Problem Solving in Python

What is a Program

- A program is a set of instructions
- Not specific to computers
- Can be written at a very low level, a very high level, or anywhere in between

High Level Program:

```
go to kitchen
make pbj sandwich
```

Lower Level Program:

```
stand up from couch
walk to kitchen
open cabinet
get peanut butter jar
get jelly jar
close cabinet
open peanut butter jar
open jelly jar
…
```

Formal vs. Natural Languages

- Natural languages are the ones we speak
- Called this because we don't invent them, we *naturally* acquire them
- Formal languages we invent for a specific purpose
- Programming languages are formal (we invented them for computers)

Natural Languages:
English
Spanish
French

Formal Languages:
Math proofs
Mathematical logic
Programming languages

Python

The Python Language

- Python is a general purpose high level programming language
- It works on many different platforms (Windows, Linux, Mac, etc.) and is free to use
- It emphasizes readability and helps minimize the number of lines needed to write a program

Following The Examples

- When we give examples of commands or code, we put it in the **`Courier`** font

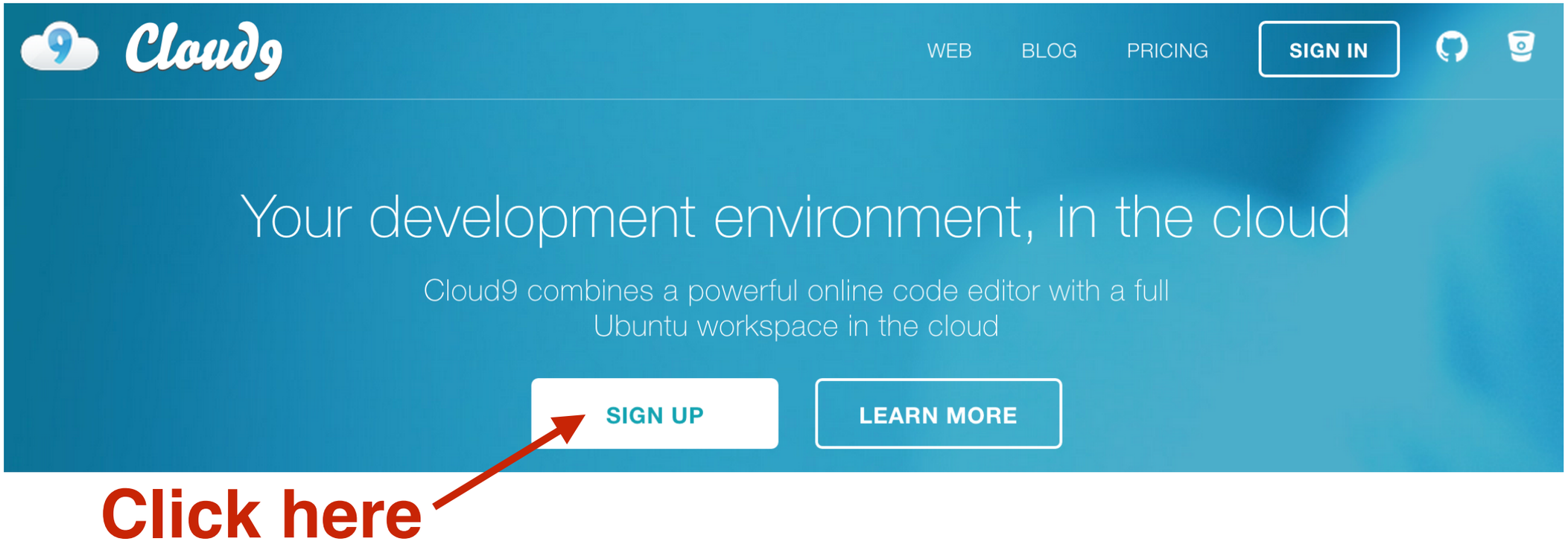
```
if divisor <= 0:
    print “Cannot divide by 0”
else:
    result = number / divisor
```

- Any examples in **`Courier`** can be followed in your Cloud9 workspace

- Now lets talk about setting up your Cloud9 workspace

- In your web browser, type c9.io into the address bar (and hit enter)

- Click the large “Sign Up” button in the center of the page



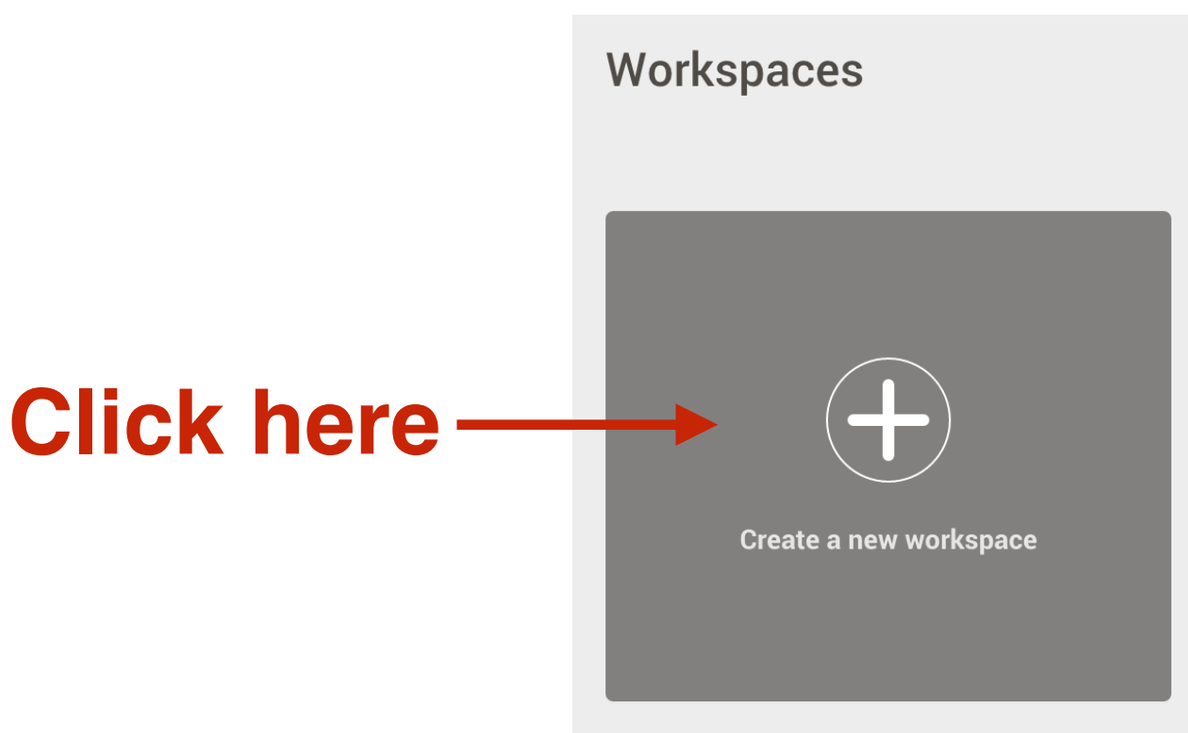
- Choose and enter your username, password, and email address, and then click “Create your free account”

- Now that you have signed in to your account (make sure to remember your username and password), now you need to create a workspace

Creating Your Workspace

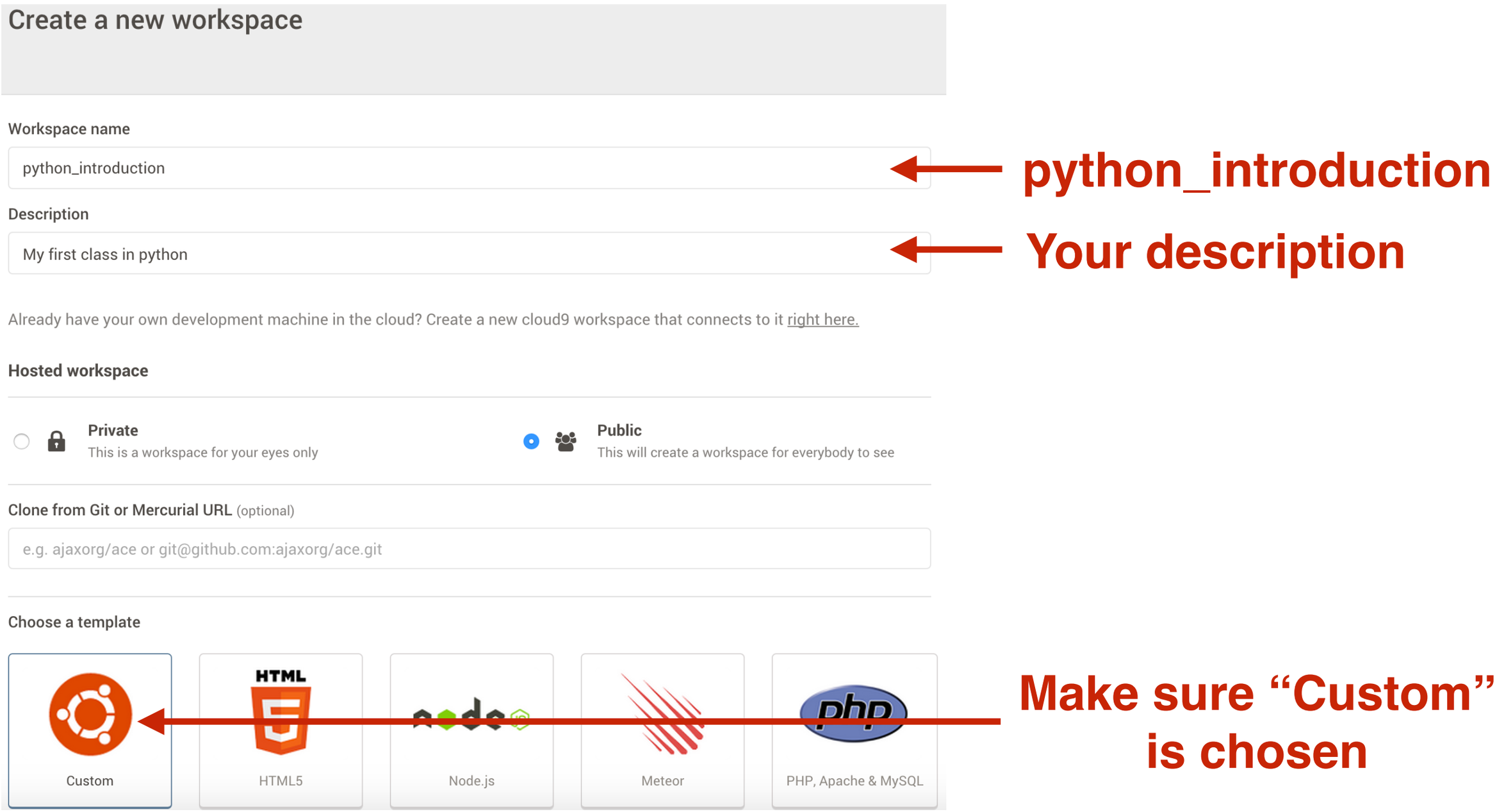
- Now that you have signed in to your account (make sure to remember your username and password), now you need to create a workspace

