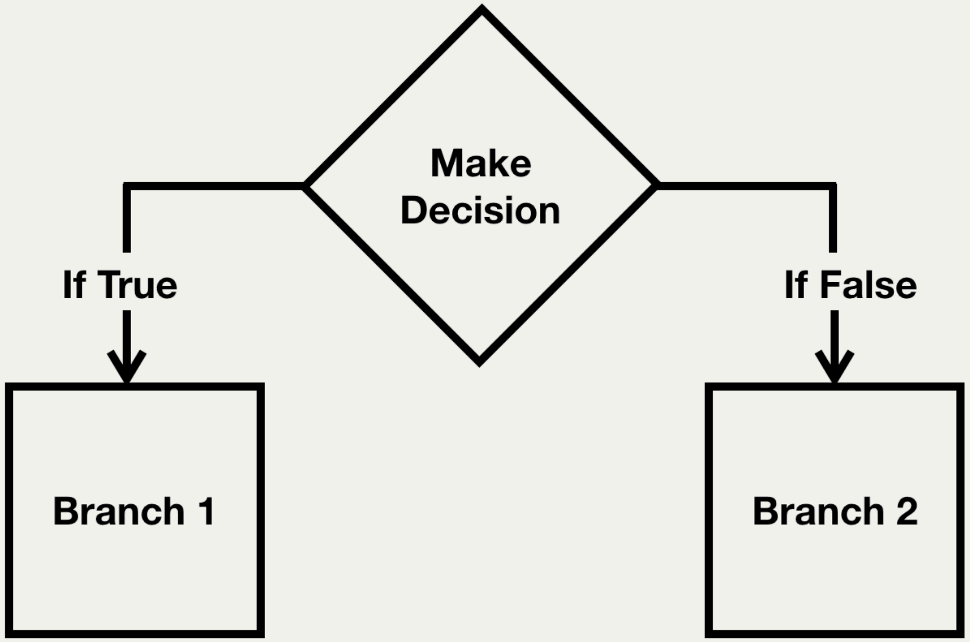
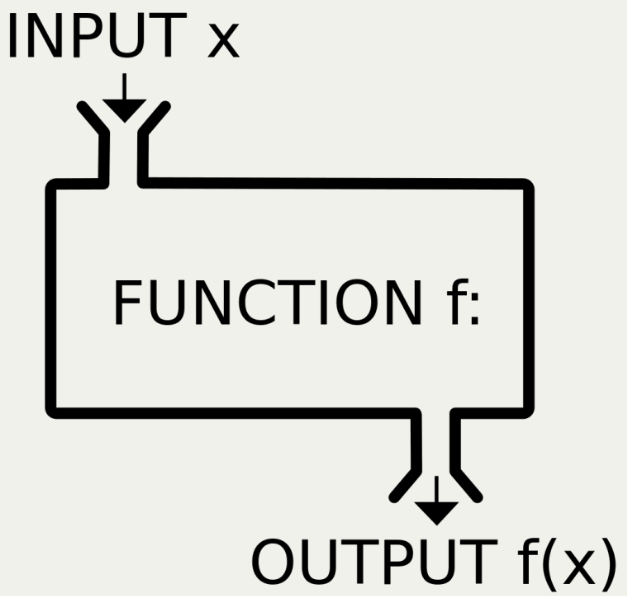
Lecture 2: Functions and Branching

CS5001 / CS5003:
Intensive Foundations
of Computer Science





PDF of this presentation

Lecture 2: Automatically creating run configurations in PyCharm

While I was writing code for this lecture, I discovered how to automatically create a configuration, instead of having to go through the process from lab and assignment 1. See the video below:

https://www.youtube.com/embed/iP8lV_G2M1s?enablejsapi=1

Lecture 2: Functions and Branching

Today:

- Functions
  - Passing values to functions
- Libraries (modules) and "batteries included" code
- Function return values
- Branching
- Testing your code with the `doctest` module

Lecture 2: Functions

During the first lecture, we saw two *functions*:

- `print()`
- `input()`

In Python, a function is a set of statements that are combined together to run as a group. There are many statements inside of the **`print`** function to get it to do its job, and we don't have to worry about any of those details to use the **`print`** function:

```
print("This gets printed to the screen!")
```

A function is called by using the name of the function, and then parentheses, and then zero or more *arguments* to the function. Arguments are the details that the function will use. In the above call to **`print`**, the there was one argument, which was the string, **`"This gets printed to the screen!"`**. We also had a single argument for the **`input`** function:

```
animal = input("What is your favorite animal? ")
```

Lecture 2: Functions

You will come across many different functions when you write python code. You will also *write* many functions, but we will discuss that in a future lecture.

If you want to find out what a function does, the Python REPL has a rudimentary help function that you can use:

```
>>> help print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
(END)
```

To exit the help screen, press the letter '**`q`**'.

The help for **`print`** has some more advanced information that what we need right now, but there are a couple of interesting elements (see next slide).

Lecture 2: Functions

Although we pass in *arguments*, when they are listed in a function definition, they are actually called the function's *parameters*. You should get used to both terms. Let's look at some of the **`print`** function's parameters:

```
>>> help print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
(END)
```

If a function has a parameter "`...`", it means that you can have as many of the previous parameter. In this case, you can have many values, e.g.,

```
>>> print("this","that","more","many","more")
this that more many more
>>>
```

Notice that the values are separated by a space: see the **`sep`** parameter in the definition!

