Intro

Today let's do something awesome. Let's start writing code. My top priority is that you understand the material. If you get a bit worn out, take a break before reading on. If something's confusing, don't skip it. We'll be going over the foundations of the rest of the class, so it's important that you really understand everything we talk about.

Primitive Types

CS is cool because you can write code about whatever you want. You can make it 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. Here's the full list.

Integer `int`	`>>> 1 + 2 * 3` `7`	Integers are whole numbers. You can do math stuff with them. We'll talk about some details later.

Float
`float`
	`>>> 8 / 2`
	`4.0`
	`>>> 3.14`
	`3.14`
		Floats are just numbers that can have decimal points. Notice that dividing two integers produces a float.

String
`str`
	`>>> "happy"`
	`'happy'`
	`>>> 'a word?'`
	`'a word?'`
	`>>> "bad'`
	`Error`
	`>>> " 'hi' "`
	`" 'hi' "`
		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. Note also that you can nest double quotes and single quotes, as in the example to the left.

Boolean
`bool`
	`>>> True`
	`True`
	`>>> 1 < 0`
	`False`
		Booleans represent whether something is `True` or `False`. Those are the only two booleans in Python. We'll talk about them in a separate note.

None	`>>> None`
	`None`
	`>>> 0 > None`
	`Error`
		None is a special value in Python that literally represents nothing. It will become important later in the course. If you try to do things with None, you'll usually encounter an error.

Make sure you have a solid understanding of these primitives before moving on to the next page.

You can also convert between primitives. You probably won't do this very often, but sometimes it can be helpful. For example, you might have a `float` and want an `int`, or have a `str` and want a `bool`. Here's how to change between them:

Integer
`int`
	`>>> int(5.9)`
	`5`
	`>>> int('1')`
	`1`
		You can convert to an integer using the function `int`.
		`int(`a float`)` rounds down.
		`int(True)` gives you `1`, and `int(False)` gives you `0`.

Float `float`	`>>> float(5)` `5.0`	You can convert to a float using the function `float`. This works pretty much the same as `int`, except you get a decimal number.

String
`str`
	`>>> str(10)`
	`'10'`
	`>>> str(True)`
	`'True'`
	`>>> str(None)`
	`'None'`
		You can convert to a string using the function `str`.
		Typically it's enough to just put quotes around your input.

Boolean
`bool`
	`>>> bool(5)`
	`True`
	`>>> bool(None)`
	`False`
		You can convert to a boolean using the function `bool`.
		Everything evaluates to `True`, except for `0`, `0.0`, `False`, `None`, and `''` or `""`. Empty lists, dictionaries, and sets also evaluate to `False`, but don't worry about those yet.

I've only gone over the not-too-obvious conversions in this chart. The rest are pretty intuitive, but don't worry because we'll make sure to go over all the weird edge cases in the practice. For now, just check that the conversions in the chart make sense to you. Also note there's no way to convert to a `None`, since that doesn't really make a lot of sense.

Variable Assignment

Now that we have a handle on the different kinds of values available to us, let's talk about how we can use them. 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 CS we prefer things to have meaningful names, like this:

```
>>> magic_level = 10
```

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

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

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

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

```
>>> magic_level = 11
>>> two_x_magic_level
20
```

Even though `magic_level` changed, `two_times_magic_level` stayed the same. Why? That has to do with how variables work in Python. 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.
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 value to what you just wrote.

Let's try it out with the example above.

`>>> magic_level = 10`

1. Cover the left	2. Evaluate the right	3. Assign the variable
`>>>` [redacted] `= 10`	`10` evaluates to `10`.	`magic_level : 10`

`>>> two_x_magic_level = 2 * magic_level`

1. Cover the left	2. Evaluate the right	3. Assign the variable
`>>>` [redacted] `=` `2 * magic_level`	`2 * magic_level` evaluates to `20`, since `magic_level` is `10`.	`magic_level : 10` `two_x_magic_level : 20`

`>>> magic_level = 11`

1. Cover the left	2. Evaluate the right	3. Assign the variable

`>>>` [redacted] `= 11`	`11` evaluates to `11`.	~~`magic_level : 10`~~
		`two_x_magic_level : 20`
		`magic_level : 11`

Notice `two_x_magic_level` is still `20`. Check this makes sense, before you continue reading.