- In the center of the page, click the “Create a new workspace” button

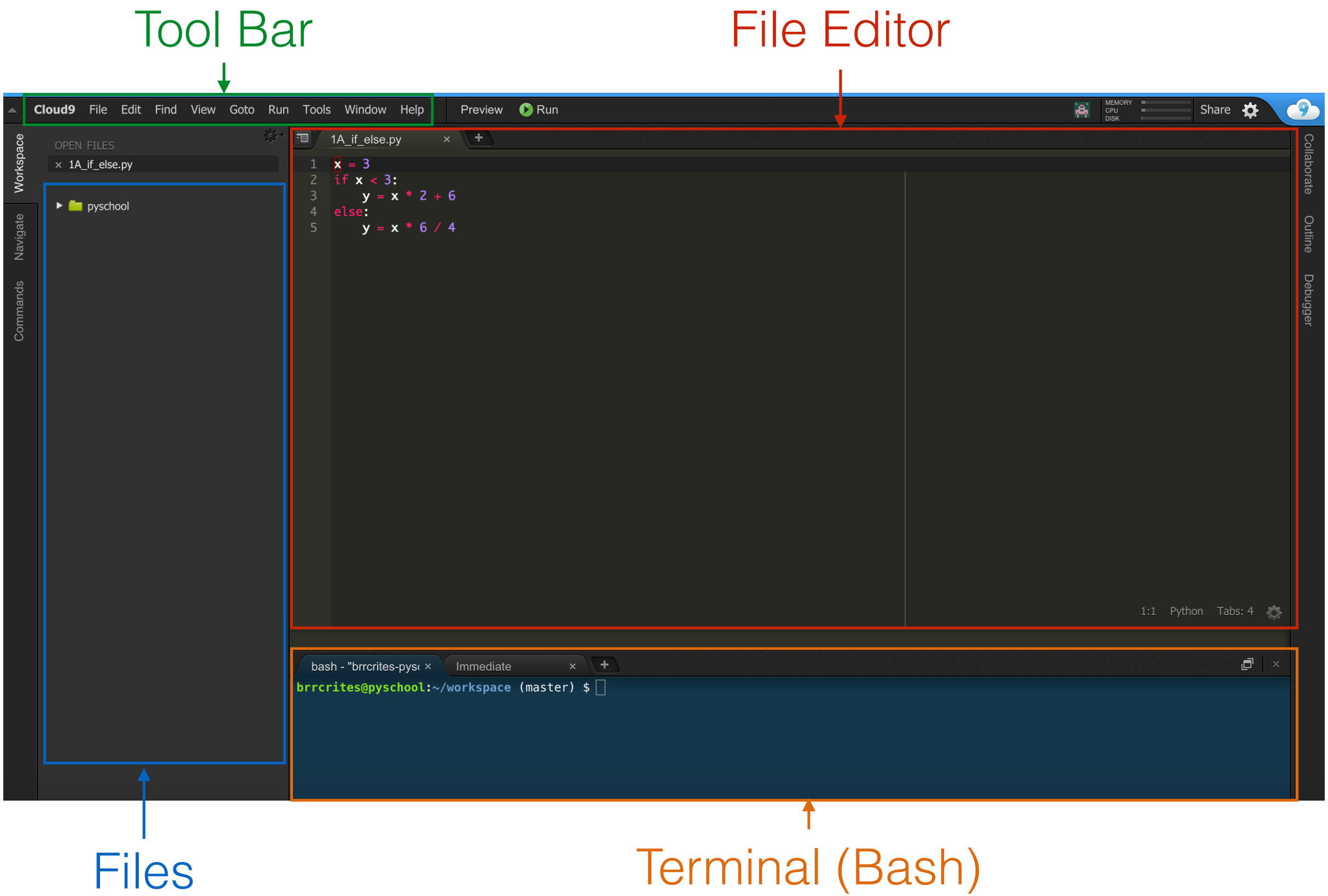


- Now, name your workspace “python_introduction”, and give it any description you would like.

- Make sure “Custom” is chosen as the template



- Now your workspace should automatically open, let's take a look at your new workspace

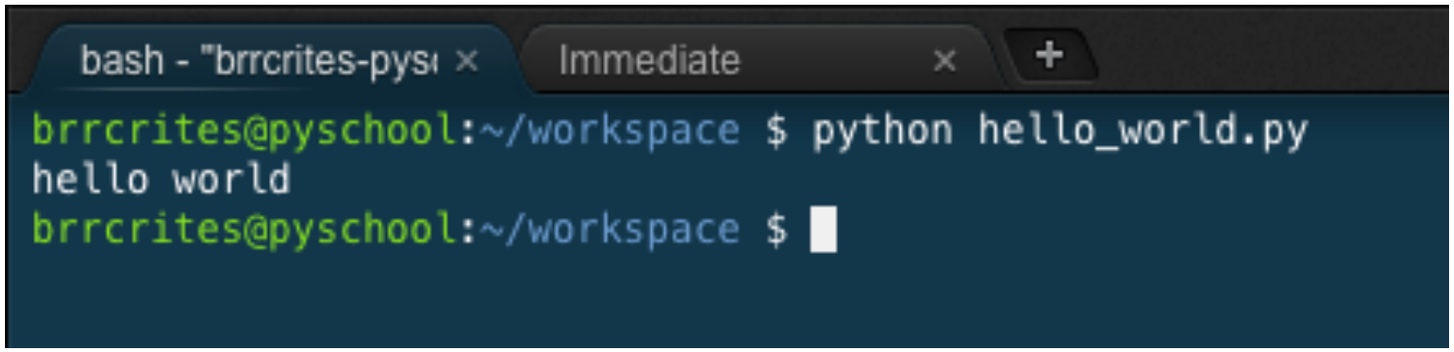


Your First Program

- In your Cloud9 workspace, go to File > New File
- Save your file by going to File > Save and save it as **`hello_world.py`** (remember python files have the .py extension)
- In your new file type the following (with the quotation marks)

```
print “hello world”
```

- Now save again with File > Save (the shortcut for this is ctrl +s)



- In order to run a script, go to your terminal (you should see the **$** symbol) and type “python” followed by the name of the script, then press enter

```
python hello_world.py
```

- You should see the output of your program after you press enter, and then another terminal

What Did We Write?

- So lets look at the first line

```
print “hello world”
```

- The word **`print`** is a keyword, keywords are words reserved by the language to perform special tasks

- In this case, it prints whatever comes after it to the terminal

What are other keywords?

```
and          or          except      finally
del          with        import      is
from         assert      print       return
not          else        class       def
while        if          exec        for
as           pass        in          lambda
elif         yield       raise       try
global       break       continue
```

- These are most of the keywords that are reserved in python

- These are the twelve keywords that you will learn in this class, including the one that you have already learned about

`and`	`in`	`else`	`def`
`not`	`if`	`while`	`return`
`or`	`elif`	`for`	**`print`**

So What Did We Print?

- Lets go back to the line of code we wrote

```
print “hello world”
```

- This prints out **`hello world`**

- **`S`**o what is **`“hello world”`**?

