Let's Learn Python!

Young Coders at PyCon 2015

(Leave this screen up before the class begins, as the kids are coming in. Use this as an opportunity to pass out name tags, if you have them, and collect any paperwork, such as photo releases. This is also a good time to help the kids boot up their Raspberry Pis and get logged in.)

Meet your teachers:

Katie Cunningham
Barbara Shaurette

(Each teacher should introduce herself/himself and talk about how they use programming - in their jobs or just to create cool things.)

What is programming?

- A **problem** to solve
- A **solution** to the problem
- The solution translated into a language the computer can understand

Before we start learning about the Python programming language, let's talk a little about what programming is.

So what do we mean when we talk about programming? Well, every program starts with a problem you want to solve. The solution to that problem is referred to as an algorithm (and we'll talk about that more in a few minutes). And finally, that solution is translated into a programming language, like Python, that the computer can understand. That package of code that's run on a computer is called a program.

- A **computer** is a machine that **stores** pieces of information.
- A computer also **moves**, **arranges**, and **controls** that information (or *data*).
- A **program** is a detailed set of **instructions** that tells a computer what to do with data.

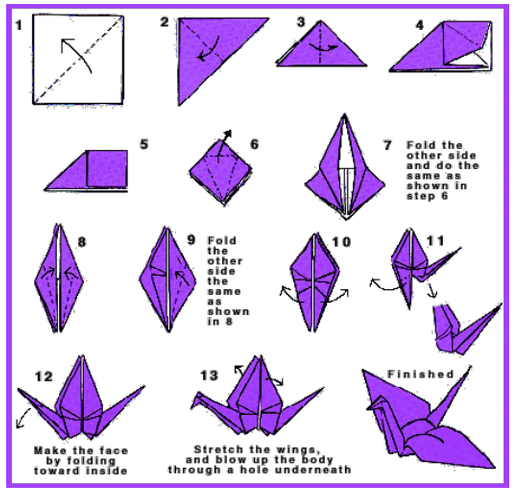
So what about computers? You probably already have a good idea of what they are and what they can do.

A computer is a kind of machine, and what it does basically breaks down to two simple things.

It stores pieces of information, which we call data.

And it does things with that data, using programs.

Computers come in a lot of different forms. You might have a laptop, or a PC. But did you know that your phone might also be a computer? Or maybe you have a game console - that's a computer too.



Sugar Cookies

Preheat oven to 375 degrees — 10 minutes

Ingredients:

2/3 cup margarine
¾ cup sugar
1 egg
½ teaspoon vanilla
2 cups flour
1 ½ teaspoons baking powder
¼ teaspoon salt
4 teaspoon milk

Instructions

Combine margarine, sugar, egg, and vanilla. Separately, combine flour baking powder, and salt. Combine two mixtures along with milk.
Chill 1 hour before baking
Roll out dough to about a ¼ of an inch thickness and cut out cookies.

But there's something you should know about computers.

They're not very smart.

On their own, computers don't really know how to do much. They can really only do what you tell them to do.

So you have to write good instructions for them. And that's what we're going to learn how to do here today.

Here are some examples of instructions you might use for people:

1. Suppose you want to make an origami crane - that's the 'problem' you want to solve. Well, here's a step-by-step guide.

2. Or maybe you need something to write on? Well, there's a diagram - follow all the steps and you'll have yourself a notebook.

3. And what if you're hungry after all that paper folding? Well here's a recipe - just some instructions for making cookies.

Algorithms

And all of those instructions we just saw are like what we call algorithms.

The word 'algorithm' is really just a fancy name for the instructions we give to computers.

They're a lot like recipes, with specific steps to follow.

There are some differences though - algorithms usually have a lot more steps than cookie recipes.

And algorithms are written using special languages - programming languages, like Python.

97 Simple Steps to a PB&J

Is making PB&J difficult?

How many steps does it feel like?

First, let's try creating a set of instructions of our own.

Have you ever made a peanut butter and jelly sandwich before? How many steps do you think it takes?

Let's try it out.

(Demonstration: Go through the steps of making a peanut butter and jelly sandwich. One teacher guides the class by asking students to call out each step, and the other teacher follows the instructions exactly, acting as the ‘computer’, until the sandwich is completed. The results are usually hilarious, and this exercise teaches the kids about how important it is to be specific. Keep a rough count of how many steps it took - at the end, ask the students how many steps they thought it took.)

Let's talk to Python!

And now it's time to start writing some instructions for computers.

There are many languages that we use to talk to computers, but the one we're going to learn about today is called Python.

(Have the students open Idle at this point. Let them know that what they're seeing is sometimes called a shell or interpreter. Introduce the term prompt.)

Idle is a program that's also a programming environment. This is where we're going to write our Python code.

It works a lot like a program called a terminal, something you might use to log in to a server. We'll see what that looks like a little later on.

For now, let's talk about some things you might notice here:

At the very top you'll see the word 'Python' followed by some numbers. Those numbers mean the version of Python that you have installed on your computer. Yours might be a little different than mine, but as long as it starts with a 2 or 3, you should be okay.

The other important thing is the three arrows at the bottom. You might see a flashing cursor next to the arrows, but you might not.

We call this a 'prompt', and this is where we're going to type our code.