Variable Reassignment

In the previous example, we wanted to increase `magic_level` from `10` to `11`. What if we didn't already know it was assigned to the number `10`, but we wanted to increase it by `1` anyways? We would need something like this:

```
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 CS, it's ok. Why? Because "=" doesn't represent a statement about the relationship between two values, like it does in math. Instead, "=" is our way of telling Python to change the left side to be whatever's on the right side. Let's apply the method from the previous section:

```
>>> magic_level = 10
```

1. Cover the left	2. Evaluate the right	3. Assign the variable
`>>>` [redacted] `= 10`	`10` evaluates to `10`.	`magic_level : 10`

```
>>> magic_level = magic_level + 1
```

1. Cover the left	2. Evaluate the right	3. Assign the variable
`>>>` [redacted] `=` `magic_level + 1`	`magic_level + 1` evaluates to `11`, since `magic_level` is `10`.	~~`magic_level : 10`~~ `magic_level : 11`

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

Syntax Tips

It's actually *really* common to do stuff like `magic_level = magic_level + 1`. It turns out to be especially useful for making counters that increment every time something special happens. In fact, it's so common that there's a special shorthand for it in Python. The next two expressions are the same in every way:

```
my_variable += 1
```

```
my_variable = my_variable + 1
```

You can actually do this with any of the 4 arithmetic operators. Here are some examples:

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

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

```
>>> x = 8
>>> x *= 2
>>> x
16
```

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

Another useful syntax you will come across is multiple assignment. It lets you assign (or reassign) multiple variables all on one line. The next two expressions are exactly the same:

```
x = 5
y = 3
```

```
x, y = 5, 3
```

It works for as many variables as you want, too.

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

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

Note that this syntax still obeys the 3-step process for variable assignment that we saw earlier. For example, let's walk through the code below:

```
>>> x = 5
>>> x, y = x + 5, x + 10
>>> x
10
>>> y
15
```

Notice `y` uses the "old" value of `x`, so it gets assigned to `15`, not `20`.

```
>>> x = 5
```

1. Cover the left	2. Evaluate the right	3. Assign the variable
>>> ▮ = 5	5 evaluates to 5.	x : 5

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

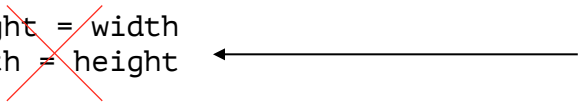
1. Cover the left	2. Evaluate the right	3. Assign the variable
>>> ▮▮▮ = x + 5, x + 10	x+5 evaluates to 10 and x+10 evaluates to 15, since x is currently assigned to 5.	~~x : 5~~ x : 10 y : 15

This is syntax particularly convenient for swapping the values of two variables. For instance, if we want to swap the values of the variables `width` and `height`, then the next two expressions are the same. Notice we need a third, temporary variable on the left, so that we don't accidentally lose the value of `height` when we reassign it.

```
temp = height
height = width
width = temp
```

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

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



This won't work, since both variables end up with the value of `width`.

Also know that if two of the variables in the expression are the same, then we go with the most recent one. For example:

```
>>> x, x, x = 1, 2, 3
>>> x
3
```

When you're assigning several things to the same value, you can also do that like so:

```
x = 5
y = 5
```

```
x = y = 5
```

This should make sense before you go on to the next section.

More On Operators

Before we get started, here are 2 quick tips. They're very important.

1. If you are doing math and any of the numbers is a float, then your output will always be a float. This is because we never want to lose accuracy by getting rid of the decimal place.
2. If you're doing math and you see `True` or `False`, pretend `True` is `1` and `False` is `0`.

We've already seen a few of the operators in Python, such as add (+), subtract (-), times (*), and divide (/). Now we will talk about them in more detail, as well as a few other operators you should know about.

1. Add : +

We've already seen that addition works with numbers. Guess what? You can also add strings. This works like you would expect. Remember, it doesn't matter whether you use single quotes or double quotes.

```
>>> 1 + 2.0 + 3 + 4
10.0
```

```
>>> "one" + 'two'
'onetwo'
```

2. Times : *

Multiplication also works between an integer and a string, like so:

```
>>> 3 * 'howdy !' * 1
'howdy !howdy !howdy !'
```

```
>>> 3.0 * 'howdy !' * 1
Error
```

3. Divide : /

Division always gives you a float, even if both the numbers you're dividing are integers. If you ever try to divide by `0`, you will get an error.

4. Exponent : **

You may occasionally want a number *to the power of* another number. Here are some interchangeable expressions:

English	Python (the hard way)	Python (the easy way)
2 to the power of 5.	`2 * 2 * 2 * 2 * 2`	`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, *only* the remainder of 1.

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

```
>>> 19 % 7
5
```

We'll see some uses of this operator in the next note.

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
>>> 18 // 3
6
>>> 20 // 3
6
```

```
>>> 9 / 4
2.25
>>> 18 / 3
6.0
>>> 20 / 3
6.666666666666667
```