- In python, words surrounded by quotation marks on each end are called strings, so **`“hello world`**” is an example of a string

Strings

- Strings are one of the basic data types in python
- They are surrounded by either **`“double quotes”`** or **`‘single quotes’`**
- Strings can contain letters, numbers, spaces, and symbols
- The only exception is putting quotes into your string

- Since strings begin and end with quotes, you cannot just put quotes into the middle of a string (since that would end the string)

- If you want to have quotation marks that appear in the string when you print it, you must put a backslash before the quote

```
"Here is an example with \"quotes\" in it"
```

- If you printed the above string, you would have the following output

```
Here is an example with "quotes" in it
```

- Now lets try printing your name in addition to hello world

- Go back to your program file **`hello_world.py`** and type the following into your file (not your terminal) on the next line after **`print “hello world”`**

```
name = “Issac Newton”
```

- Don’t forget to include the quotation marks, and replace Issac Newton with your name

- Now add the print statement on the next line

```
print name
```

- **Make sure that you save**, File > Save (or ctrl+s)

- Your complete file should look like the following

```
print “hello world”
name = “Issac Newton”
print name
```

- Now run the script from the terminal as before

```
python hello_world.py
```

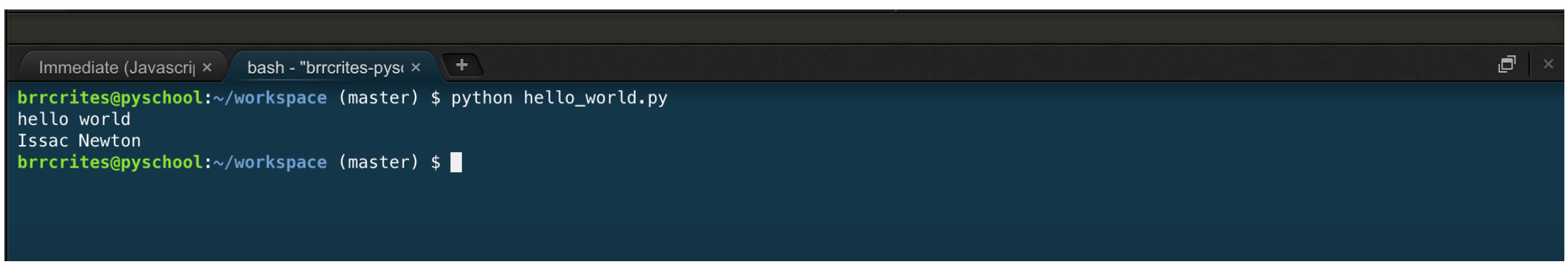
- Notice that your program first printed

```
"hello world"
```

- And then it printed your name

- This is because programs run in order from top to bottom

```
print "hello world"
name = "Issac Newton"
print name
```



So What Did We Do Now?

- We wrote another line

```
name = “Issac Newton”
```

- In this line, **`name`** is called a variable

Variables

- Variables are like labels we use to refer to something, like how we use names to refer to people, or URLs to refer to web pages
- Variables can be assigned, and re-assigned different values
- We assign and re-assign values to variables using the equals sign (=)

- This line assigns the string **`"Issac Newton"`** to the variable **`name`**, so **`name`** is a label for the string **`"Issac Newton"`**

Variable		Value
`name`	`=`	`"Issac Newton"`

- So now, whenever we want to use the string **`"Issac Newton"`**, we can use the variable **`name`** instead

- Remember before, when we printed a string

```
print “hello world”
```

- And then next we printed a variable (and that variable had a string as its value)

```
print name
```

- Notice name is a variable and not a string because it is not surrounded by quotation marks

- We can also print multiple things using the same print command by separating them with commas

```
print “hello”, name
```

- Commas in python tells the print keyword not to go to the next line before printing the next item

- So while having two prints would print each on a new line

code
```
print “hello”
print name
```

output
```
hello
Issac Newton
```

- Our example will print them both on the same line

code
```
print “hello”, name
```

output
```
hello Issac Newton
```

- This also works with two separate prints, with one ending with a comma

code
```
print “hello”,
print name
```

output
```
hello Issac Newton
```

Naming Variables

- Variables should be named after what they represent

```
pi = 3.14
```

- Variable names must start with a letter, and can only contain letters, numbers, and underscores, they also cannot be keywords (that list of words python reserves that we talked about earlier)

- Variables names cannot contain spaces, so for variables who's names should have multiple words, we usually use an underscore in place of a space

```
multiple_word_variable = “some words”
```

Integers and Floating Points

- Notice that the variable in the last example was not a string, it was a number

- In python, there are two other basic data types

- Integers, which are whole numbers

```
4, 8, 15, 16, 23, 42
```

- and floating point numbers, which are decimal numbers

```
3.14, 2.718, 0.57721
```

Variable Types

- In Python, the type of a variable is determined automatically

`pi = “pi”`	String
`pi = 3`	Integer
`pi = 3.14`	Floating Point

- We can assign and re-assign different types of values to any variable

- So if we had the following code, x would have different values at different points in the code

Execution Order

```
pi = “pi”
print pi
pi = 3
print pi
pi = 3.14
print pi
```

```
pi = “pi”

pi = 3

pi = 3.14
```

```
Immediate (Javascri| ×   bash - "brrcrites-pys( ×   +
brrcrites@pyschool:~/workspace (master) $ python pi_example.py
pi
3
3.14
brrcrites@pyschool:~/workspace (master) $
```

Expressions

- An assignment like this is called an expression

```
pi = 3.14
```

- An expression is made up of operators and operands, just like in math

Operators & Operands

- An operator is any mathematical or computational operation

+	add
-	subtract
/	divide
%	modulo
*	multiply
**	exponent
=	assignment
( )	parenthesis

- An operand is what those operators act upon, such as variables, numbers, and strings

- Let’s look at an example

```
x = (150 * 2) + 65
```

- Which parts of this statement are operators and which are operands

- The operators are in red, the operands are in green

```
x = (150 * 2) + 65
```

Evaluating Expressions

- Now that we are creating more complex expressions than simple assignments, how are they evaluated?
- Then the expression is evaluated the same way as any other math expression (remember PEMDAS)

Parenthesis ( )

Exponents **

Multiply *

Divide /

Add +

Subtract -

- PEMDAS is an acronym for the order of operations
- We evaluate mathematical expressions in this order (top to bottom), where sections in parenthesis are evaluated first, then exponents, multiplication, division, addition, and finally subtraction
- When we evaluate expression in computer science, we also follow this ordering

Evaluating Math

- Lets look back at our example, what would this be evaluated to?

```
x = (150 * 2) + 65
```

- We start with parenthesis, which has the highest priority

```
x = (150 * 2) + 65
```

- Since there is only one operation there (multiplication), we do that calculation (`150 * 2 = 300`)

```
x = 300 + 65
```

- Now we once again only have one operation (addition) so that calculation is done (`300 + 65 = 365`) and we are finished

```
x = 365
```

- Let’s look at a more complicated example now, how would this be evaluated?

```
x = (500 * 2**2 / 50) + 35 - 5 * (3 + 2**2)
```

- We see here that we have two sets of parenthesis, both of these have to be evaluated before we can move on. Let's start with the first set of parenthesis

```
(500 * 2**2 / 50)
```

- Now within, the highest priority is the exponent, so we evaluate the 2**2 first (`2**2 = 4`)

```
(500 * 4 / 50)
```

- Then the multiplication is the next highest (`500 * 4 = 2000`), followed by the division (`2000 / 50 = 40`)

```
(2000 / 50)
```

```
40
```

- After evaluating that first parenthesis, we have the following equation

```
x = 40 + 35 - 5 * (3 + 2**2)
```

- Now we need to evaluate the second parenthesis

```
(3 + 2**2)
```

- We know that exponents have a higher priority than addition, so we calculate the exponent (`2**2 = 4`)

```
(3 + 4)
```

- Then the addition (`3 + 4 = 7`), and we are done

- We have completed all the parenthesis now, so now we can move on to the rest of the equation

```
x = 40 + 35 - 5 * 7
```

- Looking back at PEMDAS, the highest priority is the multiplication (`5 * 7 = 35`)

```
x = 40 + 35 - 35
```

- Next is the addition (`40 + 35 = 75`), followed by the subtraction (`75 - 35 = 40`)

```
x = 75 - 35
```

```
x = 40
```

- Which gives us our final solution

- What about when those expressions have variables in them?