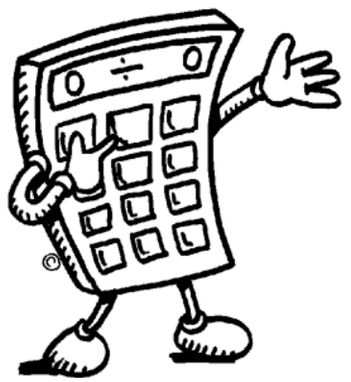
Idle is also what we call an INTERPRETER. Now in real life we know what an interpreter is - it's a person that translates from one language to another. Well in the computer world, we need a program like Idle to translate our Python code into another language that the computer can understand.

So when we type something at the prompt and hit Enter, Idle is
- translating what we've typed
- talking to the computer
- then getting an answer from the computer and sending it back to us in a language that WE can understand

Math

Try doing some math at the prompt:

```
>>> 1 + 2
>>> 12 - 3
>>> 9 + 5 - 15
```



Operators:

add: +

subtract: -

Finally! Let's get started with some very simple math.

Go ahead and type these expressions one at a time. After each one, hit the Enter key and see what happens.

We're using some symbols you probably already know - the plus and minus signs. In the programming world, we call these symbols "operators".

By the way, you have just started writing Python.

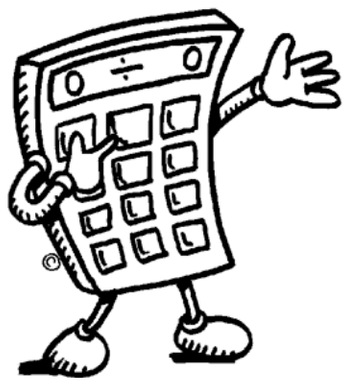
Math

More operators:

divide: /

multiply: *

```
>>> 6 * 5
>>> 6 / 2
>>> 10 * 5 * 3
```



Here are some more symbols, or operators.

For division, we use that slanted line, called a forward slash.

And for multiplication, we use a star symbol, called an asterisk.

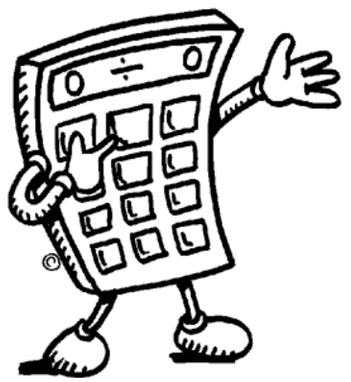
Let's try a few of these expressions and see what happens. While I'm typing these examples, feel free to try some of your own, using different numbers and operators.

Math

Try some more division:

```
>>> 8 / 5
>>> 20 / 7
>>> 10 / 3
```

Are you getting the
results you expected?



Let's try some more division.

Are you getting the results you expect?

These answers should all be in fractions, or decimals, right?

Math

Integers: 9, -55

Floats (decimals): 17.31, 10.0

```
>>> 10/3
3
>>> 10/2
5
```

```
>>> 10/3.0
3.3333333333333335
>>> 10.0/2
5.0
```

★ Rule: If you want Python to answer in floats, you must talk to it in floats.

Well here we're learning something new about how Python works with numbers.

In Python, and in many other programming languages, a decimal number is called a 'float'.

On the right here, you can see some examples of integers - or whole numbers - and how you might get them in Python.

And on the left are some examples of decimal numbers, or floats.

When we type an expression in Python that uses integers, Python will always give an integer back to us as a response.

But when we type an expression using a float, the answer will always be some sort of decimal number.

We use floats when we want the most accurate answer possible - in programming, you can probably guess that we want that most of the time.

Go ahead and try a few more expressions using different numbers. You might notice a few things:

1. Only one number in the expression needs to have a decimal in it for Python to RETURN a float.

2. And numbers that divide evenly - like 10 divided by 2 - will still return a float if there are decimal numbers anywhere in the expression

(demonstrate 10/2, 10.0/2, and 10/2.0)

So the rule we've learned here is that if you want Python to answer with floats, you have to talk to it with floats.

Before we move on to the next thing, let's take a quick minute to mention some words I've been using.

When I say that Python 'returns' something, I'm talking about the answers that Python gives you. When you type something at the prompt and hit Enter, you're asking Python a question. Then Python gives back, or returns, an answer to you.

I've also been using this word 'expression' - you'll hear that a lot more in these videos. So far the expressions we've seen have been math problems, but expressions can be a lot more. We're also going to see expressions that use words and other kinds of symbols.

Math

Comparison operators:

==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

In Python, we can do a lot of things with numbers besides just adding or dividing.

What if we need to find out if one number is larger than another? Or if one number is equal to another? Suppose you're making a video game with Python - you might want to do something like this to compare scores between two players, for example.

Well, here are some more symbols (or operators) that we can use. These are called COMPARISON operators.

(read through the list)

Later on, you'll see some examples of how these can be used in programming. But for now, let's look at how these comparisons work.

Math

Practice:

```
>>> 5 < 4 + 3

>>> 12 + 1 >= 12

>>> 16 * 2 == 32

>>> 16 != 16

>>> 5 >= 6
```

==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Let's practice with a few examples so you can see what kinds of answers you get.

Type each of these expressions at the prompt and hit the Enter key after each one.

Over on the right side of the screen is a guide to help you remember what each of the comparison operators mean.

Take your time, and try a few other expressions if you feel like it.

What happens if you use floats? Does it make a difference?