Lecture 2: Functions

```
>>> help print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
(END)
```

The **`sep`** parameter is the "string inserted between values, default a space." With so-called "named" parameters, we need to name them when we print. So, we could do the following:

```
>>> print("these","values","are","separated","by","a", "letter", sep='X')
theseXvaluesXareXseparatedXbyXaXletter
```

We changed the separator to "**`X`**" in this case. You can also change the end. If we want the **`print`** function to not put a new line after it runs, we can use **`end=''`**:

```
print("This line does not have a newline.", end='')
print("See? This line follows directly after the previous one.")
```

```
This line does not have a newline.See? This line follows directly after the previous one.
```

Lecture 2: Aside: quotes in quotes and ”escaping” a character

Note: in Python, you can interchange the double and single quotes for a string, but the beginning and end have to be the same. The following are equivalent:

```
>>> print("This is using double quotes")
This is using double quotes
>>> print('This is using single quotes')
This is using single quotes
```

This ability makes it easy to embed one type of quote inside a string:

```
>>> print("You won't believe it, but I can use a single quote in a string")
You won't believe it, but I can use a single quote in a string
>>> print('The teacher said, "You can use double quotes in strings"')
The teacher said, "You can use double quotes in strings"
```

Another way to embed a particular type of quote inside the *same* type of quote is to *escape* the quote character, by putting a *backslash* in front of it:

```
>>> print('You won\'t believe it, but I can use a single quote in a string')
You won't believe it, but I can use a single quote in a string
>>> print("The teacher said, \"You can use double quotes in strings\"")
The teacher said, "You can use double quotes in strings"
```

We will see other uses for escaping characters, soon.

Lecture 2: Functions from libraries (also called ”modules”)

One of the reasons that Python has become such a popular language is because it "comes with batteries included." This means that there are a lot of things included with Python that aren't directly included in other languages. Most of these extras come in the form of a *library (or "module")*, which is a set of related functions that you use with "dot notation" (which we have already briefly seen with the **`format`** function). To use a library, you *import* it into your program. One library that includes many useful functions is the **`math`** library. Let's see how it works:

```
>>> import math
>>> math.sqrt(25)
5.0
```

So, the **`math`** library has a square root function, that takes a number (integer or floating point) and produces (or *returns* -- more on this later) the square root of the number.

If you want to see the other functions in the **`math`** library, type **`dir(math)`**:

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Lecture 2: The math library

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

The **`dir`** function produces a list of attributes and functions in a library. Some of the items listed above are not functions. You can use the **`type`** function to determine what an object is:

```
>>> type(math.cos)
<class 'builtin_function_or_method'>
>>> type(math.pi)
<class 'float'>
>>> math.pi
3.141592653589793
>>> type(math.floor)
<class 'builtin_function_or_method'>
>>> type(math.isnan)
<class 'builtin_function_or_method'>
```

Some of those functions sound weird -- **`math.isnan`**? What is that?

Lecture 2: The math library

Some of those functions sound weird -- **`math.isnan`**? What is that? Let's find out:

```
help(math.isnan)

Help on built-in function isnan in module math:

isnan(x, /)
    Return True if x is a NaN (not a number), and False otherwise.
(END)
```

Ah -- it is "not a number," or "NaN". This is for cases where you have a number that isn't mathematical, like dividing a 0 by 0 (though that produces an error in Python). Most of the time, numbers are not NaN:

```
>>> math.isnan(42)
False
```

As with the **`print`** function, some **`math`** library functions have multiple parameters:

```
>>> help(math.gcd)

Help on built-in function gcd in module math:

gcd(x, y, /)
    greatest common divisor of x and y

>>> math.gcd(48,60)
12
```

Lecture 2: Different ways to import library functions

If you only need some functions from a library, you can import them directly, without the need for dot notation:

```
>>> from math import pi
>>> pi
3.141592653589793
```

You should be careful with this type of import, as there might already be another variable called **`pi`**, and this would change it.

If you want to import all functions from a library, you can do the following and not have to use dot notation, but it is generally frowned upon because there are so many names that could be changed:

```
>>> from math import *
>>> pi
3.141592653589793
>>> sqrt(100)
10.0
```

In this case, **`*`** is a *wildcard* character, which means "all" (this is a common use for the asterisk character).

Lecture 2: Variables and function return values

Some functions do not *return* any values. We call these functions *void* functions. The **`print`** function is such a function. If you attempt to set a value to the return type of a void function, it becomes *None*, which means that it has no value:

```
>>> a = print("Hello!")
Hello!
>>> print(a)
None
>>> type(a)
<class 'NoneType'>
```

A value with a type of *None* is different than an undefined variable:

```
>>> a = None
>>> type(a)
<class 'NoneType'>
>>> type(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Lecture 2: Variables and function return values

For functions that return something, technically they can only return a *single* value (we will soon see how to seemingly return multiple values). We can set a variable to the return value of the function:

```
>>> a = math.sqrt(200)
>>> print(a)
14.142135623730951
```

Then, of course, we can use the variable in other expressions:

```
>>> print(a / 2)
7.0710678118654755
>>> print("The number returned was {}".format(a))
The number returned was 14.142135623730951
```

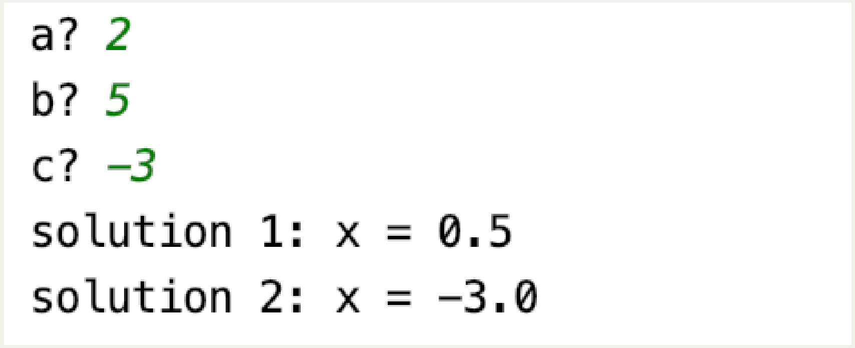
Lecture 2: Your turn: the quadratic formula

From your high school math days, you may remember the *quadratic formula* for solving equations of the form: $ax^2 + bx + c = 0$

There are two solutions for *x*, as defined by the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

With a partner, use one of your computers to create a Python program in PyCharm that requests *a*, *b*, and *c* from the user and then prints the two solutions to the associated quadratic equation, as follows:



```
a? 2
b? 5
c? -3
solution 1: x = 0.5
solution 2: x = -3.0
```

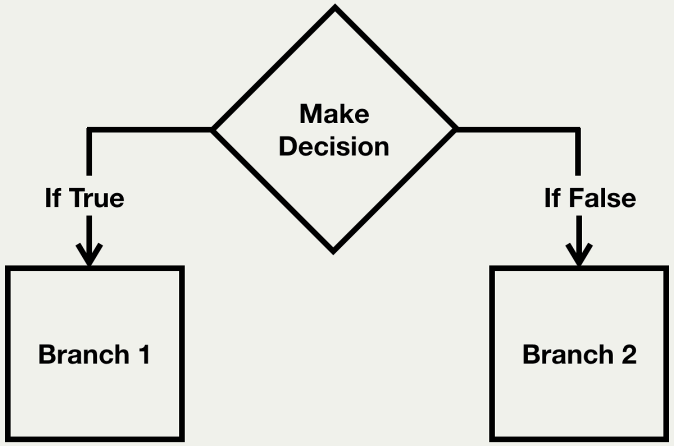
We will all go through the setup of a new project together, to remind everyone how to do it.

Hint: don't forget all the parentheses you need to make the calculation correct!

Lecture 2: Branching

So far, our programs have been pretty boring. ☹️ One line follows the next, and that's it. We have not yet written any programs that make *decisions*, and yet this is a critical and important thing that computers can do!

We want to use *logic* to decide what to do next in a program:



For example, let's say we ask the user to type an integer, and then we do one thing if the number is less than 50, another thing if the number is greater than or equal to 50. Let's write this program!

Lecture 2: Branching

```python
num = int(input("Please type an integer: "))
if num < 50:
    print("Your number was less than 50.")
else:
    print("Your number was greater than or equal to 50.")
```

Example run:

```
Please type a number between 1 and 100: 85
Your number was greater than 50.
```

This is an example of *branching*, where the program does not follow one line after the other, but instead makes a logical decision that takes the program in one direction or another. In the program above, there were three decisions that we checked: less than 50, equal to 50, or greater than 50.

One way to make a program branch is to use an **`if`** statement, as we did above. We will describe the **`if`** statement on the next slide.

Lecture 2: Branching and the `if` statement

The **`if`** statement is defined as follows:

```
if expression:
   statement(s)
else:
   statement(s)
```

The **`if`** statement does not need to have an else statement, and in its most basic form, it is simply this:

```
if expression:
   statement(s) # if expression is true, execute the lines in this block
# no matter what, keep executing after this line
```

A *block* is an indented section of code that either is executed or not, depending on the result of the *expression* (more on this in a moment). All lines in a block must be indented exactly the same. In the following example, the three lines in the block will be executed if **`num`** is greater than 100. If the number is less than 100, none of the three lines in the block will be executed, and the code would go to line 6 immediately:

```
1 num = int(input("Please type an integer: "))
2 if num > 100:
3     print("This line will get printed if the number is greater than 100")
4     print("This line will also get printed")
5     print("And so will this line")
6 print("This will always get printed")
```

Lecture 2: Branching and boolean expressions

The **`if`** statement is defined as follows:

```
if expression:
   statement(s)
else:
   statement(s)
```

The *expression* in the **`if`** statement must evaluate to a *boolean* value. Boolean values are either **`True`** or **`False`**. You can compare any Python objects (numbers, strings, etc.) with the following comparison operators:

```
Op     Meaning            Example producing True
<      less than          6 < 7
<=     less than          5 <= 5
         or equal to
>      greater than       8 > 3
>=     greater than       8 >= 3
         or equal to
==     equal to           7 == 7
!=     not equal to       7 != 8
```

Strings can also be compared, with all uppercase letters always less than all lowercase letters, and letters earlier in the alphabet less than letters later in the alphabet:

```
>>> "cat" < "dog"
True
>>> "cat" < "Dog"
False
>>> "Cat" < "Dog"
True
```

Lecture 2: Branching and boolean expressions

Two boolean operators you may not have seen before are **==** and **!=**

```
Op    Meaning          Example producing True
==    equal to         7 == 7
!=    not equal to     7 != 8
```

The double-equals sign (**==**) compares the value on the left to the value on the right (like the other operators). A single equals sign *does not* mean the same thing, and a single equals sign is reserved to assign a value to a variable (as we have seen before):

```
>>> a = 5
>>> b = 6
>>> if a = b:
  File "<stdin>", line 1
    if a = b:
         ^
SyntaxError: invalid syntax
```

You may make this mistake many times before you get used to it, but Python (unlike many other languages) will give you a syntax error if you try it.

Lecture 2: The else and elif clauses

```
if expression:
   statement(s)
else:
   statement(s)
```

The **`else`** clause in an **`if`** statement does the "other" thing when the **`if`** expression is false. We saw this in the earlier example:

```
num = int(input("Please type an integer: "))
if num < 50:
    print("Your number was less than 50.")
else:
    print("Your number was greater than or equal to 50.")
```