- First, **variables are replaced with their value**

- Then the expression is evaluated the same way was it would be in math (PEMDAS)

```
a = 10
b = 20
c_squared = a**2 + b**2
```

- How would we evaluate the following expression?

```
c_squared = a**2 + b**2
```

- We start by **replacing the variables a and b with their values** (**`10`** and **`20`**)

```
c_squared = 10**2 + 20**2
```

- Next, we solve for the exponents (`10**2 = 100` and `20**2 = 400`)

```
c_squared = 100 + 400
```

- Finally, we do the addition (`100 + 400 = 500`) and we are finished

```
c_squared = 500
```

Evaluating Strings

```
beginning = “it was the best of times,”
ending = “ it was the worst of times”
```

- In python, many math operators can be used on other data types

```
tale_of_two_cities = beginning + ending
```

- In this case, the + symbol has been written to return the second string added, or appended, to the end of the first

```
tale_of_two_cities = beginning + end
```

```
tale_of_two_cities = “it was the best of
times,” + “ it was the worst of times”
```

```
tale_of_two_cities = “it was the best of
times, it was the worst of times”
```

Comments

- We use comments to make notes to ourselves and other programmers, they are ignored by Python
- These comments are lines that start with a # symbol

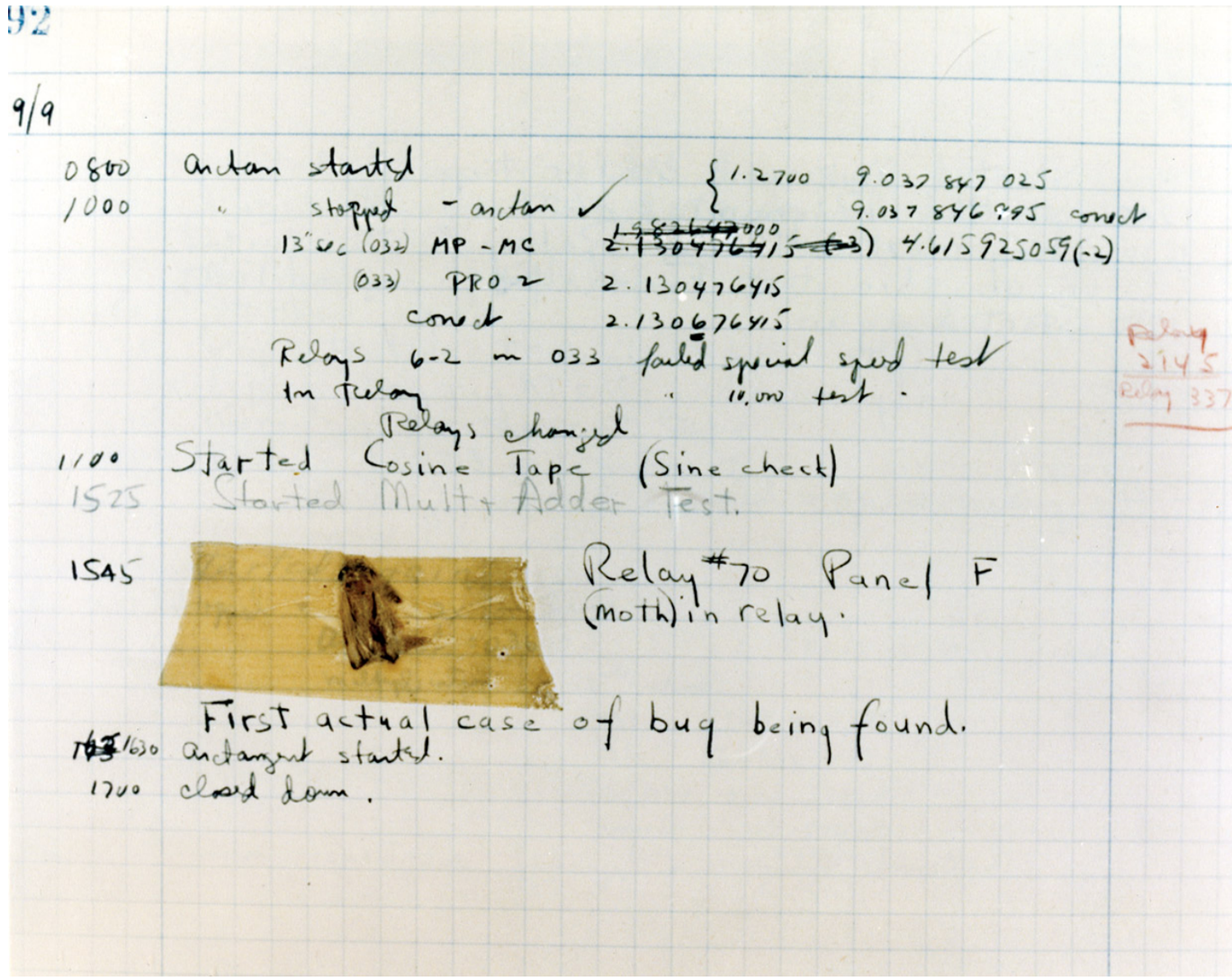
```
#Conversion from Fahrenheit to Celsius
C = (F - 32) * (5/9)
```

- Comments are very important for making your code easier to read and understand

Programming

- Now that we understand how statements are written and evaluated, we can write more complex code
- More complex code leads to…

Bugs



The first computer bug was a real bug
A moth flew into a relay, causing a problem with the machine

Bugs

- When there is an error in your code, it is called a bug
- Bugs are **very common**
- We call the process of finding and fixing bugs debugging
- There are three major types of bugs: syntax errors, runtime errors, and logic errors

Syntax Errors

- Errors in syntax (grammar) of the program that make it so the computer can't understand what you are trying to instruct it to do

```
>>> pritn “hello world”
File "<stdin>", line 1
  pritn "hello world"
                     ^
SyntaxError: invalid syntax
```

- “print” is spelled incorrectly, so it returns an “invalid syntax” error

Runtime Errors

- Errors that occur during the running of the program, usually resulting from the program trying to run an impossible instruction

```
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

- Since `x` was not defined, the program does not know what to do when we ask it to print `x`

Logic Errors

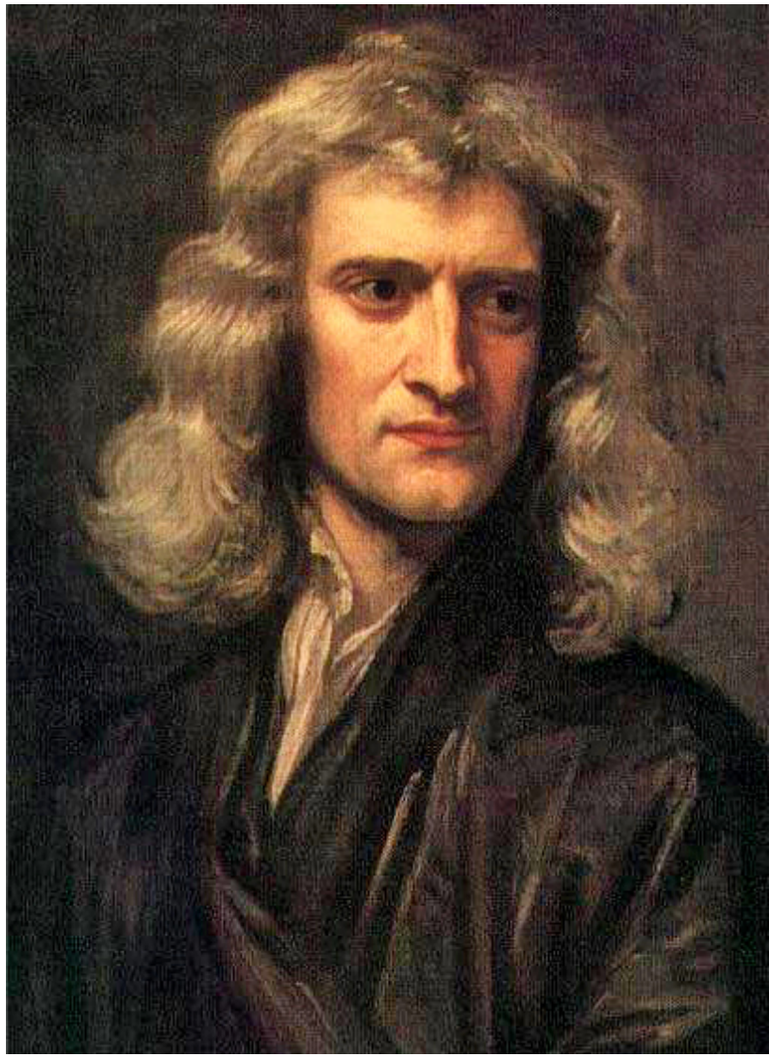
- Errors that prevent the program from doing what you intended for it to do, but do allow it to run to completion