Math

Practice:

```
>>> 5 < 4 + 3
True
>>> 12 + 1 >= 12
True
>>> 16 * 2 == 32
True
>>> 16 != 16
False
>>> 5 >= 6
False
```

(Explain the first few expressions: identify the comparison operator in each expression, and talk about the ‘order of operations’)

So here are the answers you should have gotten.

Probably the first thing you noticed is that the answers you got back aren't numbers.

Well that makes sense, right? If we're asking if one number is greater than another, the answer is going to be either True or False, something we call a Boolean. We'll talk about Booleans in just a few minutes.

One other thing I want to talk about here is something called the 'order of operations'.

Let's look at the first expression - five less than four plus three.

When the computer reads the expression, do you think it sees:

"Is five less than four? Then add 3."

Hopefully not, because the answer to "Is five less than four?" is False, and you can't add 3 to that.

Instead, Python does the adding first. Four plus three. And four plus three is seven, so what this expression is really asking is whether five is less than seven.

So there's another rule about doing math with Python: simple expressions like adding or multiplying will always be calculated before any of the comparisons.

Strings

We've just covered the basics of using math in Python, so now it's time to talk about a new kind of data, called STRINGS.

When we use the word 'string' in programming, we're talking about characters, like letters or symbols, or a bunch of characters put together, like words.

But maybe the best way to explain what a string is would be to show some examples.

Strings

Examples:

```
>>> "garlic breath"
>>> "Hello!"
```

Try typing one without quotes:

What's the result?

```
>>> apple
```

★ Rule:
A string must be in quotes

```
>>> "apple"
>>> "What's for lunch?"
>>> "3 + 5"
```

Try typing the first two examples exactly as they're written, with quotes around them, and see what you get.

Now try typing the third example and hit Enter.

Did you get an error message?

We'll talk about these error messages a little later on, but for now we've learned a new rule about Python:

If you want Python to read a string, it must be inside quotes.

Look at some of those other examples at the bottom. It looks like numbers, or math expressions, can also be strings - as long as they're inside quotes!

There's one other thing you might have noticed. See that second-to-last example, the one asking "What's for lunch?" ?

That string has a single quote, or an apostrophe, inside it. And that's fine, because the quotes on the outside are double quotes.

But you can also put single quotes inside a string - as long as there are no single quotes inside it.

(talk about escaping quotes?)

Strings

String operators:

concatenation: +
multiplication: *

Try concatenating:

```
>>> "Hi" + "there!"
```

Try multiplying:

```
>>> "HAHA" * 250
```

Remember those operators we used for doing math? Well, we can also use some of them to do things with strings.

And here's a new word: concatenate. Concatenation is a little bit like adding - we use it to put words together, but side to side.

And multiplying controls how many times we show a string.

So try these examples and see what Python gives you.

Strings: Indexes

Strings are made up of **characters**:

```
>>> "H" + "e" + "l" + "l" + "o"
'Hello'
```

Each character has a position called an *index*:

```
H e l l o
0 1 2 3 4
```

In Python, indexes start at **0**

We talked earlier about how strings can be made up of characters, like letters or numbers, or even punctuation marks.

Well each of those characters has a position in the string, and that position is called an INDEX.

Let's look at the example we have here. This string is the word "hello", and it's made up of five characters - five letters.

We could start counting to identify the position of each letter. But here's something interesting to know about Python - instead of counting from one, we're going to start counting from zero.

So in our example, the letter 'H' has an index of zero, the letter 'e' has an index of one, and so on.

Strings: Indexes

```
>>> print “Hello”[0]
H


>>> print “Hello”[4]

>>> print “Hey, Bob!”[4]

>>> print “Hey, Bob!”[6-1]
```

It's pretty easy to figure out indexes when we can count by hand. Now let's try doing it programmatically (that is, using Python).

Try the first example. At your prompt, type the word 'print', then the word "Hello" in quotes, and finally these square brackets with the number zero inside. Then hit Enter.

Since we're asking for the character in this string with an index of zero, Python returns the letter 'H'.

Now let's try the second example, asking Python for the character with an index of four. Did you get the letter 'o'? Let's see why.

If we start at the beginning of the string, the letter 'H' has an index of zero. Then index one is the letter 'e', indexes two and three are 'l'. And index four is the letter 'o'.

Okay, let's try some longer examples. Let's type the word 'print', then in quotes "Hey", comma, space, "Bob", exclamation point. Then right next to that, inside square brackets, type the number four and hit Enter.

Did Python return anything? Are you sure? Let's count by hand and see what the character at index four is. Remember to start with zero - that's the letter 'H'. Index one is the letter 'e', two is the letter 'y', three is the comma, and four is the space. Is that what Python returned?

Okay, one last example. We already know that we can enter an index number inside the square brackets to find its matching character in the string.

But we can also do math inside the square brackets. Here we're subtracting one from six, which is ... five. So what is the character at the fifth index in this string?

Go ahead and type this expression to find out.

Strings: Indexes

Rules:

- ★ Each character's position is called its *index*.
- ★ Indexes start at 0.
- ★ Spaces inside the string are counted.

Here are the rules to remember about indexes of strings.

We now know that each character in a string has a position, and that position is called an index.

We also know that in Python - and in many other programming languages - we start counting for indexes at zero instead of one.

And finally we've seen that all the characters in a string are counted - even spaces.

Variables

So far we've learned a bit about computer basics, and we've used Python to work with numbers and strings.

Now let's look at another way of working with data in Python - something called a variable.

Variables

Calculate a value:

```
>>> 12 * 12
144
```

How can you save that value?

