Let's Learn Python!

Young Coders at PyCon 2014

Leave this screen up before the class begins, as the kids are coming in.

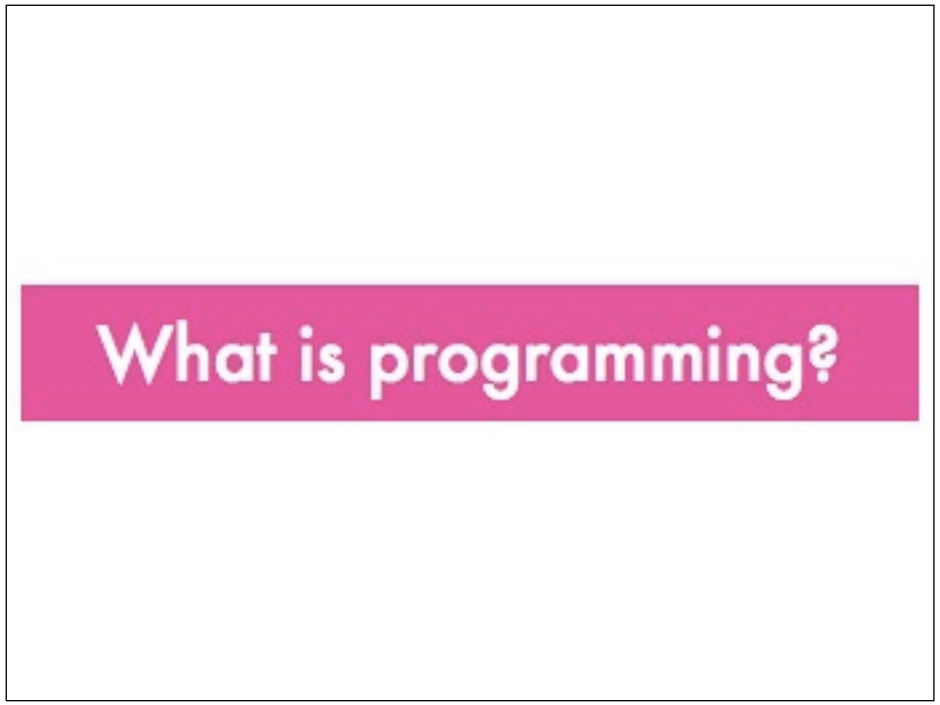
Use this as an opportunity to pass out name tags (if you have them), and collect any paperwork (such as photo releases).

This is also a good time to help the kids boot up their Raspberry Pis and get logged in.

Meet your teachers:

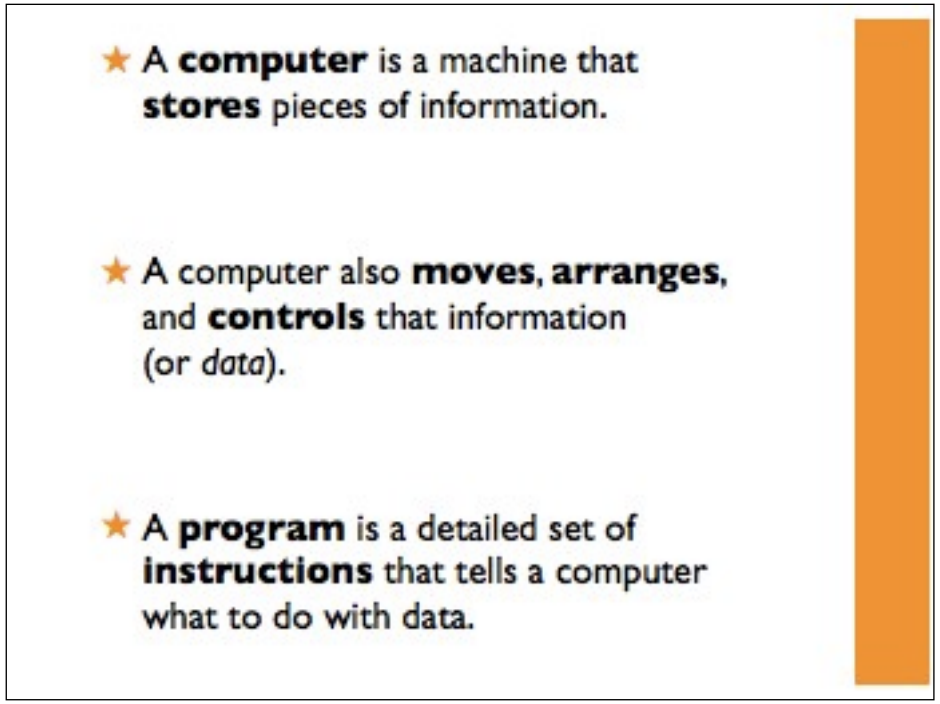
Katie Cunningham
Barbara Shaurette

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



"We're going to start by taking a little bit of time to talk about what 'programming' is."

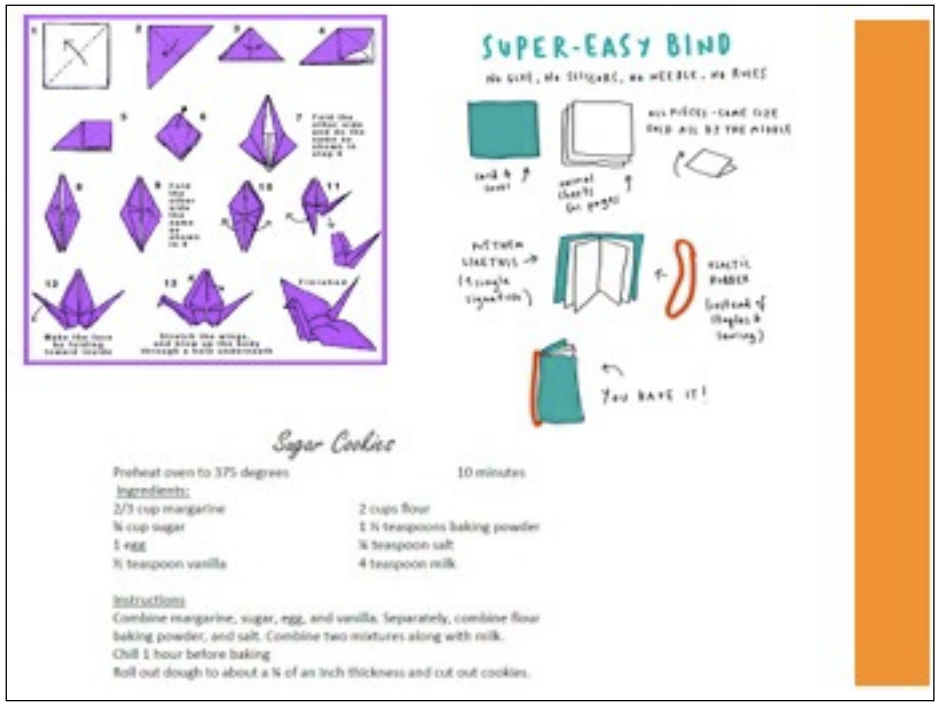
Ask the kids what they think the word 'programming' means. Do they know anyone who does programming? Maybe one or both parents are programmers – do they know how their parents use programming?



Let's start with some basics. We all know what a computer is, right?

A computer stores information – which we also call data.

A computer can also do things with that information – using what we call a program.



There's something that you should know about computers - they are not very smart.