```
>>> C = (F - 31) * (5/9)
```

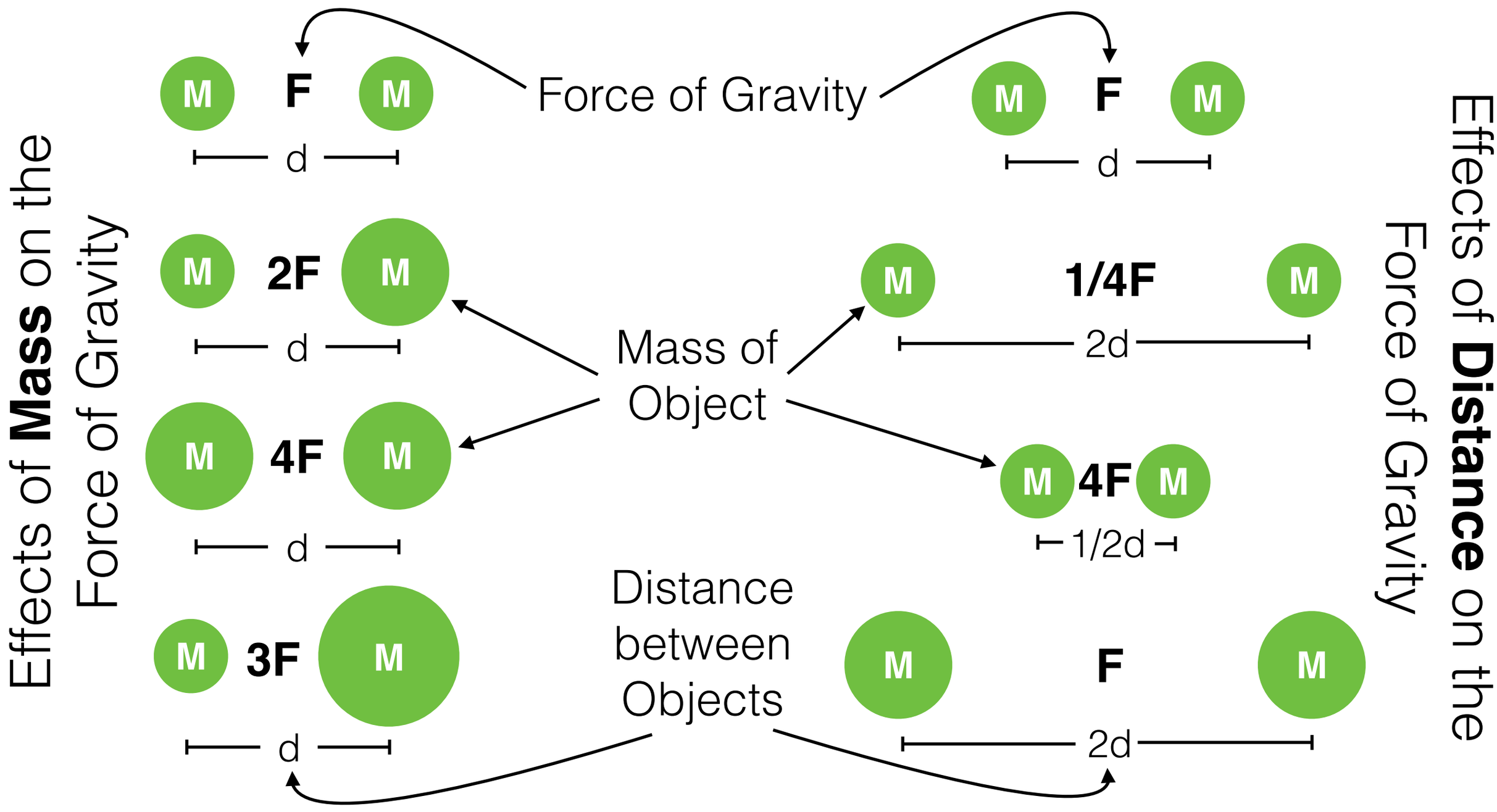
- Some of you may recognize this as the conversion from Fahrenheit to Celsius
- You may also notice that it should be `(F - 32)`
- Logic errors are not (usually) caught by the language and are usually the hardest to debug

Universal Law of Gravity

- Sir. Isaac Newton's Universal Law of Gravity states that any two bodies in the universe attract each other with a force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between them



- Here we see examples of the changes in mass being directly proportional to their product, and the changes in distance being inversely proportional to the square of the distance



- The formula below is Sir. Isaac Newton's theory formalized mathematically

$$F = G*(m1*m2/d**2)$$

- We can see that the m1*m2 section refers to the formula being proportional to the product of the masses

- We can also see that the 1/d**2 section refers to it being inversely proportional to the square of the distance

- G is the gravitational constant and is the physical constant involved in calculating the gravitational force between two bodies

Universal Law of Gravity Formula

F = G*(m1*m2/d**2)

```
G = 6.673*(10**(-11))
1m = 5.972*(10**24)
2m = 7.34767309*(10**22)
d = 384400*1000
F = (g*1m*2m/d)**2
print F
```

- We have written the Universal Law of Gravity code in python here, where `G` is the gravitational constant, `1m` is the mass of the earth and `2m` is the mass of the moon, `d` is the distance between the planets, and `F` is the force between the planets

- However, there are some bugs in this code, can you find them, identify their type, and correct them?

- The variable `G` is declared as capitol `G`, but referenced later as lower case `g`
- Python thinks these are two different variable names and that `g` is undeclared
- Undeclared variables cause a runtime error, so lets change `g` to `G`

```
G = 6.673*(10**(-11))
1m = 5.972*(10**24)
2m = 7.34767309*(10**22)
d = 384400*1000
F = (g*1m*2m/d)**2
print F
```

Error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
```

- Remember when naming variables, they need to start with a letter
- These improperly named variables would cause a syntax error

```
G = 6.673*(10**(-11))
1m = 5.972*(10**24)
2m = 7.34767309*(10**22)
d = 384400*1000
F = (g*1m*2m/d)**2
print F
```

Error:

```
  File "<stdin>", line 1
    1m = 5.972*(10**24)
     ^
SyntaxError: invalid syntax
```

- Let's look back at the original equation for the universal law of gravity

F = G*(m1*m2/d**2)

- Notice that instead of the denominator **d** being squared, the entire equation is squared

- Since the program runs fine but is incorrect, this is a logic error

```
G = 6.673*(10**(-11))
1m = 5.972*(10**24)
2m = 7.34767309*(10**22)
d = 384400*1000
F = (g*1m*2m/d)**2
print F
```

Error:

Result:
5.80249454707e+63

Correct Result:
1.98163649329e+20

- The runtime error is easy to fix, simply rename `g` to `G`

```
G = 6.673*(10**(-11))
1m = 5.972*(10**24)
2m = 7.34767309*(10**22)
d = 384400*1000
F = (g*1m*2m/d)**2
print F
```

```
G = 6.673*(10**(-11))
1m = 5.972*(10**24)
2m = 7.34767309*(10**22)
d = 384400*1000
F = (G*1m*2m/d)**2
print F
```

- We can fix the syntax error by putting the numbers after the letters so `1m` and `2m` become `m1` and `m2`

```
G = 6.673*(10**(-11))
1m = 5.972*(10**24)
2m = 7.34767309*(10**22)
d = 384400*1000
F = (G*1m*2m/d)**2
print F
```

```
G = 6.673*(10**(-11))
m1 = 5.972*(10**24)
m2 = 7.34767309*(10**22)
d = 384400*1000
F = (G*m1*m2/d)**2
print F
```

- The logic error can be fixed by correcting the equation for force

```
G = 6.673*(10**(-11))
m1 = 5.972*(10**24)
m2 = 7.34767309*(10**22)
d = 384400*1000
F = (G*m1*m2/d)**2
print F
```

```
G = 6.673*(10**(-11))
m1 = 5.972*(10**24)
m2 = 7.34767309*(10**22)
d = 384400*1000
F = G*(m1*m2/d**2)
print F
```

Law of Gravity

- Now, lets try writing that corrected Universal Law of Gravity code we just saw as a script

- Create a new file named **`law_of_gravity.py`** and enter the following code

```
G = 6.673*(10**(-11))
m1 = 5.972*(10**24) #Mass of Earth
m2 = 7.34767309*(10**22) #Mass of the Moon
d = 384400*1000
F = (G*m1*m2)/d**2
print F
```

- Now, run your code from the terminal (`python law_of_gravity.py`)
- What result do you get printed to your terminal?

- The answer is approximately

```
1.98163649329e+20
```

- your answer will be slightly different due to rounding, but if you got something very different you likely have a logic error

Scientific Notation

```
1.98163649329e+20
```

- Note that python prints large numbers in a scientific notation called e notation

- The e breaks up the real number and the exponent in a number written in scientific notation

- So the above example would be equivalent to

```
1.98163649329*10**(+20)
```

- The program we wrote is nice, because if we did it on a calculator we would need to enter the gravitational constant every time, and there is a good chance that we could enter something incorrectly

- However, it would be great if we didn’t have to open and edit the code every time we wanted to calculate different planets with different distances and masses

User Input

- You can prompt the user of your program to give you a value, this is called user input
- This is done with the `raw_input` function
- A function is a set of instructions that can be run with a single command, in this case the command `raw_input`

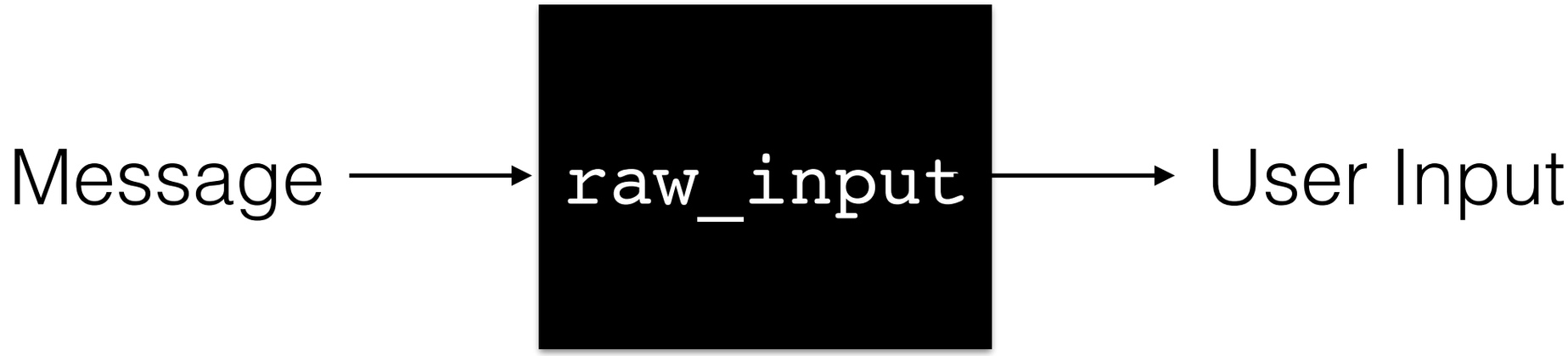
- Functions take some number of parameters (that number can be 0) and may or may not return some value