Give the value a name:

```
>>> donuts = 12 * 12
>>> donuts
144
```

Suppose you use a math expression to calculate a value. We've done this already. It's pretty simple, right?

Let's do the first one at the top. Type twelve times twelve and hit Enter. The value you get back should be ... one hundred and forty-four.

But what if you want to use that value again?

You could type twelve times twelve again, but if you're writing a program you might have to type that a lot.

Luckily Python gives us an easier way to do it. You can give your value a name, then you can use that name over and over again.

Take a look at our second example. Here we're using the name 'donuts' then saying that donuts is equal to twelve times twelve. We're ASSIGNING the value of twelve times twelve to the variable 'donuts'.

Go ahead and type the second example exactly as it's written here - donuts equals twelve times twelve - and hit Enter.

You shouldn't get any answer back - we haven't asked for one yet. But type the word donuts again and hit Enter. Do you see your value this time?

Go ahead and type and enter the word donuts a few times. You should always get the same answer back.

Notice that the words 'donuts' does not have any quotes around it. If it had quotes around it, Python would treat it like a string. Without quotes, Python knows that it's a variable.

Variables

Create a variable and give it a value:

```
>>> color = “yellow”
>>> color
‘yellow’
```

Now assign a new value:

```
>>> color = “red”
>>> color
‘red’

>>> color = “fish”
>>> color = 12
```

And once you make a variable, you can give it a new value.

Try typing the example at the top. You're making a variable with the name 'color', then giving it the string 'yellow' as a value.

Next, when you type the word 'color' and press Enter, Python should return the value of your variable, 'yellow'.

Now let's give 'color' a new value. Go ahead and type the second example as it's written - remember that the variable doesn't have quotes around it, but the new value, 'red', does have quotes because it's a string.

When you enter the variable name again, you should see the new value.

Go ahead and try giving your variable some other new values, like different strings or even some numbers.

Variables

- ★ Calculate once, keep the result to use later
- ★ Keep the same name, change the value

Here are some of the rules to remember about variables.

The first thing is that you can use them to store values. You only have to do the calculation once, but you can keep the result around to use later.

The second thing is that you can keep the same name for your variable, but give it different values.

Variables

Some other things we can do with variables:

Math operations

```
>>> donuts = 12 * 12
>>> fishes = 3
>>> fishes + donuts
```

String operations

```
>>> color = “yellow”
>>> day = “Monday”
>>> color + day
>>> color * fishes
>>> color + day * fishes
```

We can also use variables to do some of the same things we do with numbers and strings.

Let's create a couple of variables and give them numeric values. We can use our donuts example from earlier, and also create one new variable. I'm going to give that new variable a value of three.

Now what happens if we add the two variables? Is the answer what you expect?

Go ahead and try some other math operations with these two variables. You can subtract, divide, multiply, even compare them.

And we can do some of the same things with variables that have string values.

Let's create two more variables - color equals yellow and day equals Monday.

The first thing we'll do is put them together with a plus sign. Do you remember what the plus sign does for strings? It concatenates them.

And what happens if we multiply a variable with a string value by a variable with a number value?

Now let's look at this expression where we add and multiply. What do you think Python will return here? Let's type this at our prompt and find out.

Was that the answer you expected? Okay, let's break this down and see what happened. Our expression is color plus day times fishes. We know that color is yellow, day is Monday, and fishes is three.

So when we asked Python to calculate - or EVALUATE - this expression, what did it return? Let's take a look.

('yellowMondayMondayMonday')

You might have been expecting Python to concatenate color and day first, and then multiply those strings times three.

Do you remember when we talked about order of operations earlier? Maybe not? Okay, let's recap that then. "Order of operations" is a rule that determines which parts of an expression are calculated first. In most programming languages, multiplication is always done before adding or subtracting. So in this case, Python did the multiplication side of the operation first - so you got the word Monday three times - and then concatenated that with the word yellow.

But if you wanted the addition to be done first, there is a different way you could write this expression. You could put parentheses around "color plus day".

>>> (color + day) * fishes

In that case, the string "yellow" would be concatenated with the string "Monday" first, and then the whole thing would be multiplied by three. So instead of yellowMondayMondayMonday, you'd end up with yellowMondayyellowMondayyellowMonday.

Variables

More things we can do with variables:

Get an index from a string:

```
>>> fruit = "watermelon"
>>> print fruit[2]
```

Do some math to get the index:

```
>>> mynumber = 3
>>> print fruit[mynumber-2]
```

Let's look at just a couple more things you can do with variables.

Remember when we were looking at strings, and we talked about indexes? An INDEX is the position of a character in a string. Well, if a variable has a value that's a string, we can get an index from it as well.

Let's go ahead and try this first example. You might already have created a variable named fruit from earlier in this lesson. Now let's find the character that's at the index of two in that variable's string value.

Type the print command, the variable, and then a number inside the square brackets. When you hit Enter, did you get the character you were expecting?

And here's one last easy example. We've already seen a few ways we can use variables to do math. Well here's one more way - we can use a variable inside those square brackets to calculate a number.

Go ahead and type this example - create the variable 'mynumber', then use it to get an index from the 'fruit' variable.

Variables

Assigning values or making comparisons?

```
>>> fruit = "watermelon"
>>> 5 = 6
```

```
>>> fruit == "watermelon"
>>> 5 == 6
```

Here's one last thing I want to add before we move on - and this doesn't just apply to variables, but to strings and numbers as well.