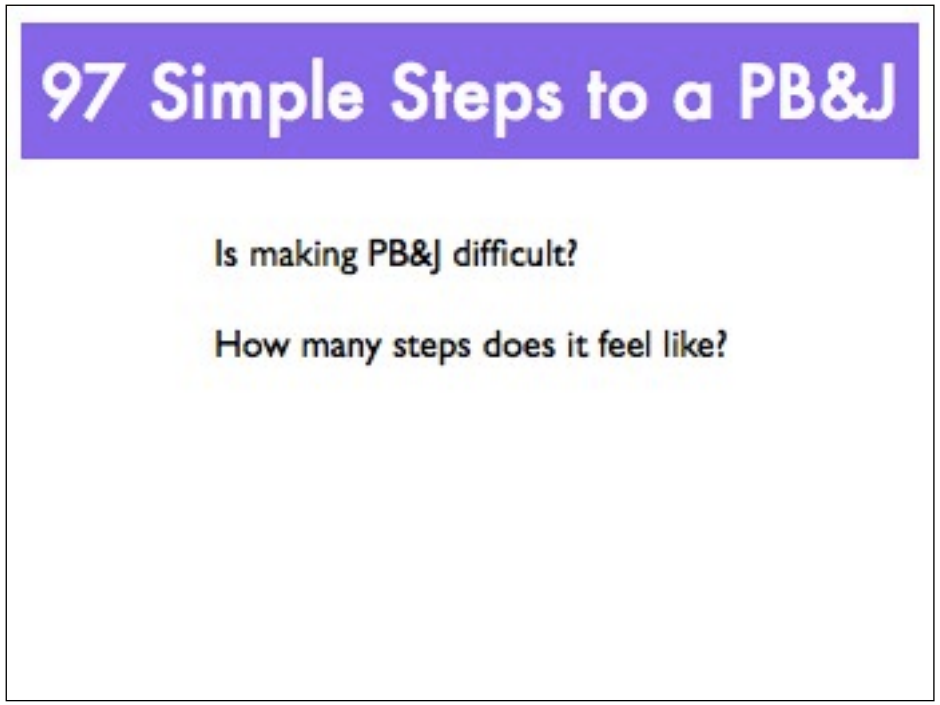
They can only do what you tell them to do. So you have to write good instructions for them. And that's what we're going to learn how to do today.

Here are some examples of instructions that you might have for people - we've got some step-by-step instructions for making an origami crane up in one corner. Then there's a diagram with steps for making a simple book. And down at the bottom, there's a recipe with the steps for making sugar cookies.

Algorithms

Maybe you've heard this word 'algorithm' before. An algorithm is really just a fancy name for the instructions that 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 chocolate cookie recipes. And algorithms are written using special languages - programming langages, like Python, which we're going to talk about in a few minutes.



But first, let's try creating a set of instructions of our own. How many in the class have 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.)

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, but let them know that there will be more explanation of those things in a few minutes.

Math

Try doing some math in the interpreter:

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

Operators:

add: +

subtract: -

Let's start with some very simple math. Go ahead and type these simple expressions in the interpreter, hit enter, and see what happens.

We're using some familiar symbols to add and subtract - in the programming world, we call those 'operators'.

By the way, you are **already** writing Python.

Math

More operators:

divide: /

multiply: *

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

Here are some more symbols, or operators. For division, that slanted line - or forward slash - probably looks familiar. But for multiplication, we use an asterisk.

Try a few of these expressions in your interpreter and see what happens - feel free to experiment and use some different numbers and expressions.

Math

Try some more division in the interpreter:

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

Are you getting the results you expected?

These answers should all be in fractions, or decimal numbers, right?

Math

Integers:	Floats (decimals):
9	17.318
-55	10.0
>>> 10/3	>>> 10/3.0
3	3.6666666666666665
>>> 10/2	>>> 10.0/2
5	5.0

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

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

On the right, you see some examples of integers, and on the left is a list of decimal numbers, or floats.

When we type an expression using a 'float', the answer will always be some sort of decimal number. We use floats when we want to get the most accurate answer possible - which, in programming, is pretty much all of the time.

(Some other things to point out:
- only one side of the expression needs to have a decimal in it for the number returned to be a float

Math

Comparison operators:

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

Suppose we need to find out if one number is larger than another number? Another thing we can do in Python is use **comparison operators**.

(read through the list of operators)

Later on, you'll see some examples of how these are used in programming. But for now let's just look at some examples of how 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 comparisons on the interpreter so that you can see what kinds of answers you get.

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

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')

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.

Strings

Examples:

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

Try typing one without quotes:

```
>>> apple
```

What's the result?

★ Rule:
A string must be in quotes

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

When we use the word 'string' in programming, we're talking about characters – like letters – or a series of characters, like words.

Try typing the first two examples and see what you get.