- For example, we would call `raw_input` in the following way

```
name = raw_input("What is your name?")
```

- It has a single parameter, in this case the string `"What is your name?"`, and this function will print this string to the terminal and wait for the user to type a response

- The function then has a return value of what the user typed into the terminal, which must be captured by assigning it to a variable, in this case the variable `name`



- You can think of it like a black box, parameters go in and a result comes out

- In the case of **`raw_input`**, a message string is input as a parameter

- The function itself displays the message as a prompt to the user and then waits for user input

- It's return value is then the user input string that was entered in the terminal

- The print keyword is actually a function with the string message as its parameter and no return value

- Print is used so often though that it is designed to be used without the parentheses around the parameters (remember python is designed to be easily written and read)

- It will still work with the parentheses however, go ahead and try it yourself

```
print(“hello world”)
```

Hello User

- Try this yourself, open your **`hello_world.py`** program
- Instead of assigning **`name`** a string that is your name, use the **`raw_input`** function to ask the user for their name

```
print "Hello World"
x = raw_input("Enter you name here: ")
print x
```

- Now, try run your program from terminal, you'll notice that the program won't complete until you have entered in your name when prompted and hit enter

```
brrcrites@pyschool:~/workspace (master) $ python 6_user_input_hello_world.py
Hello World
Enter you name here:
```

```
brrcrites@pyschool:~/workspace (master) $ python 6_user_input_hello_world.py
Hello World
Enter you name here: Issac Newton
Issac Newton
brrcrites@pyschool:~/workspace (master) $
```

- The function **`raw_input`** takes terminal user input, and that input is always seen as a string (even when you enter a number)

- If we want our input to be of another type (like an integer) we have to specifically tell the program that what the user input is an integer instead of a string

- We do this by **casting** the value to an integer

Casting

- You can cast the string that you get from the function **`raw_input`** to an integer by calling the keyword **`int()`**, with the value as the only argument

```
input = raw_input(“Enter Number “)
number = int(input)
```

- the function **`int(input)`** will return the user input as an integer instead of a string, which we need to save into the variable **`number`** (which will be an integer)

Universal Law of Gravity

- Now lets try doing our universal law of gravity example again, this time with user input
- Open your **`law_of_gravity.py`** file, and now ask the user for the mass of each planet (**`mass1`**, **`mass2`**) and the distance between them (**`distance`**) using **`raw_input`**

Control Flow

- What if we only want certain expressions to be run at certain times?
- So far we have only seen programs that execute every expression, in order, from top to bottom
- We need a way to control which expressions will and will not be run, also known as the flow of the code
- This concept is called control flow (because we control the flow of the code)

If

- One way we can control the flow of the program is with the `if` keyword

```
if i < 4:
    print "less than four"
```

- If the condition is true (`i < 4`), then all indented code will execute (`print "less than four"`)

```
if i < 4:
    print "less than four"
```

- Like with our **raw_input** function, the **if** is also a function, where **if** is the command you use to run a set of instructions

- In this case, that parameter is the condition (i < 4)

- Since the **if** keyword is used so often, it does not require parenthesis around the parameter (in the same way as **print**), however you do have to add a colon (:) at the end of the condition

- It also has no return value, it simply changes the control flow

Conditions

- We’ve talked a little about conditions, but lets look a little deeper now

```
if i < 4:
```

- The condition `i < 4` makes sense in the case of an if condition
- We know it means if `i` is less than 4, then run the code associated with that if
- The condition can either be met or not met

Boolean Expressions

- When a condition is met, we say that the condition evaluates to **True**
- When the condition is not met, we say that the condition evaluates to **False**
- These values are called boolean values, and they can only have either a true or false value
- Any expression that evaluates to a boolean value is known as a boolean expression

- We have already seen that the less than (<) symbol gives a boolean value, since the evaluation of something like `a < b` can only be either true or false

- Here are some other common symbols used in boolean expressions

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal
!=	not equal

- You should take special notice of the equal condition, which is a double equal sign (==)
- Notice that to test for equality we use double equal signs (==), but when we assign a variable a value we use a single equals (=)
- Take extra care not to use a single equal sign in a condition, such as an if statement

- For example, when we assign the variable **`name`** to string value **`“Issac Newton”`**, we do it like this

```
name = “Issac Newton”
```

- But if we want to write a condition that tests if the variable **`name`** is **`“Issac Newton”`** or not, we would test it like this

```
if name == “Issac Newton”:
    print “Hello Mr. Newton”
```

Indenting in Python

- In Python, all control flow is determined by indentation and keywords
- Whenever a keyword that modifies control flow is used, the following code that is indented the same will all be run
- When the indent is removed, the change in control flow ends
- The if keyword and condition is known as an **if statement**, and all the code that will run when that if statement is true is collectively known as an **if block**

program

```
i = 2
j = 1
print “unindented code will always print”
if i == 1:
    print “but this only prints if i is 1”
if i == 2:
    print “and this only prints if i is 2”
    if j == 1:
        print “this prints when i is 2 and j is 1”
```

Two Indents, because there is an if statement nested within another if statement

program	output
`i = 6`	
`print "i > 3?"`	`i > 3?`
`if i > 3:`	
`    print "yes"`	`yes`
`if i <= 3:`	
`    print "no"`	
`print "i > 8?"`	`i > 8?`
`if i > 8:`	
`    print "yes"`	
`if i <= 8:`	
`    print "no"`	`no`
`print "end"`	`end`

- Code with no indentation will always be run
- That is why `i > 8` is printed, because it has no indentation and therefore does not belong to an if statement

```
program
a = 5
b = 10
if a < b:
    print “a < b”
if a + 2 < b:
    print “a + 2 < b”
if a + 5 < b:
    print “a + 5 < b”
```

- What will be output by this program?

program

```
a = 5
b = 10
if a < b:
    print “a < b”
if a + 2 < b:
    print “a + 2 < b”
if a + 5 < b:
    print “a + 5 < b”
```

output

```
a < b
a + 2 < b
```

- It does not print `a + 5 < b` because `a + 5 = 10`, and 10 is exactly equal to the value of `b`
- If the line had been `if a + 5 <= b:` then it would have printed, since `a + 5` is **less than or equal to** b

```
program
a = 5
b = 10
if a < b:
    print “a < b”
if a + 2 < b:
    print “a + 2 < b”
if a + 5 < b:
    print “a + 5 < b”
```

- It is also important to note here that while we have `a + 2` and `a + 5` in the condition, the value of a will not change since we do not assign the result back to a variable

- For that we would need `a = a + 2` and `a = a + 5`, which would not be the same condition

```
program
x = 2
y = 1
if y < x:
    print “y < x”
    y = y + 10
if x < y:
    print “x < y”
    x = x + 15
if x > 10:
    print x
if y > 10:
    print y
```

- How about this program?
- Create a new file called `if.py` and write this program, then run it and see if the results match your ideas
- Pay careful attention to the indentation

program

```
x = 2
y = 1
if y < x:
    print "y < x"
    y = y + 10
if x < y:
    print "x < y"
    x = x + 15
if x > 10:
    print x
if y > 10:
    print y
```

- When an if statement is executed, it executes all the code with the same indent level after the if
- Because of this, both the print statements and the additions to x and y are executed