When we create a variable and give it a value, we use a single equals sign to assign that value.

But remember earlier when we were working with numbers and we used the double equals sign to do comparisons?

One equal sign or two? How can you remember which one to use?

Well here's how I remember.

When we're assigning a value, we're just saying "this equals that". That's a short sentence, so it only gets one equal sign.

But when we're comparing values, we're asking "is this thing equal to that thing?". And that's a longer sentence, so it gets two equal signs.

Errors

Now we're going to talk about something that's really important in programming - errors and error messages.

Error messages are our friends - they're a good thing, because they tell us what went wrong. Without error messages, it's hard to fix something that's broken.

Errors

```
>>> “friend” * 5
‘friendfriendfriendfriendfriend’

>>> “friend” + 5
Error

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Do you remember what ‘concatenate’ means?
What do you think ‘str’ and ‘int’ mean?

Try entering these expressions and see what answers Python gives you. The first expression multiplies the word ‘friend’ and prints it 5 times.

What is the second expression supposed to do? Should it concatenate “friend” and 5? What happens instead?

Errors

```
>>> “friend” + 5

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
    ^                   ^         ^          ^
```

- Strings: ‘str’
- Integers: ‘int’
- Both are objects
- Python cannot concatenate objects of different *types*

Now let’s take a look at that error message and see what it’s really telling us.

The first two lines are pretty common to all error messages, so they won’t really help us much here. But the last line gives us some valuable information.

We tried to concatenate two pieces of data - which Python calls ‘objects’ - the string “friend” and the number 5. But Python isn’t able to concatenate those two objects because they’re of different types. And so we get what’s called a “TypeError”.

Errors

How can we fix this error? Concatenation won't work.

```
>>> "friend" + 5
Error
```

What if we make 5 a string?

```
>>> "friend" + "5"
friend5
```

What's another way that we could fix this error?

Let's do something new with the `print` command:

```
>>> print "friend", 5
friend 5
```

(point out the comma used with the print command, and how it adds a space between the two objects)

Types of data

Data types

We already know about three types of data:

“Hi!”	string
27	integer
15.238	float

Python can tell us about types using the `type()` function:

```
>>> type(“Hi!”)
<type ‘str’>
```

Can you get Python to output `int` and `float` types?

Just like ‘print’, ‘type’ is a built-in function that comes with Python.

Python has a lot of built-in functions, which we’ll learn about later.

Data type: Booleans

Now let's talk about another new type that you'll work with in Python.

Booleans

A Boolean value can be: `True` or `False`.

Is 1 equal to 1?

```
>>> 1 == 1
True
```

Is 15 less than 5?

```
>>> 15 < 5
False
```

A boolean only has two possible values - it can either be True or False.

Booleans are a pretty simple idea, but they're very important - we use them in programming a lot when we need to make decisions about what to do in our code. For example, 'If an expression is True, do something; if that expression is False, do something else instead'.

Notice that we're also using some of our comparison operators here - equal to and less than.

Booleans

What happens when we type Boolean values in the interpreter?

```
>>> True
>>> False
```

When the words 'True' and 'False' begin with *upper case* letters, Python knows to treat them like Booleans instead of strings or integers.

```
>>> true
>>> false
>>> type(True)
>>> type("True")
```

Booleans

and

If both are True:

```
>>> 1==1 and 2==2
True
```

If only one is True:

```
>>> 1==1 and 2==3
False
```

If both are False:

```
>>> 1==2 and 2==3
False
```

Here's an interesting way that we can use some of those 'comparison' operators we talked about earlier.

Let's see what happens when we use the word 'and' between two comparisons. If both comparisons are True, then the whole expression will be true. But what happens if one of the expressions is False?

(use the example of purchasing fish from a pet store - you must have an id card AND an aquarium to put it in - if you have both of those things, you can buy a fish)

Booleans

or

If both are True:

```
>>> 1==1 or 2==2
True
```

If only one is True:

```
>>> 1==1 or 2!=2
True
```

If both are False:

```
>>> 1==2 or 2==3
False
```

When we use the word ‘and’ between two comparisons, we get different answers.

With ‘or’, as long as at least one comparison is True, the whole expression is considered True.

But if both are False, then the whole thing is False.

(extend the example of purchasing fish from a pet store - you can pay with cash OR a credit card OR a check - as long as you have one of those things, you can make your purchase)

Booleans

not

You can use the word **not** to reverse the answer that Python gives:

```
>>> 1==1
True
>>> not 1==1
False
```

Any expression that is True can become False:

```
>>> not True
False
```

So what do you think happens when we use the word ‘not’ in front of a comparison?

Booleans

You can also use booleans in their own expressions:

```
>>> True and True
>>> True and False
>>> False and False

>>> True or True
>>> False or True
>>> False or False

>>> not True and True
>>> not True or True
```

Booleans: Practice

Try some of these expressions in your interpreter.

See if you can predict what answers Python will give back.

