_level + 7
27
```

We could even assign another variable, using the one we just defined.

```
>>> twice_magic_level = 2 * magic_level
>>> magic_level
10
>>> twice_magic_level
20
```

What happens if `magic_level` increases to 11? We might expect `twice_magic_level` to increase to 22, but that's not actually the case.

```
>>> magic_level = 11
>>> twice_magic_level
20
```

Why? **This is very important.** Once `twice_magic_level` has been assigned to 20, it stays that way until we explicitly change it by writing something like `twice_magic_level = 22`. We could also say `twice_magic_level = 2 * magic_level`, and this would have the same effect since we just reassigned `magic_level` to be 11.

Variables are pretty useful, so make use of them. They can make your code more efficient, and easier to read. **If you ever want Python to remember a value, you should make a variable for it.**

How variable assignment works

In the examples we've seen so far, the thing on the left side of the equals sign has always been a variable name. This is for good reason.

You actually can't write `11 = magic_level`. This is because `=` doesn't represent a statement about the relationship between two values, like it does in math. Instead, `=`

change 11 to be equal to 10. That doesn't make sense.

You also couldn't write `twice_magic_level / 2 = magic_level`. That's because `twice_magic_level / 2` is a mathematical expression, not a variable. It involves a variable, but Python isn't smart enough to do the algebra and rearrange this into a legal expression like `twice_magic_level = 2 * magic_level`.

To summarize, you need the left side of the equals sign to be a variable name. It can't be a value, like 11. It can't be an expression that involves a variable name, like `twice_magic_level / 2`. The left side of the equals sign has to be a variable name, and that's it.

In order to avoid these mistakes, let's talk about how it works when Python assigns variables. Whenever you encounter an equals sign, this is what you have to do:

1. Do *not* look at the left side of the equals sign. It doesn't matter yet.
2. Evaluate the right side of the equals sign. Write the result somewhere with enough space.
3. Now look at the left side of the equals sign, and assign that variable to the value you just wrote down.

So for example, when you see `magic_level = 10`:

1. Cover the left side. For now, we just care about the `= 10` part.
2. Evaluate the right side. `10` evaluates to the integer 10.
3. Now look at the left side. We see the name `magic_level`, so we assign `magic_level` to 10.

Python always follows this 3-step procedure, whenever you want to give a new value to a variable. Don't forget it. We'll see some pretty complicated stuff in later chapters, so it will serve you well to remember this little procedure. Even when you're writing code, just keep in mind that the thing on the right gets evaluated before we bind it to the thing on the left.

Updating variables

In our earlier example we wanted to increase `magic_level` from 10 to 11. This was pretty straightforward, because we just had to write `magic_level = 11`. But what if we didn't know the current value of `magic_level`, and we wanted to increase it anyways? Referring back to the 3-step procedure we just covered, would need something like this to change `magic_level`:

We just have to fill in the blank with an expression that evaluates to 1 more than the current value of `magic_level`. So, what evaluates to 1 more than the current value of `magic_level`? Let's try `magic_level + 1`.

```
magic_level = magic_level + 1
```

It looks a bit weird. In math class you would never see something like this. But in computer science, it's okay. Why? Recall from earlier, `=` is different in math than it is in computer science. The expression above is *not* a statement claiming that `magic_level` is the same as `magic_level + 1`. It's a command, telling Python to change `magic_level` so that it equals the current value of `magic_level + 1`. To see how it works, we'll apply the 3-step procedure that we just learned:

1. Cover the left side. For now, we just care about the `= magic_level + 1` part.
2. Evaluate the right side. `magic_level + 1` evaluates to 11, if the current value of `magic_level` is 10.
3. Now we look at the left side. We see the name `magic_level`, so we assign `magic_level` to 11. We successfully increased it!

Make sure you understand why this works, before you keep reading.

Shorthand for updating variables

It's actually *really* common to do stuff like `magic_level = magic_level + 1`. In fact, it's so common that there's a special shorthand for it in Python: `magic_level += 1`. You can do this with any of the four arithmetic operators. Here are some examples:

```
>>> x = 8
>>> x += 2
>>> x
10
```

```
>>> x = 8
>>> x -= 2
>>> x
6
```

```
>>> x
16
```

```
>>> x = 8
>>> x /= 2
>>> x
4.0
```

Multiple assignment

It's even possible to assign multiple variables on one line. These next two expressions are exactly the same:

```
>>> x = 10
>>> y = 'dragon'
>>> z = True
```

```
>>> x, y, z = 10, 'dragon', True
```

But assigning multiple variables on one line is not always the same as assigning each variable on a separate line. For instance, consider this code:

```
>>> x = 5
>>> x = x + 5
>>> y = x + 10
```

In the end, `x` is 10 and `y` is 20. It's different if we combine the second and third line:

```
>>> x = 5
>>> x, y = x + 5, x + 10
```

Now `x` ends up 10 and `y` ends up 15. To understand why, we'll have to walk through it more carefully. The first line is pretty simple, we just bind `x` to 5. The next line is less simple, so let's use the 3-step procedure for variable assignment that we learned earlier.

1. Cover the left side. For now, we just care about the `= x + 5, x + 10` part.

3. Now we look at the left side. We see the names `x` and `y`, so we assign `x` to 10 and we assign `y` to 15.

Pay close attention the second step. We use the old value of `x` to calculate both `x + 5` and `x + 10`. That's because everything on the right gets evaluated before we update `x` or `y`. Whenever you come across a complicated expression, you should refer back to the 3-step procedure for variable assignment.

This syntax is also particularly convenient for swapping the values of two variables. Pretend we want to swap the values of the variables `width` and `height`. Here's how we might attempt it, without using multiple assignment:

```
>>> height = width
>>> width = height
```

But this won't work, since both variables end up with the original value of `width`. Instead, we would need to bring in a third, temporary variable:

```
>>> old_height = height
>>> height = width
>>> width = old_height
```

It's a lot simpler using multiple assignment, since both values on the right get evaluated before updating them on the left:

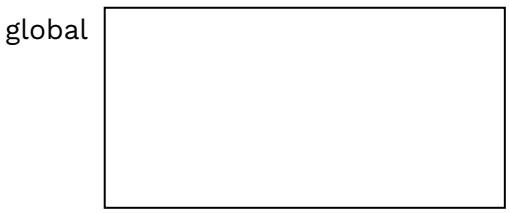
```
>>> height, width = width, height
```

Review everything above, and check that it all makes sense up to this point, before you go to the next section.

Tracking state

Up until now, we've been keeping track of all our variables mentally. But as we write more and more complicated programs, it's going to get much harder to do that. Like, literally impossible. It's time to meet your most useful tool for learning computer science. (Yes, even more useful than computers. Actually.) A pyagram, short for "Python diagram", is a way of tracing how your code runs. You use pyagrams to work your way through confusing bugs, or edge cases you don't understand. How do they

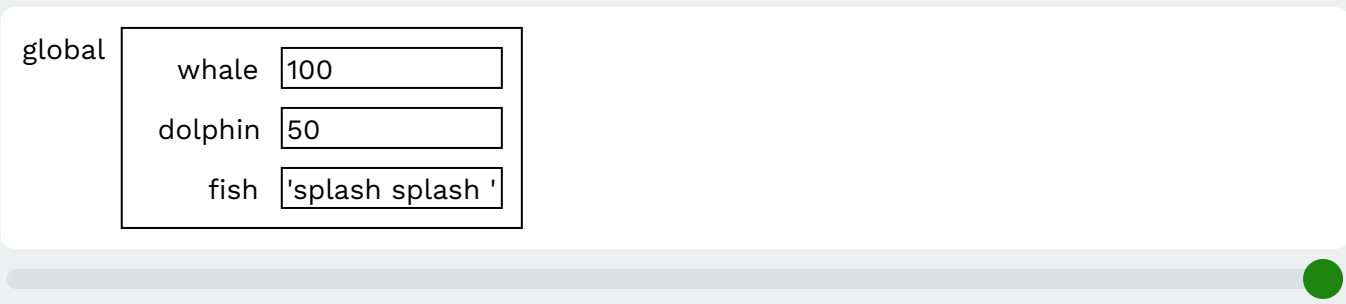
Here's an empty pyagram. The box is called a frame, which just means it's a place where code gets executed. The word "global" tells you this particular frame is the global frame. It's always the first thing you draw in a pyagram, and for this chapter we'll constrain ourselves to the global frame only.



Whenever you make a pyagram, you'll have a piece of code that you're analyzing. Let's use this code as an example:

```
>>> whale = 50 + 50
>>> dolphin = 50
>>> whale * dolphin
5000
>>> fish = int(whale / dolphin)
>>> fish *= 'splash '
```

Here's how that get's expressed in our pyagram:



We just have one more line to go: `fish *= 'splash '`. It's shorthand for saying `fish = fish * 'splash '`. Since `fish` is currently 2, it should get reassigned to the string `'splash splash '`. If that's confusing, we could also try applying the 3-step procedure from before.

1. Cover the left side. For now, we just care about the `= fish * 'splash '` part.
2. Evaluate the right side. `fish * 'splash '` evaluates to `'splash splash '`, since `fish` is 2.

Now we have finished the pyagram.

We'll be using pyagrams a lot in this class, especially as we get into more complicated material, so make sure you're comfortable with them now.

More about operators

Now we have a good understanding of variable assignment, and how to keep track of our variables using pyagrams. Let's talk about some of the operators available to us, in more detail. Before we get started, though, here are 2 rules about the math operators in Python:

- If you're doing math and any of the numbers is a float, then your output will also be a float. This is because we never want to lose accuracy by getting rid of the decimal place.
- If you're doing math and you see `True` or `False`, pretend `True` is 1 and `False` is 0. This is because computers work using binary, a language of all ones and zeros. The boolean values `True` and `False` are encoded with these numbers.

Now let's talk about Python's math operators.

1. Add: `+` We've already seen that addition works with numbers. Guess what? You can also add strings. This works like you would expect. It doesn't matter whether you use single quotes or double quotes.

```
>>> 1 + 2.0
3.0
>>> "taco" + 'cat'
'tacocat'
```

2. Times: `*` Multiplication also works between an integer and a string, like so:

```
>>> 3 * 'vroom!'
'vroom!vroom!vroom!'
>>> 3.0 * 'vroom!'
Error
```

4. Exponent: `**` You may occasionally want a number to the power of another number. Here are some interchangeable expressions:
    - English: 2 to the power of 5.
    - Python (the hard way): `2 * 2 * 2 * 2 * 2`
    - Python (the easy way): `2 ** 5`
5. Mod or remainder: `%` Mod is just another word for remainder. For example, 9 mod 4 is 1, since 9 divided by 4 is 2, with the remainder 1. Mod doesn't care about the 2, the remainder of 1.

    ```
    >>> 9 % 4
    1
    ```

6. Floor divide: `//` Floor divide is kind of like the opposite of mod. Instead of giving you the remainder, it gives you everything except the remainder. You can also think of it like normal division, but rounded down.

    ```
    >>> 9 // 4
    2
    >>> 9 / 4
    2.25
    ```

Practice: the Fibonacci sequence

Using what we've learned today, we can already do snazzy calculations. The Fibonacci sequence is a good example. It starts with the numbers 0 and 1, and every subsequent number is defined by the following recurrence relation, where F_n denotes the nth number in the sequence:

$$F_n = F_{n-1} + F_{n-2}$$

That means every Fibonacci number is the sum of the previous two. The Fibonacci sequence also grows pretty quickly. Just the 15th number in the sequence is already in the hundreds, so it's not the kind of thing you want to calculate by hand. Let's see if we can use our cool new programming skills to find it.

We'll start by defining `fib_0` and `fib_1` properly. Then we'll be able to build up from there.

Now we can proceed to calculate the remaining Fibonacci numbers, according to the formula above. We'll stop at `fib_14`, since we started counting at `fib_0`.

```
>>> fib_2 = fib_1 + fib_0
>>> fib_3 = fib_2 + fib_1
>>> fib_4 = fib_3 + fib_2
>>> fib_5 = fib_4 + fib_3
>>> fib_6 = fib_5 + fib_4
>>> fib_7 = fib_6 + fib_5
>>> fib_8 = fib_7 + fib_6
>>> fib_9 = fib_8 + fib_7
>>> fib_10 = fib_9 + fib_8
>>> fib_11 = fib_10 + fib_9
>>> fib_12 = fib_11 + fib_10
>>> fib_13 = fib_12 + fib_11
>>> fib_14 = fib_13 + fib_13
>>> fib_14
377
```

For extra practice, think about how we might make a pyagram for the code above.

We'll see very soon how to make this calculation easier and more efficient.