Now try the third example. Did you get an error? We'll talk about those error messages a little later, but for now we've learned a new rule about Python: If you want Python to read a string, it must be in quotes.

A number can also act like a string, but only if it's wrapped in quotes.

Strings

String operators:
concatenation: +
multiplication: *

Try concatenating:

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

Try multiplying:

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

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

Concatenation is a lot like adding – we use it to put words together. And multiplying controls how many times we show a string.

Try these examples and see what answers 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**

Here are some other things you need to know about strings:

All strings are made up of characters, and each character has a numbered position, called an index.

(In the class with older kids, some of them asked "why?" indexing starts at zero – we felt it was too much to address in this class. You might give a brief explanation, then suggest that the kids to research it (via reading or Google) for something deeper.)

Strings: Indexes

```
>>> print "Hello"
Hello

>>> print "Hello"[0]
H
>>> print "Hello"[4]
o

>>> print "Hey, Bob!"[6]
o
>>> print "Hey, Bob!"[6 - 1]
B
```

Let's try typing some examples that use indexes.

We're using something else new here – the 'print' command. 'Print' does just what it sounds like – it will **print** any expression that comes after it.

(explain the expressions – point out the print command, the string, and the index number that comes in brackets after the string)

Strings: Indexes

```
>>> print "Hey, Bob!"[4]
```

What did Python print?

Rules:

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

Variables

We've talked about using math and strings in Python.

Now let's talk about another way of working with both of those things - 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. Try typing in the expression at the top.

The value you get back is '144'. But what do you do if you want to use that value again?

You could type '12 * 12' again, but if you're writing a program you might have to type that a lot. There's an easier way to do it – give your value a name, then you can use that name over and over.

Notice that the word 'donuts' is not inside quotes in our example. That's because we're using the word as a **variable** – if it were in quotes, it would be a **string**.

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
```

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' and giving it the value 'yellow'. When you type the word 'color' and press enter, the interpreter should give you the value of your variable – 'yellow'.

Now assign a new value and type 'color' in the interpreter. Try giving your variable other values, like different strings or even numbers.

Variables

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

Some other things we can do with variables:

Get an index from a string:

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

Do some math:

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

Here are some of the rules to remember about variables:
- 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
- you can keep the same name for your variable, but give it different values

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

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 "**Type**Error".

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?

Data type: Lists

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.

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

Lists have indexes just like strings.

Remember how we got the index of a string?

```
>>> print "apple"[0]
a
```

We can do the same thing with a list:

```
>>> fruit
['apple','banana','grape']

>>> print fruit[0]
'apple'
```

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.)

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
```

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")
```

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

and

If both comparisons are True:

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

If only one of the comparisons is True:

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

If both of the comparisons 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?

Booleans

or

If both comparisons are True:

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

If only one of the comparisons is True:

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

If both of the comparisons 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.

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
```

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.

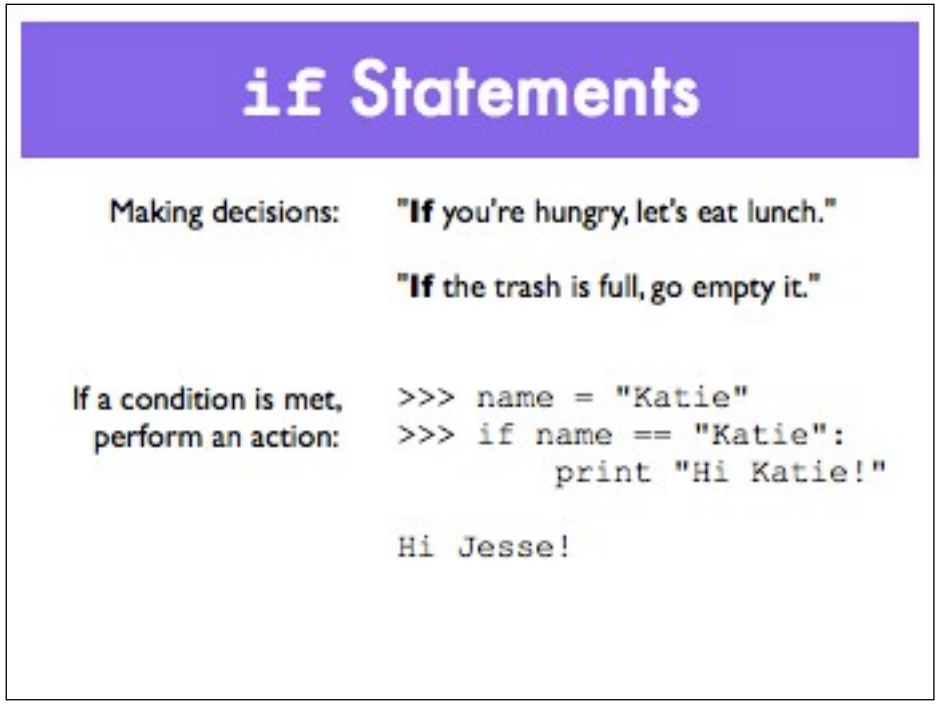
```
>>> True and True
>>> False and True
>>> 1 == 1 and 2 == 1
>>>"test" == "test"
>>> 1 == 1 or 2 != 1
>>> True and 1 == 1
>>> False and 0 != 0
>>> True or 1 == 1
>>>"test" == "testing"
>>> 1 != 0 and 2 == 1
```