```
Please type an integer: 12
Your number was less than 50.
```

```
Please type an integer: 83
Your number was greater than or equal to 50.
```

If you want to add a condition to your **`if`** statement, you can use the **`elif`** keyword, which is shorthand for "else if" and means "if the first expression was false, check this one, too":

```
num = int(input("Please type an integer: "))
if num < 50:
    print("Your number was less than 50.")
elif num == 50:
        print("Your number was exactly 50.")
else:
    print("Your number was greater than 50.")
```

You can put as many **`elif`**s in as you want, for as many cases as you have to check.

Lecture 2: Practice with boolean expressions

Let's practice booleans by writing a program using boolean expressions. Let's (together) write a game that asks the player of the game for a number between 5 and 20. Then, the computer will choose a random number (we'll see how) between 0 and the number the user chose. Next, the computer will ask the player three times to try and guess the number. If the player guesses too low, the computer will tell the player that, and then will ask again. Likewise if the player guesses too high, the computer will say that it is too high and ask again. If the player guesses the number correctly, then the game ends. If the player doesn't guess correctly after three tries, the game ends and the computer tells the player the number that it chose.

To start, let's talk about random numbers. There is a **`random`** library that allows the computer to seemingly choose random numbers. They are actually *pseudorandom* numbers, because they are generated with an algorithm, but it is almost impossible to tell that they aren't random. We care about the **`random.randint`** function for this program:

```
>>> import random
>>> dir(random.randint)

Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
(END)
```

Lecture 2: Practice with boolean expressions

Let's start the program by asking the user for a number, and then getting a random number between 0 and that number:

```
import random

maximum = int(input("Let's play a game. Choose a number between 5 and 20: "))

computer_choice = random.randint(0,maximum)
```

Let's actually make this a bit more advanced. First, let's set up a *constant* that will be the maximum number. We like to make constant values (ones that won't change) **`UPPERCASE`** so someone reading our code knows that number won't change:

```
import random

MAX_NUM = 20

maximum = int(input("Let's play a game. Choose a number between 5 and {}: ".format(MAX_NUM)))

computer_choice = random.randint(0,maximum)
```

Now, if we want to change the maximum number, we know where can do it -- right at the beginning.

For this class: any time you have a "special" number, make it a constant!

Lecture 2: Practice with boolean expressions

```python
import random

MAX_NUM = 20

maximum = int(input("Let's play a game. Choose a number between 5 and {}: ".format(MAX_NUM))

computer_choice = random.randint(0,maximum)
```

Now that we have our number from the user, let's use some boolean logic to see if the player followed directions:

```python
import random

MAX_NUM = 20

maximum = int(input("Let's play a game. Choose a number between 5 and {}: ".format(MAX_NUM))

if maximum > MAX_NUM:
    print("Sorry, that number is too high. Goodbye!")
    quit()
elif maximum < 0:
    print("Sorry, that number is too low. Goodbye!")
    quit()

computer_choice = random.randint(0,maximum)
```

Now we know that we have a number in the correct range.

Lecture 2: Practice with boolean expressions

```python
import random

MAX_NUM = 20

maximum = int(input("Let's play a game. Choose a number between 5 and {}: ".format(MAX_NUM)))

if maximum > MAX_NUM:
    print("Sorry, that number is too high. Goodbye!")
    quit()
elif maximum < 0:
    print("Sorry, that number is too low. Goodbye!")
    quit()

computer_choice = random.randint(0,maximum)
```

At this point, we can start asking the player for their guesses. We can use branching to give the user feedback:

```python
print("I have chosen a number between 0 and {}, inclusive.".format(maximum))
guess = int(input("Try to guess my number. You have 3 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()
```

Lecture 2: Practice with boolean expressions

```
print("I have chosen a number between 0 and {}, inclusive.".format(maximum))
guess = int(input("Try to guess my number. You have 3 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()
```

Note: the **`quit()`** function does what you may think -- it ends the program.

At this point, if the player guesses incorrectly, they get a message that says whether they guessed too low or too high. What do we do next? We can keep asking:

```
guess = int(input("Try to guess my number. You have 2 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()
```

Hmm -- this seems redundant. We will learn how to fix the redundancy shortly!

Lecture 2: Practice with boolean expressions

```
guess = int(input("Try to guess my number. You have 2 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()
```

We seem to need more redundancy (though it is a bit different) to continue (and finish) the game:

```
guess = int(input("Try to guess my number. You have 1 try left. "))

if guess < computer_choice:
    print("Too low! You lose!")
elif guess > computer_choice:
    print("Too high! You lose!")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()

print("The number I chose was {}.".format(computer_choice))
```

Lecture 2: Full game – first attempt:

```python
import random

MAX_NUM = 20

maximum = int(input("Let's play a game. Choose a number between 5 and {}: ".format(MAX_NUM)

if maximum > MAX_NUM:
    print("Sorry, that number is too high. Goodbye!")
    quit()
elif maximum < 0:
    print("Sorry, that number is too low. Goodbye!")
    quit()

computer_choice = random.randint(0,maximum)

print("I have chosen a number between 0 and {}, inclusive.".format(maximum))
guess = int(input("Try to guess my number. You have 3 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()

guess = int(input("Try to guess my number. You have 2 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()

guess = int(input("Try to guess my number. You have 1 try left. "))

if guess < computer_choice:
    print("Too low! You lose!")
elif guess > computer_choice:
    print("Too high! You lose!.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()

print("The number I chose was {}.".format(computer_choice))
```

This program works, and does what we want it to, but it is pretty long, and has a lot of redundancy.

When you find redundancy in your programs, you should immediately be thinking, "I can write a function to fix this!" Remember, functions are *reusable* pieces of code that can be used multiple times.

Let's turn our attention to writing our own functions.

Lecture 2: Writing our own functions

The following is the outline for a function definition:

```
def function_name(paramater1, parameter2, ...):
    statement_1
    statement_2
    ...
    return ...
```

Functions contain blocks (like `if` statements) that must be indented to the same level. They may or may not `return` a value, and they may or may not have parameters. Here is a simple function:

```
def printName(name):
    print("The name to print is {}".format(name))
```

You can define a function in the REPL, though you can't modify the function once you create it. The "`...`" defines the block indentation for you:

```
>>> def printName(name):
...     print("The name to print is {}".format(name))
...
>>> printName("Chris")
The name to print is Chris
>>> printName("Yiya")
The name to print is Yiya
```

Now we have our own function that we can use over and over again.

Lecture 2: Writing our own functions

Here is another simple function that returns a value:

```
def cube(x):
    '''cubes x and returns the result'''
    return x * x * x
```

In the REPL:

```
>>> def cube(x):
...     '''cubes x and returns the result'''
...     return x * x * x
...
>>> cube(4)
64
>>> cube(2)
8
>>> cube(85)
614125
>>> a = cube(5)
>>> print(a)
125
```

Note that we have a *docstring* at the beginning of our function. We can now use **`help()`** on our function!

```
>>> help(cube)

Help on function cube in module __main__:

cube(x)
    cubes x and returns the result
(END)
```

This is also helpful for anyone reading the code we write -- they can also use the **`help`** function to see what our functions do.

Lecture 2: Testing our own functions

One of the most important things a programmer needs to be able to do is to *test* their code.

```
def cube(x):
    '''cubes x and returns the result'''
    return x * x * x
```

How could we test to see if our **`cube`** function works?

Well, one thing we could do is to just test it with numbers we know. We did that in the REPL:

```
>>> def cube(x):
...     '''cubes x and returns the result'''
...     return x * x * x
...
>>> cube(4)
64
>>> cube(2)
8
>>> cube(85)
614125
>>> a = cube(5)
>>> print(a)
125
```

Python actually provides a way to do the tests *automatically* for you, so that if you change the code, you can ensure that it still works. This will become more important as your functions get larger and more complex, but we can use it in our functions. We want you to use it in all of your functions, as well.

Lecture 2: The `doctest` module

To test your code automatically, you add tests to your docstring, based on how you would test your code in the REPL.

```python
def cube(x):
    '''cubes x and returns the result'''
    return x * x * x
```

☞ This is the original.

```python
def cube(x):
     '''cubes x and returns the result
     >>> cube(4)
     64