```
>>> True and True
>>> False and True
>>> 1 == 1 and 2 == 1
>>> "test" == "test"
>>> 1 == 1 or 2 != 1
>>> True and 1 == 1
>>> False and 0 != 0
>>> True or 1 == 1
>>> "test" == "tests"
>>> 1 != 0 and 2 == 1
```

(In case we want to send them online later: http://bit.ly/boolean-practice)

Data type: Lists

Lists

List: a sequence of objects

```
>>> fruit = [“apple”, “banana”, “grape”]
>>> numbers = [3, 17, -4, 8.8, 1]
```

Guess what this will output:

```
>>> type(fruit)


>>> type(numbers)
```

Lists

List: a sequence of objects

```
>>> fruit = [“apple”, “banana”, “grape”]
>>> numbers = [3, 17, -4, 8.8, 1]
```

Guess what this will output:

```
>>> type(fruit)
<type ‘list’>

>>> type(numbers)
<type ‘list’>
```

Lists

Index: Where an item is in the list

```
>>> fruit = [“apple”, “banana”, “grape”]
>>> fruit[0]
'apple'
```

```
['apple', 'banana', 'grape']
    0         1         2
```

Python always starts at zero!

Lists

Make a **list** of three of your favorite colors.

Use an **index** to print your favorite color's name.

(Give the students a few minutes to work out the solution for themselves, then switch over to Idle and demonstrate the solution as seen in the next slide.)

Lists

Make a **list** of three of your favorite colors.

```
>>> colors = ['red', 'orange', 'purple']
```

Use an **index** to print your favorite color's name.

```
>>> print colors[1]
```

(If there's time and interest, this is a good place to switch over to Idle and demonstrate .append() to add items to the list.)

Logic

When we talk about logic, we're talking about making decisions about what to do next in our code.

`if` Statements

One of the ways we do that is with “if statements”.

`if` Statements

Making decisions:

"**If** you're hungry, let's eat lunch."

"**If** the trash is full, go empty it."

If a condition is met, perform an action:

```
>>> name = "Katie"
>>> if name == "Katie":
        print "Hi Katie!"

Hi Katie!
```

Here are some examples of some ‘if statements’ that you might use in real life, and an example of how you might use that in code.

(Explain indentation here - Idle should automatically indent for them, but they need to know that in most Python interpreters they will need to indent 4 spaces themselves)

`if` Statements

Adding a choice:

"**If** you're hungry, let's eat lunch.
Or **else** we can eat in an hour."

"**If** there's mint ice cream, I'll have a scoop.
Or **else** I'll take vanilla."

Adding a choice in our code with the *else clause*:

```
>>> if name == "Katie":
        print "Hi Katie!"
    else:
        print "Impostor!"
```

Now let's add one extra choice.

`if` Statements

Adding many choices:

"**If** there's mint ice cream, I'll have a scoop.
Or **else** if we have vanilla, I'll have 2!
Or **else** if there's chocolate, give me 3!
Or I'll just have a donut."

Adding more choices in our code with the *elif clause*:

```
>>> if name == "Katie":
        print "Hi Katie!"
    elif name == "Barbara":
        print "Hi Barbara!"
    else:
        print "Who are you?"
```

But what if we have many options to choose from?

`if` Statements

`if/elif/else` practice

Write an if statement that prints "Yay!" if the variable named color is equal to "yellow".

Add an *elif clause* and an *else clause* to print two different messages for other values of the variable.

(Here's our last example)

```
>>> name = "Katie"
>>> if name == "Katie":
        print "Hi Katie!"
    elif name == "Barbara":
        print "Hi Barbara!"
    else:
        print "Who are you?"
```

`if` Statements

`if/elif/else` practice

Write an if statement that prints "Yay!" if the variable named color is equal to "yellow".

Add an *elif clause* and an *else clause* to print two different messages for other values of the variable.

```
>>> color = "blue"
>>> if color == "yellow":
        print "Yay!"
    elif color == "purple":
        print "Try again!"
    else:
        print "We want yellow!"
```

Loops

Loops

Loops are chunks of code
that repeat a task over and over again.

★*Counting* loops repeat a
certain number of times.

★*Conditional* loops keep
going until a certain thing happens
(or as long as some condition is True).

Loops

Counting loops repeat a certain number of times - they keep going until they get to the end of a count.

```
>>> for mynum in [1, 2, 3, 4, 5]:
        print "Hello", mynum
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

The *for* keyword is used to create this kind of loop, so it is usually just called a *for loop*.

(Break down the “for” statement - for each element in the list)

Loops

Conditional loops repeat until something happens
(or as long as some condition is True).

```
>>> count = 0
>>> while (count < 4):
        print 'The count is:', count
        count = count + 1
The count is: 0
The count is: 1
The count is: 2
The count is: 3
```

The *while* keyword is used to create this kind of loop,
so it is usually just called a *while loop*.

(Point out how the counter is increasing each time we go through the loop. Talk about infinite loops.)

Functions

Functions

Remember our PB&J example?

Which looks easier?:

1. Get bread
2. Get knife
4. Open peanut butter
3. Put peanut butter on knife
4. Spread peanut butter on bread
5. ...

1. Make PB&J

Functions are a way to *group* instructions.

Functions

What it's like in our minds:

"Make a peanut butter and jelly sandwich."

In Python, you could say it like this:

```
make_pbj(bread, pb, jam, knife)
```

function **name** function **parameters**

We all know how to make a PB&J.

But we don't have to think about all the steps it takes every time, because the steps are already grouped together in our minds as "make a peanut butter and jelly sandwich".

Functions

What if we wanted to make many kinds of sandwiches?

> "Make a peanut butter and jelly sandwich."
> "Make a cheese and mustard sandwich."

In Python, it could be expressed as:

```
make_sandwich(bread, filling, toppings)
```

function ***name*** *function* ***parameters***

Functions

Let's define a function in the interpreter:

```
>>> def say_hello():
        print 'Hello'
```

Now we'll call the function:

```
>>> say_hello()
Hello
```

Let's put together some of the things we've already learned and write some functions.