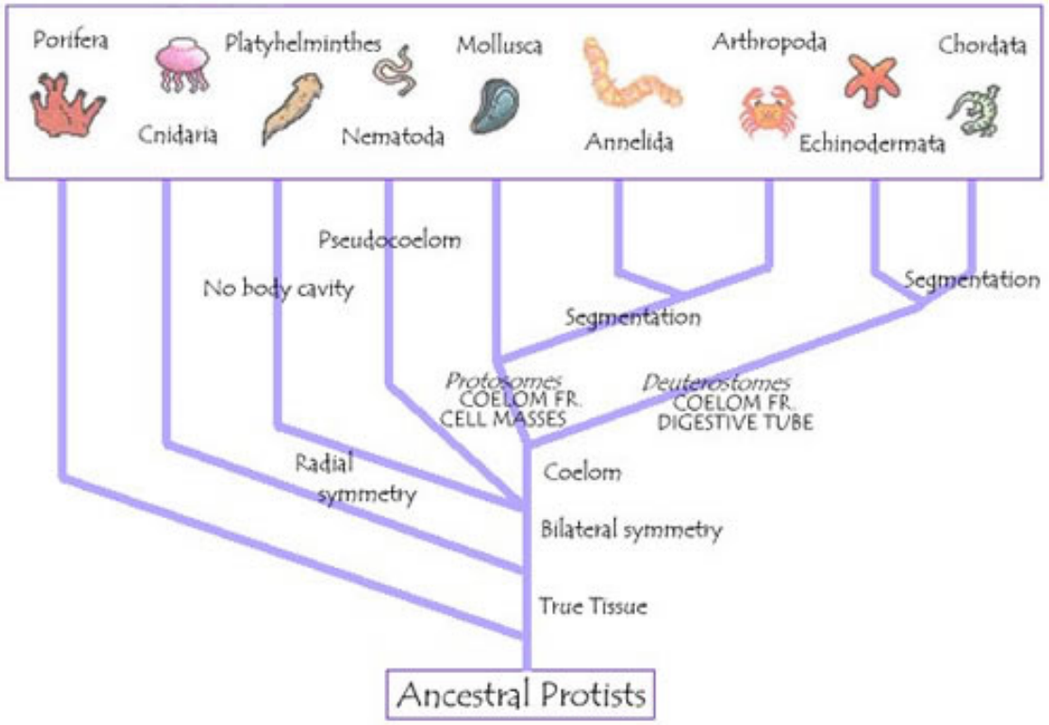
output

```
y < x
x < y
17
11
```

Taxonomy

- Taxonomy is the science of defining groups of biological organisms on the basis of shared characteristics and giving names to those groups (known as taxa)

- We usually determine an organisms taxa by using a branching classification tree

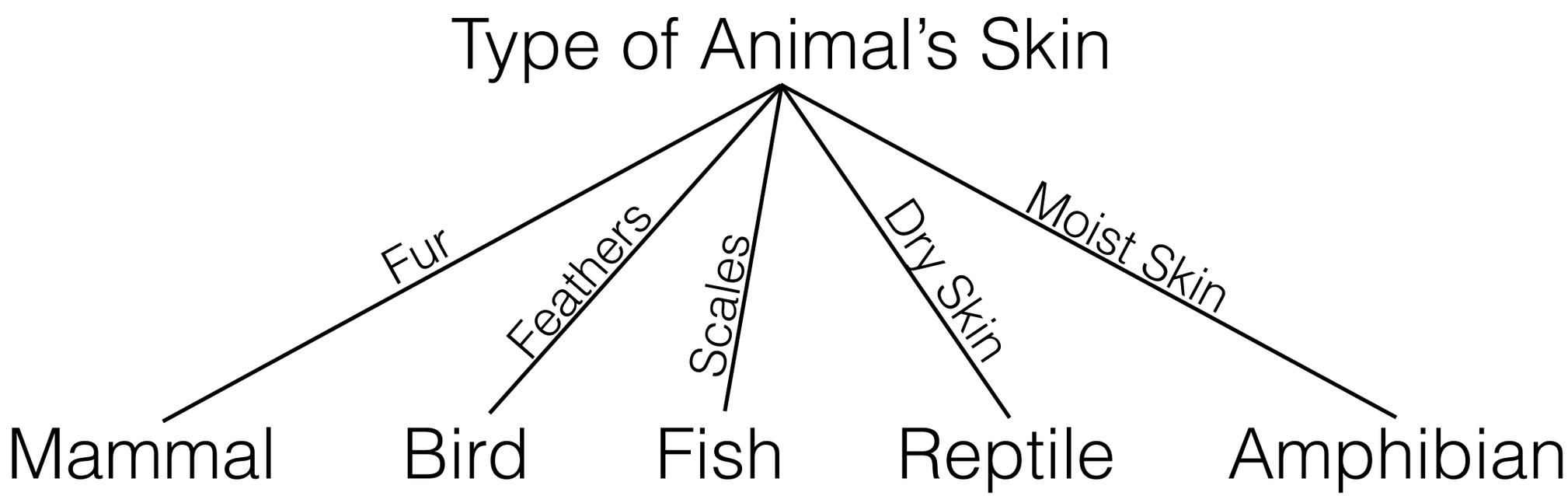


Classification Trees

- Classification trees ask simple questions about an organism's characteristics, and each answer reduces the number of possible taxa until enough questions are answered that only one taxa remains

- Let's apply what we have learned about the if statement to create a system that automatically determines the taxa of an animal by asking the user simple questions that would appear on a classification tree

- For example, if we wanted to determine the class of an animal, we might have a classification tree like this



- Let's turn this classification tree into a program called **`classification_tree.py`**

- Ask the user for the type of skin the animal has, check the type of skin with if statements, and print out what class the animal belongs to

If

```
i = 6
print “i > 3?”
if i > 3:
    print “yes”
if i <= 3:
    print “no”
```

- Notice the two if statements in this piece of code cover all possibilities for `i`
- `i` must be either greater then 3, less than 3, or equal to 3
- The second if statement covers all options for the value of `i` not covered by the first if statement
- There is actually a keyword that acts as a shortcut for this

Else

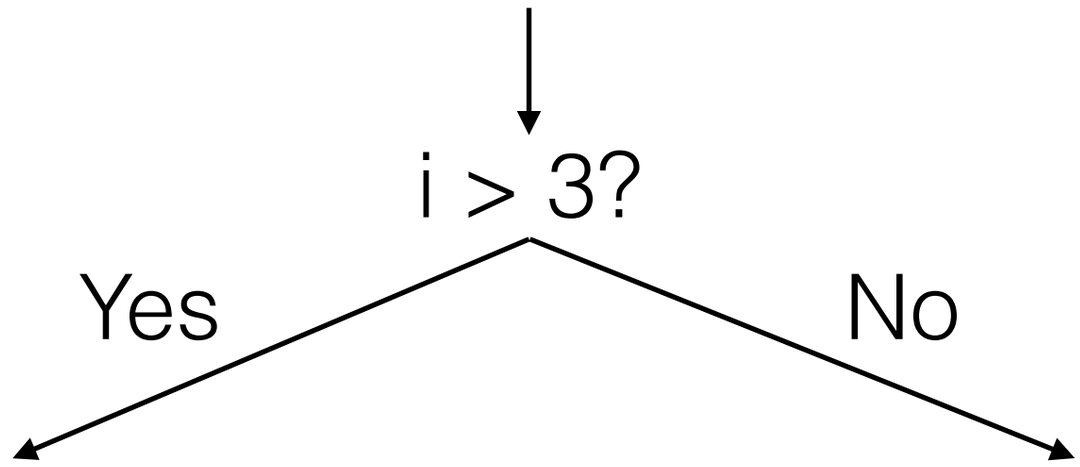
```
i = 6
print “i > 3?”
if i > 3:
    print “yes”
else:
    print “no”
```

- `else` is a keyword that follows an if statement, and the else statement will run whenever the if does not run
- In this case, since `i` is greater than 3, the if statement will run, so the else statement will not run

```
i = 1
print “i > 3?”
if i > 3:
    print “yes”
else:
    print “no”
```

- Now that `i = 1`, the if will be false (since `i` is not greater than 3), so the else will execute.
- This makes logical sense, since if `i` is not greater than 3, it must be either less than or equal to 3

```
i = 1
print "i > 3?"
```




```
if i > 3:
    print "yes"
```

```
else:
    print "no"
```

- You can think of every pair of if and else statements as a branching choice

Classification Tree

- Let's go back to **`classification_tree.py`**, and add an enhancement using else

- In the last program, if the user entered something invalid the program would simply end without printing anything

- Lets add an else statement that will tell the user that they entered an invalid input

- What if we want more than just two choices?
- In the last example, we can only choose between `i` being greater than 3 or less than or equal to 3
- What would we do if instead we wanted we wanted to the options of `i` greater than 3, `i` less than 3 but greater than 2, or `i` less or equal to 2?
- There is another keyword for that called…

Elif

- `elif` is pronounced “else if” because that is what it abbreviates
- It is also a function keyword, and is written the same as an if statement
- `elif` works like a hybrid of the if and else statements

- Both Else and Else if statements must follow either an if statement or an else if statement

- Else statements represent the end of a logical block (more on blocks in a minute), saying that if none of the above statements have happened, then this must happen

- However, else if statements work a little differently, they execute only if none of the if or else if statements above them have been executed, and their condition is true

Elif

```
i = 15
if i > 10:
    print “i > 10”
if i > 5:
    print “10 >= i > 5”
else:
    print “i < 5”
```

- Notice here we have two if statements followed by an else statement
- In this case, since we have `i = 15` at the top, it will print both `i > 10` and `10 >= i > 5`, which is incorrect