     >>> cube(2)
     8

     >>> cube(85)
     614125

     >>> a = cube(5)
     >>> print(a)
     125
     '''
     return x * x * x
```

☞ This is the code with some tests. Notice that they are *identical* to what would happen in the REPL.

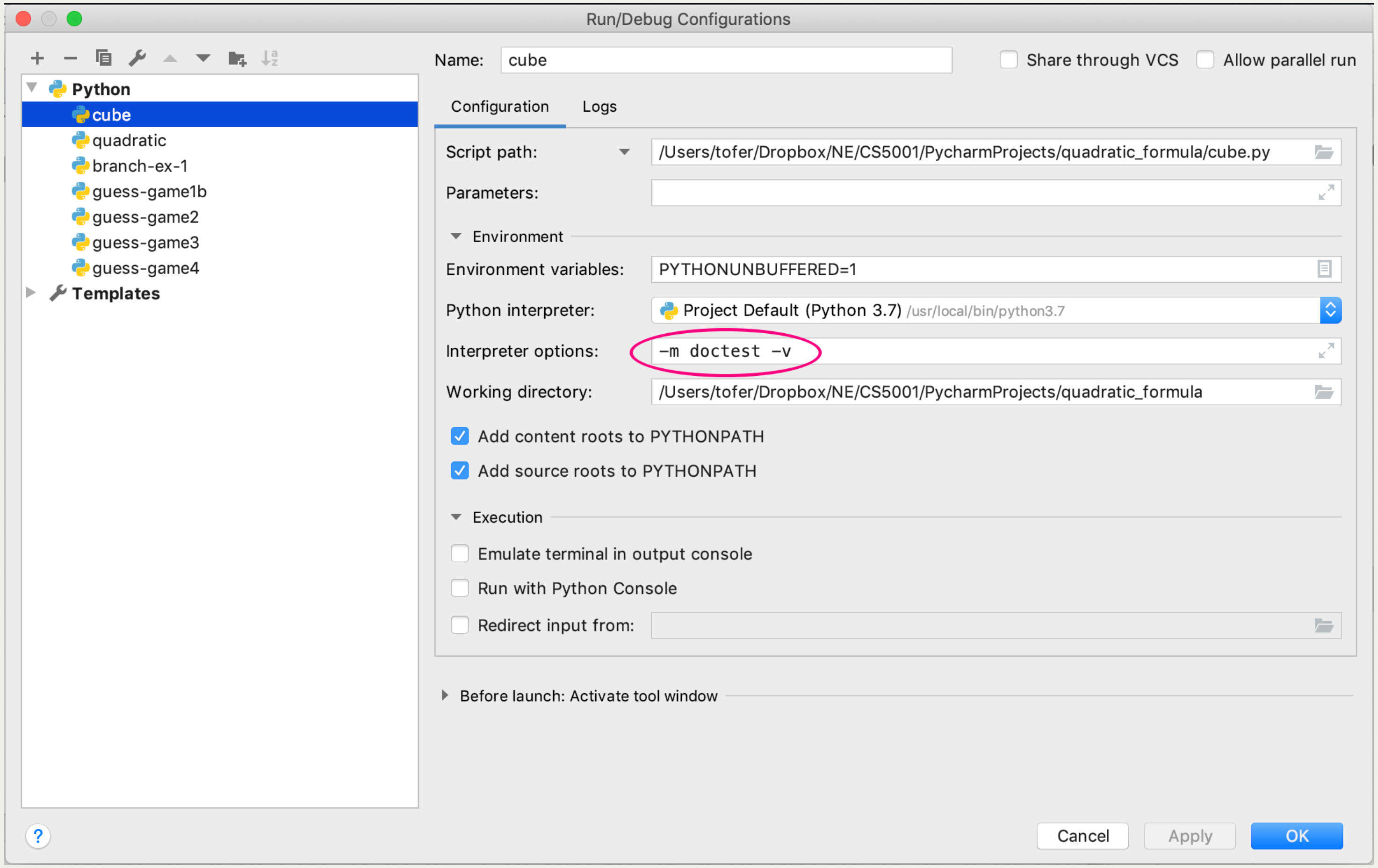
All of the tests are in the docstring, and not only do they show more information about the function, they also allow us to test them.

Lecture 2: The `doctest` module

To test your functions in PyCharm, you can change the "Interpreter options" to

-m doctest -v

and this will run the tests when you run the program (see next slide).



Name: cube
Share through VCS [ ]
Allow parallel run [ ]

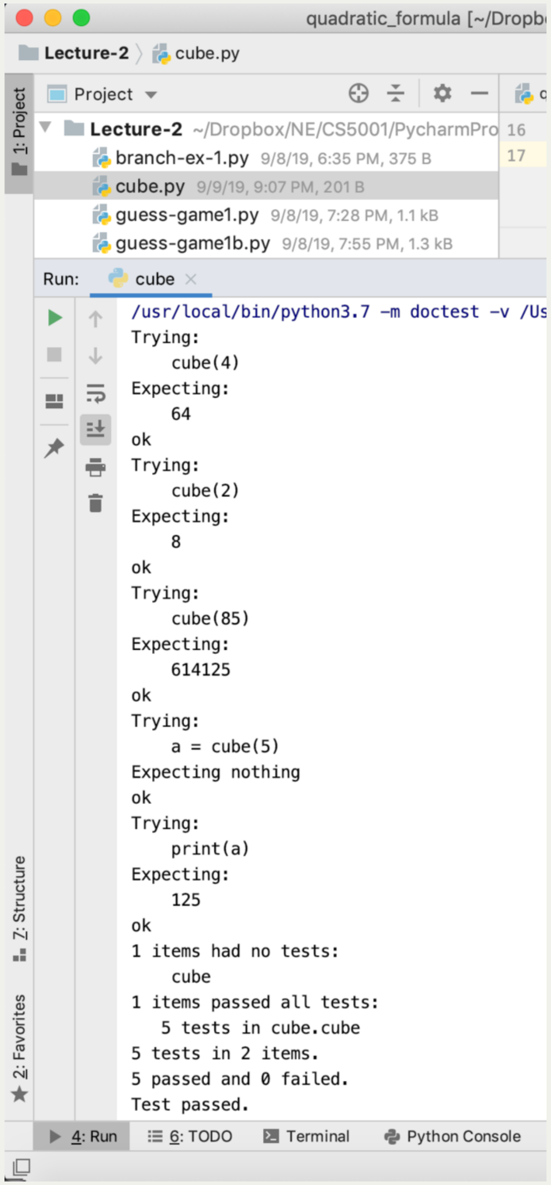
Script path: /Users/tofer/Dropbox/NE/CS5001/PycharmProjects/quadratic_formula/cube.py
Parameters:
Environment variables: PYTHONUNBUFFERED=1
Python interpreter: Project Default (Python 3.7) /usr/local/bin/python3.7
Interpreter options: `-m doctest -v`
Working directory: /Users/tofer/Dropbox/NE/CS5001/PycharmProjects/quadratic_formula

[✓] Add content roots to PYTHONPATH
[✓] Add source roots to PYTHONPATH

[ ] Emulate terminal in output console
[ ] Run with Python Console
[ ] Redirect input from:

Before launch: Activate tool window

Lecture 2: The `doctest` module



quadratic_formula [~/Dropb
Lecture-2 〉 cube.py
1: Project
Project
Lecture-2 ~/Dropbox/NE/CS5001/PycharmPro
branch-ex-1.py 9/8/19, 6:35 PM, 375 B
cube.py 9/9/19, 9:07 PM, 201 B
guess-game1.py 9/8/19, 7:28 PM, 1.1 kB
guess-game1b.py 9/8/19, 7:55 PM, 1.3 kB
Run: cube
7: Structure
2: Favorites
4: Run
6: TODO
Terminal
Python Console

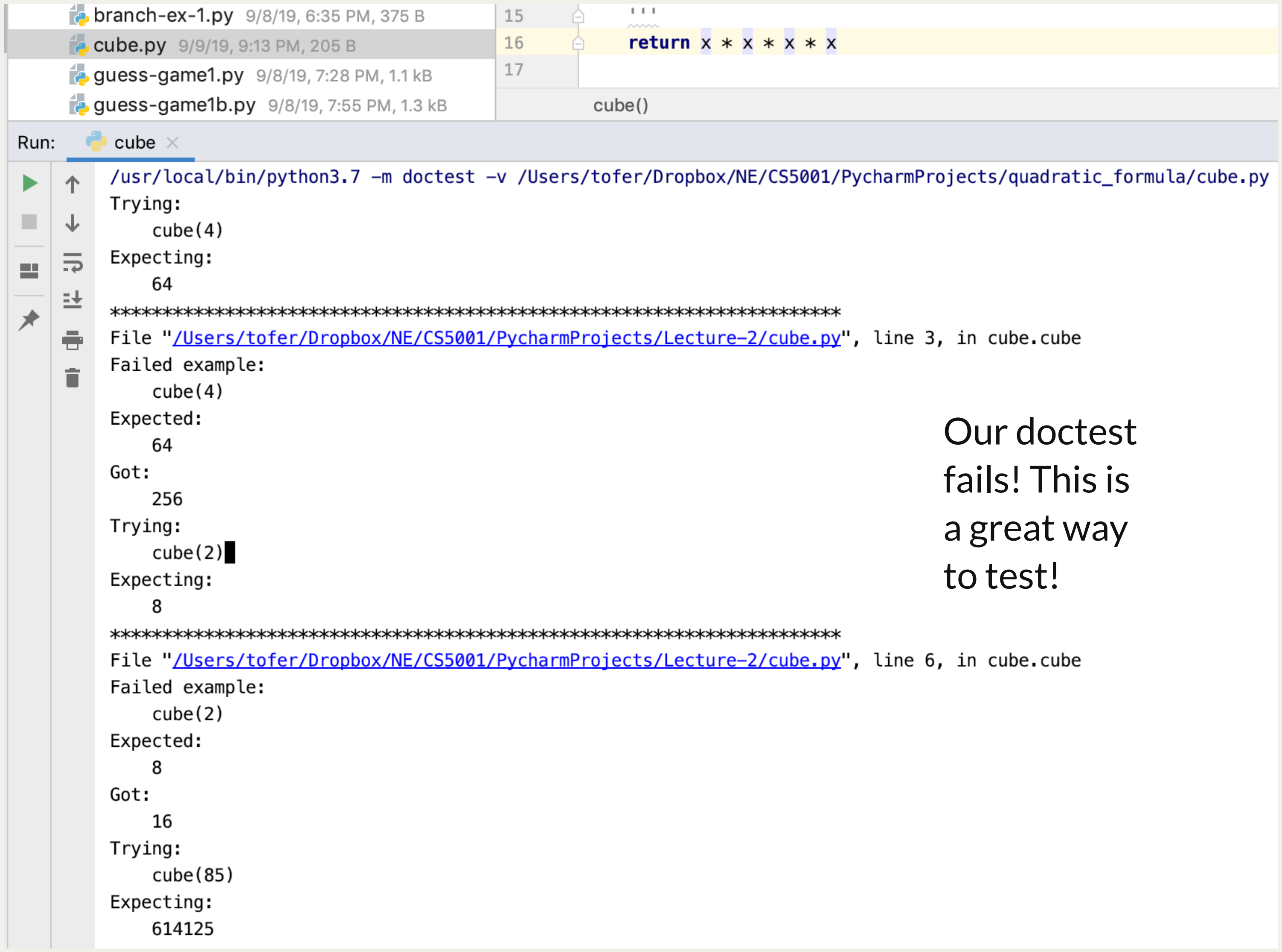
```
/usr/local/bin/python3.7 -m doctest -v /Us
Trying:
    cube(4)
Expecting:
    64
ok
Trying:
    cube(2)
Expecting:
    8
ok
Trying:
    cube(85)
Expecting:
    614125
ok
Trying:
    a = cube(5)
Expecting nothing
ok
Trying:
    print(a)
Expecting:
    125
ok
1 items had no tests:
    cube
1 items passed all tests:
   5 tests in cube.cube
5 tests in 2 items.
5 passed and 0 failed.
Test passed.
```

Notice that all of the tests came out "ok". There are as many tests as there are "**>>>**" lines in the code itself.