This function doesn't have any parameters, but it still does something - that 'something' that it does is called the **body** of the function.

(Reinforce the idea of indentation, make sure the students are checking that each time they enter a line.)

(Emphasize the difference between defining and calling the function.)

Functions

Let's define a function with parameters:

```
>>> def say_hello(myname):
        print 'Hello', myname

>>> say_hello("Katie")
Hello Katie

>>> say_hello("Barbara")
Hello Barbara
```

(Reinforce the notion of indentation, make sure the students are checking that each time they enter a line.)

See how we can make our function do different things just by passing in different arguments?

(Reinforce the difference between defining and calling a function)

Functions

A few things to know about functions ...

```
>>> def say_hello(myname):
        print 'Hello', myname
```

`def`	This is a **keyword**	We use this to let Python know that we're defining a function.
`myname`	This is a **parameter** (and a **variable**).	We use this to represent values in the function.
`print ...`	This is the **body**	This is where we say what the function *does*.

(Emphasize that the 'myname', and that it can be named anything as long as it matches what's in the body of the function)

Functions: Practice

1. Work alone or with a neighbor to create a function that **doubles a number** and prints it out.

(Here's our last example)

```
>>> def say_hello(myname):
        print 'Hello', myname
```

(Give the students some time to work it out for themselves, then move to Idle and demonstrate the solution for them.)

Functions: Practice

1. Work alone or with a neighbor to create a function that **doubles a number** and prints it out.

```
>>> def double(number):
        print number * 2

>>> double(14)
28

>>> double("hello")
hellohello
```

Functions: Practice

2. Work alone or with a neighbor to create a function that takes **two numbers**, multiplies them together, and prints out the result.

(Here's our last example)

```
>>> def double(number):
        print number * 2

>>> double(14)
28
```

(Give the students some time to work it out for themselves, then move to Idle and demonstrate the solution for them.)

Functions: Practice

2. Work alone or with a neighbor to create a function that takes **two numbers**, multiplies them together, and prints out the result.

```
>>> def multiply(num1, num2):
        print num1 * num2

>>> multiply(4, 5)
20

>>> multiply("hello", 5)
hellohellohellohellohello
```

Functions: Output

`print` displays something to the screen.

```
>>> def double(number):
        print number * 2
```

We call the function, passing it the number 12:

```
>>> double(12)
24
```

We call the function again, passing it the number 12 and assigning it to the variable new_number:

```
>>> new_number = double(12)
24
```

But what happens here?

```
>>> new_number
```

We've worked with the 'print' function in a few of our examples, so we know what it does - we give it a value and it shows that value in our interpreter. But all it does is display that value - the value isn't saved.

Let's look at our example. We've defined a function called 'double' that takes a number and multiplies it by two. The first time we call that function and assign its value to the variable 'new_number', it will return the number 24.

But the next time you enter 'new_number', it doesn't have that value (24) anymore - the value hasn't been saved.

Functions: Output

This time let's use `return` instead of `print`.

We call the function, passing it the number 12:

We call the function again, passing it the number 12 assigning the value to the variable new_number:

Now what happens here?

```
>>> def double(number):
        return number * 2

>>> double(12)
24

>>> new_number = double(12)
24

>>> new_number
```

This time when you give 'new_number' a value from the function, it will return that value (24), and now the value is saved.

When you type 'new_number' again, you'll see the same value (24) until you decide to change it.

Functions

- ★ Functions are **defined** using **`def`**.
- ★ Functions are **called** using **parentheses**.
- ★ Functions take **parameters** and can return **outputs**.
- ★ `print` *displays* information, but does not give a value
- ★ `return` gives a **value** to the caller (that's you!)

We've learned a lot about functions in Python - let's go over some of the rules.

Input

Input

Input is information we pass to a function so that we can do something with it.

```
>>> def hello_there(myname):
      print "Hello", myname
```

In this example, the string "Katie" is the input, represented by the variable myname.

```
>>> hello_there("Katie")
'Hello there Katie'
```

Input

The `raw_input()` function takes *input* from the user - you give that input to the function by typing it.

```
>>> def hello_there():
        print "Type your name:"
        name = raw_input()
        print "Hi", name, "how are you?"
```

Here's another way to pass input to a function - 'raw_input' is a built in Python function that let's us interact with our function in a different way.

Input

```
>>> def hello_there():
        print “Type your name:”
        name = raw_input()
        print “Hi”, name, “how are you?”

>>> hello_there()
Type your name:
Barbara
Hi Barbara how are you?
```

Input

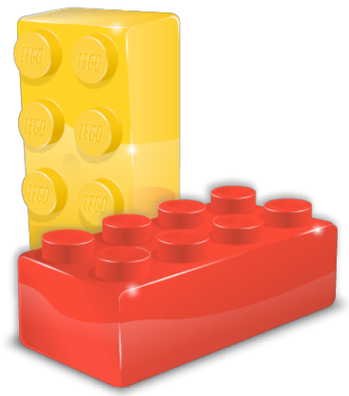
A shortcut:

```
>>> def hello_there():
        name = raw_input("Type your name: ")
        print "Hi", name, "how are you?"

>>> hello_there()
Type your name: Barbara
Hi Barbara how are you?
```

Modules

Modules



A module is a block of code that can be combined with other blocks to build a program.