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

Logic

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

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



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

Let's define a function in the interpreter:

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

Remember that the second line should be indented 4 spaces.

After typing the second line, hit enter one more time until you see the prompt again.

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

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

Functions

Now we'll call the function:

```
>>> say_hello("Katie")
Hello, Katie

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

Use your new function to say hello to some of your classmates!

(Emphasize the difference between **defining** and **calling** the 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.

(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
```

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
```

(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.

(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`.

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

We call the function, passing it the number 12:

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

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

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

Now what happens here?

```
>>> 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.

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

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

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

Input

`raw_input()` takes *input* from the user - you.

```
>>> 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?
```

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

Here's an even easier way to use `raw_input()`:

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

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

Objects

We've used this word – 'objects' – in a few of our examples, but so far we haven't talked much about what they are. Think of them as 'things' – it's really that simple.

In Python, everything can be an object. We know about some of our data types – numbers, strings, and lists are all objects. And once we define a function, that function also becomes an object.

Objects

In the real world, objects have:

- things that you can do to them (actions)
- words that describe them (properties)

In Python:

- "things you can do" to an object are called *methods*
- "words that describe" an object are called *attributes*

Objects

If this ball is an *object* named myBall, it might have these *attributes*:

```
myBall.color
myBall.size
myBall.weight
```

You can display them:

```
print myBall.size
```

You can assign values to them:

```
myBall.color = 'green'
```

You can assign them to attributes in other objects:

```
anotherBall.color = myBall.color
```

Objects

The ball object might have these *methods*:

```
myBall.kick()
myBall.throw()
myBall.inflate()
```

Methods are the things you can *do* with an object.

Methods are chunks of code - *functions* - that are included inside the object.

Objects

In Python, a *class* is like a description - or blueprint - of an object.

```
class Ball:

  color = 'red'
  size = 'small'
  direction = ''

  def bounce(self):
    if self.direction == 'down':
      self.direction == 'up'
```

Here's how we create a new object.

We use the **class** keyword to name our object. The attributes that we talked about – color, size, and direction are inside this class. So is the method, bounce() – it looks a lot like a function, doesn't it?

Objects

Once we create an object by defining its class, we can call it in a program by creating an instance of it.

Creating multiple instances of an object:

```
>>> my_ball = Ball()
>>> your_ball = Ball()
>>> her_ball = Ball()
```

Giving these instances some attributes:

```
>>> my_ball.color = "purple"
>>> your_ball.color = "red"
>>> her_ball.color = "yellow"
```

Now let's try out one of the methods:

```
>>> my_ball.bounce()
```

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(2014, 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()
```

You can find out about other modules at: 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.)

Raspberry Rogue

(Introduce the Raspberry Rogue code at https://github.com/kcunning/Katie-s-Rougish-PyGame)

Congratulations!
You're now a Pythonista!

What can YOU do with Python?

- Make more games
- Play 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.)

(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.)