```
def cube(x):
     '''cubes x and returns the result
     >>> cube(4)
     64

     >>> cube(2)
     8

     >>> cube(85)
     614125

     >>> a = cube(5)
     >>> print(a)
     125
     '''
     return x * x * x
```

Lecture 2: The `doctest` module

Let's say we made a mistake in our code, and multiplied by x four times instead of three:



```
branch-ex-1.py 9/8/19, 6:35 PM, 375 B
cube.py 9/9/19, 9:13 PM, 205 B
guess-game1.py 9/8/19, 7:28 PM, 1.1 kB
guess-game1b.py 9/8/19, 7:55 PM, 1.3 kB
```

```
    '''
    return x * x * x * x

cube()
```

Run: cube

```
/usr/local/bin/python3.7 -m doctest -v /Users/tofer/Dropbox/NE/CS5001/PycharmProjects/quadratic_formula/cube.py
Trying:
    cube(4)
Expecting:
    64
**********************************************************************
File "/Users/tofer/Dropbox/NE/CS5001/PycharmProjects/Lecture-2/cube.py", line 3, in cube.cube
Failed example:
    cube(4)
Expected:
    64
Got:
    256
Trying:
    cube(2)
Expecting:
    8
**********************************************************************
File "/Users/tofer/Dropbox/NE/CS5001/PycharmProjects/Lecture-2/cube.py", line 6, in cube.cube
Failed example:
    cube(2)
Expected:
    8
Got:
    16
Trying:
    cube(85)
Expecting:
    614125
```

Our doctest fails! This is a great way to test!

Lecture 2: `if __name__ == '__main__'`

We need to do one more thing to ensure that our tests run correctly. From now on, we will always put the code that we want to run immediately (i.e., outside of functions) inside of the following conditional:

```python
if __name__ == '__main__':
  # our code goes here
  print("Hello, World!")
```

When you run a program, it has a variable called `__name__` with the value of `'__main__'`. So, this checks to ensure that we are running our program as a normal program.

When we run using the `doctest` module, `__name__` is *not* called `'__main__'`, so it doesn't get run (which is what we want).