You can use different combinations of modules to do different jobs, just like you can combine the same LEGO blocks in many different ways.

We've learned about how to define some functions of our own.

But Python has a lot of functions that come built in - let's learn how to use a few of them.

Modules

Lots of modules are included in the Python Standard Library.

Here's how you can use a few of these modules:

Generate a random number between 1-100:

```
>>> import random
>>> print random.randint(1,100)
```

What time zone does your computer think it's in?:

```
>>> import time
>>> time.tzname
```

Print a calendar for this month!:

```
>>> import calendar
>>> calendar.prmonth(2015, 4)
```

When you 'import' a module, you can use all the functions inside that module.

(New term: "dot notation")
(Emphasize the distinction between the module name and the function name.)

Modules

Print the names of all the files in a directory:

```
>>> import os
>>> for file in os.listdir("/home/pi"):
        print file
```

Open a web page and read it:

```
>>> import urllib
>>> myurl =
  urllib.urlopen('http://www.python.org')
>>> print myurl.read()
```

(Talk about how many other modules can be found in the standard library.)

Modules

Turtles!

```
>>> import turtle
>>> turtle.reset()
>>> turtle.forward(20)
>>> turtle.right(20)
>>> turtle.forward(20)
>>> turtle.bye()
```

You can find out about other modules at: http://docs.python.org

Talk about how many other modules can be found in the standard library.

Let's make a game!

(Go over the guessing game examples.)

Games!

Open a new window (File > New Window) and type these lines:

```
secret_number = 7

guess = input("What number am I thinking of? ")

if secret_number == guess:
    print "Yay! You got it."
else:
    print "No, that's not it."
```

From the menu, choose Run > Run Module. Save as 'guess.py'.
What do you see in the interpreter?

In Idle, open a new window. Type the code, then choose Run > Run Module. If Idle asks you to save, just press OK and then give the file a name like 'guess.py' and press enter.

Now you should be back in your interpreter - what do you see there? Let's walk through the code and see if we can tell what it's doing.

Games!

Open a new window (File > New Window) and type these lines:

```
from random import randint

secret_number = randint(1, 10)

while True:
    guess = input("What number am I thinking of? ")

    if secret_number == guess:
        print "Yay! You got it."
        break
    else:
        print "No, that's not it."
```

Choose Run > Run Module. Save as ‘guess2.py’.
What do you see in the interpreter?

Let’s change our game a little bit. Open a new window, type this code, then choose Run > Run Module. If Idle asks you to save, just press OK and give the file a name like 'guess2.py'.

What do you see in your interpreter this time? (Walk through the code and have the students explain what it’s doing.)

Games!

Open a new window (File > New Window) and type these lines:

```
from random import randint

secret_number = randint(1, 10)

while True:
    guess = input("What number am I thinking of? ")

    if secret_number == guess:
        print "Yay! You got it."
        break
    elif secret_number > guess:
        print "No, that's too low."
    else:
        print "No, that's too high."
```

Choose Run > Run Module. Save as ‘guess3.py’.
What do you see in the interpreter?

Let’s change our game even more. Open another new window, type this code, then choose Run > Run Module. This time, give the file the name 'guess3.py'.

What do you see in your interpreter this time? (Walk through the code and have the students explain what it’s doing.)

Minecraft!

(Put this slide up at the end so that students know they where to get help setting up their Raspberry Pis once they're home.)

Minecraft!

```
>>> from mcpi import minecraft
>>> mc = minecraft.Minecraft.create()

>>> mc.postToChat("Hello world”)

>>> pos = mc.player.getPos()
>>> print pos.x, pos.y, pos.z

>>> mc.player.setPos(pos.x, pos.y+100, pos.z)

>>> mc.setBlock(pos.x+1, pos.y, pos.z, 1) (Air: 0, Grass: 2, Dirt: 3)
```

(Put this slide up at the end so that students know they where to get help setting up their Raspberry Pis once they’re home.)

Minecraft!

```
>>> from mcpi import block

>>> dirt = block.DIRT.id
>>> mc.setBlock(x, y, z, dirt)

>>> stone = block.STONE.id
>>> mc.setBlocks(x+1, y+1, z+1, x+11, y+11, z+11, stone)

>>> tnt = 46
>>> mc.setBlocks(x+1, y+1, z+1, x+11, y+11, z+11, tnt)
>>> mc.setBlocks(x+1, y+1, z+1, x+11, y+11, z+11, tnt, 1)
```

(Put this slide up at the end so that students know they where to get help setting up their Raspberry Pis once they're home.)

Congratulations!

You're now a Pythonista!

(Optional - if you have extra time to use up, send the kids here to take a quiz and test their new programming knowledge: http://tinyurl.com/yc-quiz)

What can YOU do with Python?

- Make more games
- Edit music and videos
- Build web sites
- Write a program that does your homework for you ...
- What are some other ideas?

Now that you've learned the basics of Python, and even written a few programs of your own, what new programs would you like to write?

(Use this slide to open up some discussion at the end of the class, especially if you have extra time to fill.)

Raspberry Pi

Help setting up your new computer:

http://www.raspberrypi.org/quick-start-guide

Minecraft on your new Raspberry Pi:

https://www.raspberrypi.org/learning/getting-started-with-minecraft-pi/worksheet/

http://www.stuffaboutcode.com/p/minecraft-api-reference.html

(Put this slide up at the end so that students know they where to get help setting up their Raspberry Pis once they're home.)