```python
def cube(x):
    '''cubes x and returns the result
    >>> cube(4)
    64

    >>> cube(2)
    8
    '''
    return x * x * x

if __name__ == '__main__':
    value = int(input("What number would you like me to cube? "))
    print(cube(value))
```

Lecture 2: The `doctest` module

We will expect that all functions you write for this class have at least two different doctests in each function.

We will grade your assignments accordingly.

The only exceptions are for functions that take input -- you can't really test this because you don't know what input you will get. But--often, you can re-write your functions so they are *pure*, meaning that they don't take input.

Before you hand in your assignments, always run the doctests, and make sure that they all pass.

Testing code is one of the most critical parts of programming! If you start running tests early in your programming career, you will be a much better programmer throughout your career.

Dad, please help me! I've been debugging this code for days!
Which unittest fails?
I don't have tests...

Lecture 2: Writing a function for our guessing game

Earlier, we wrote a lot of code for part of our guessing game:

```python
guess = int(input("Try to guess my number. You have 3 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()

guess = int(input("Try to guess my number. You have 2 tries left. "))

if guess < computer_choice:
    print("Too low! Guess again.")
elif guess > computer_choice:
    print("Too high! Guess again.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()

guess = int(input("Try to guess my number. You have 1 try left. "))

if guess < computer_choice:
    print("Too low! You lose!")
elif guess > computer_choice:
    print("Too high! You lose!.")
else:
    print("You guessed my number!")
    print("Goodbye!")
    quit()
```

Because each of the three guesses are (basically) the same, we should be able to write a function to do it, and we could save a lot of code!

Let's write a function to determine if the guess was too low, too high, or correct. We will leave out the input (not easy to doctest, remember?)

This should save a lot of code.

Lecture 2: Writing a function for our guessing game

Things to think about when writing a function:

- What will the function be named (this is more important than you think! You want your function to have a meaningful name.
- What will the function do? (a good function does one thing)
- What will the parameters be?
- What will the function return, if anything?

Let's answer the questions for our function:

- What will the function be named (this is more important than you think! You want your function to have a meaningful name.
    - Because we are trying to evaluate the guess, a good name might be **`evaluate_guess`**
- What will the function do? (a good function does one thing)
    - If the guess is too low, print("Too low! Guess again.")
    - If the guess is too high, print("Too high! Guess again.")
    - If the guess is correct, print("You guessed my number!") and then "Goodbye!"
- What will the parameters be?
    - **`computer_choice`** and **`guess`**
- What will the function return, if anything?
    - -1 for too low
    - 0 for correct
    - 1 for too high

This is a typical strategy for too low / too high / just right answers

Lecture 2: Writing a function for our guessing game

Why are we returning a value from the function?

- We will want to use the result from the function to make other choices.

Here is our function:

```python
def evaluate_guess(computer_choice, guess):
    '''Checks the computer choice against the guess and ends game
       if the guess is equal to computer_choice.
       Also tells player whether the guess was too high or too low
       Return value: -1 for too low, 0 for correct, 1 for too high
    '''
    if guess < computer_choice:
        print("Too low! Guess again.")
        return -1
    elif guess > computer_choice:
        print("Too high! Guess again.")
        return 1
    else:
        print("You guessed my number!")
        print("Goodbye!")
        return 0
```

What do we still need to do? doctest!

Lecture 2: Writing a function for our guessing game

We should write a doctest for each different possibility:

```python
def evaluate_guess(computer_choice, guess):
    '''Checks the computer choice against the guess and ends game
       if the guess is equal to computer_choice.
       Also tells player whether the guess was too high or too low
       Return value: -1 for too low, 0 for correct, 1 for too high

       >>> evaluate_guess(8,3)
       Too low! Guess again.
       -1

       >>> evaluate_guess(5,9)
       Too high! Guess again.
       1

       >>> evaluate_guess(20,20)
       You guessed my number!
       Goodbye!
       0
    '''
    if guess < computer_choice:
        print("Too low! Guess again.")
        return -1
    elif guess > computer_choice:
        print("Too high! Guess again.")
        return 1
    else:
        print("You guessed my number!")
        print("Goodbye!")
        return 0
```

Lecture 2: Writing a function for our guessing game

Now we can update our original program to use our function:

```python
if __name__ == '__main__':
 # ... (see previous program)
 computer_choice = random.randint(0,maximum)

    print("I have chosen a number between 0 and {}, inclusive.".format(maximum))

    guess = int(input("Try to guess my number. You have 3 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    guess = int(input("Try to guess my number. You have 2 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    guess = int(input("Try to guess my number. You have 1 try left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    print("The number I chose was {}.".format(computer_choice))
```

Notice that we use the return value from our function to determine whether or not to quit. We could have put the quit inside the function, but you generally don't want to quit inside a function for stylistic reasons.

Lecture 2: Writing a function for our guessing game

It still seems like we have a lot of redundant code!

```python
    guess = int(input("Try to guess my number. You have 3 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    guess = int(input("Try to guess my number. You have 2 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    guess = int(input("Try to guess my number. You have 1 try left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()
```

Not only that, but what if we wanted to let the user guess more than three times. What if wanted the user to guess one hundred times?

We will find out how to do this next week!

Lecture 2: Wrap-up

What did we learn tonight?

- Functions
  - Passing values to functions
- Libraries (modules) and "batteries included" code
- Function return values
- Branching
  - The **`if`** statement
- Testing your code with the **`doctest`** module
- Writing our own functions