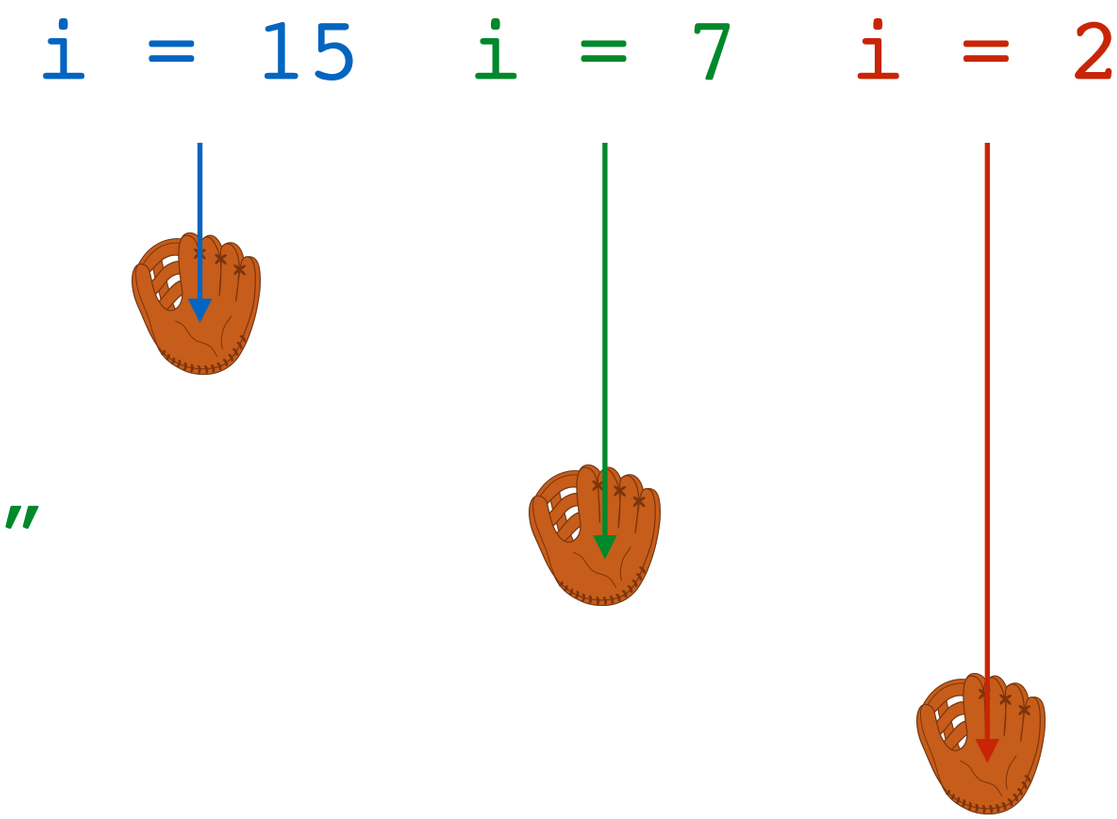
```
i = 15
if i > 10:
    print “i > 10”
elif i > 5:
    print “10 >= i > 5”
else:
    print “i < 5”
```

- Now with `i = 15` at the top of this code, `i > 10` would print, and `10 >= i > 5` would not print
- If we replace `i = 15` with `i = 7` at the top, `10 >= i > 5` would print
- If we had `i = 2` at the top, `i < 5` would print

Control Flow

- Think of it like the variables are falling down through the code
- The variables can fall through any number of if statements, only causing them to run if the condition is met
- They can only fall through else if statements if they didn't get caught by an if or else if statement that comes before it
- They can never fall through an else statement without triggering it, unless it was already caught by an if or else if that came before it

```
if i > 10:
    print “i > 10”
elif i > 5:
    print “10 >= i > 5”
else:
    print “i < 5”
```



- i = 15 is caught by if(i > 10), since there are only else if and else statements afterwards, so it stops
- i = 7 falls through if(i > 10) and is caught by elif(i > 5). There is only an else after it, so it stops
- i = 2 falls through if(i > 10) and elif(i > 5), so the else will catch it

Elif

```
i = 2
if i > 10:
    print “i > 10”
elif i > 5:
    print “10 >= i > 5”
```

- Remember though, if statements and else if statements aren’t required to have a matching else statement
- In this case, nothing would get printed since `i` satisfies neither the if statement nor the else if statement

If, Elif, and Else Blocks

- Sets of if, else if, and else statements create logical blocks, and understanding how these blocks work is very important
- It is very easy to make a mistake in your block that will cause a bug that is very hard to debug

Block Construction

- There are a couple of simple rules for making blocks
  - An if statement is **always** the start of a block
  - Any number of else if statements can go in between the start and the end of a block
  - If your block has an else statement, it is **always** the end of a block

```
i = 6
if i > 10:
    print “i > 10”
elif i > 5:
    print “10 >= i > 5”
elif i > 3:
    print “5 >= i > 3”
elif i > 1:
    print “3 >= i > 1”
else:
    print “i <= 1”
```

This is one single block, starting with an if statement and ending with an else statement. The control flow will continue until it is caught or it hits the else

```
i = 6
if i > 10:
    print “i > 10”
elif i > 5:
    print “10 >= i > 5”
elif i > 3:
    print “5 >= i > 3”
if i > 5:
    print “i > 5”
else:
    print “i <= 5”
```

Block 1

Block 2

This is two different blocks. Since an if statement always starts a block, when an if statement appears any open blocks are automatically closed.

- Now Let's take a look at these blocks

```
x = 7
y = 6
if x < 5:
    print "x is too small"
elif y < 5:
    print "y is too small"
else:
    print "both numbers are large enough"
if x > 10:
    print "x is too large"
if y > 10:
    print "y is too large"
```

- Can you identify where the blocks start and stop?

```
x = 7
y = 6
if x < 5:
    print “x is too small”
elif y < 5:
    print “y is too small”
else:
    print “both numbers are large enough”
if x > 10:
    print “x is too large”
if y > 10:
    print “y is too large”
```

B1

B2

B3

Classification Tree

- Let's go back to our `classification_tree.py` program again
- Since an animal can only belong to one class, we know that if one of the conditions is true, then none of the others can be true
- Use a combination of if, elif, and else statements to re-write the program so that it will stop when it has found a correct classification
- Don't forget we still need to check for invalid user input

Nesting Statements

- You can also create blocks of if, else if, and else statements within other blocks
- As long as you follow the rules for constructing blocks, you can have an infinite number of blocks within other blocks

```
if x < 5:
    if y == “big”:
        z = 20
    else:
        z = 2
else:
    if y == “big”:
        z = 10
    elif y == “med”:
        z = 5
    else:
        z = 3
```

Outer Block 2

```
if x < 5:
    if y == “big”:
        z = 20
    else:
        z = 2
else:
    if y == “big”:
        z = 10
    elif y == “med”:
        z = 5
    else:
        z = 3
```

Inner Block 1

```
if x < 5:
    if y == “big”:
        z = 20
    else:
        z = 2
else:
    if y == “big”:
        z = 10
    elif y == “med”:
        z = 5
    else:
        z = 3
```

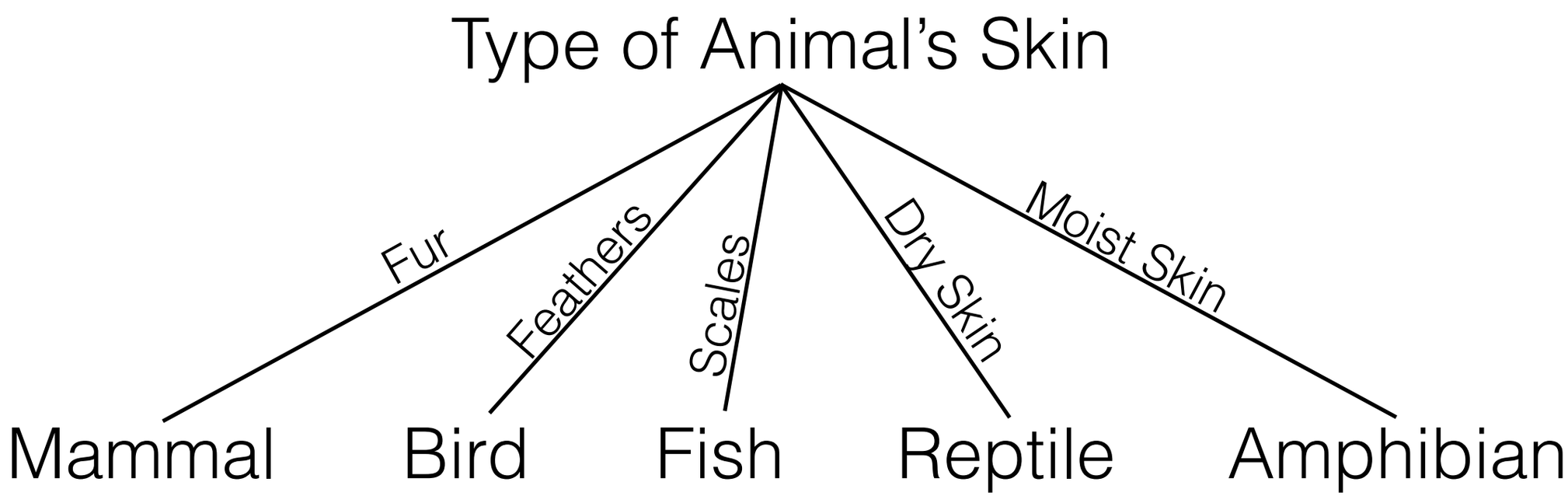
Inner Block 2

- Notice that for the Inner Block 1 to have a chance to trigger one of its branches, first the `x < 5` condition must be found True, and for Inner Block 2 to have a chance to trigger any of its branches, `x < 5` must be evaluated False

Classification Tree

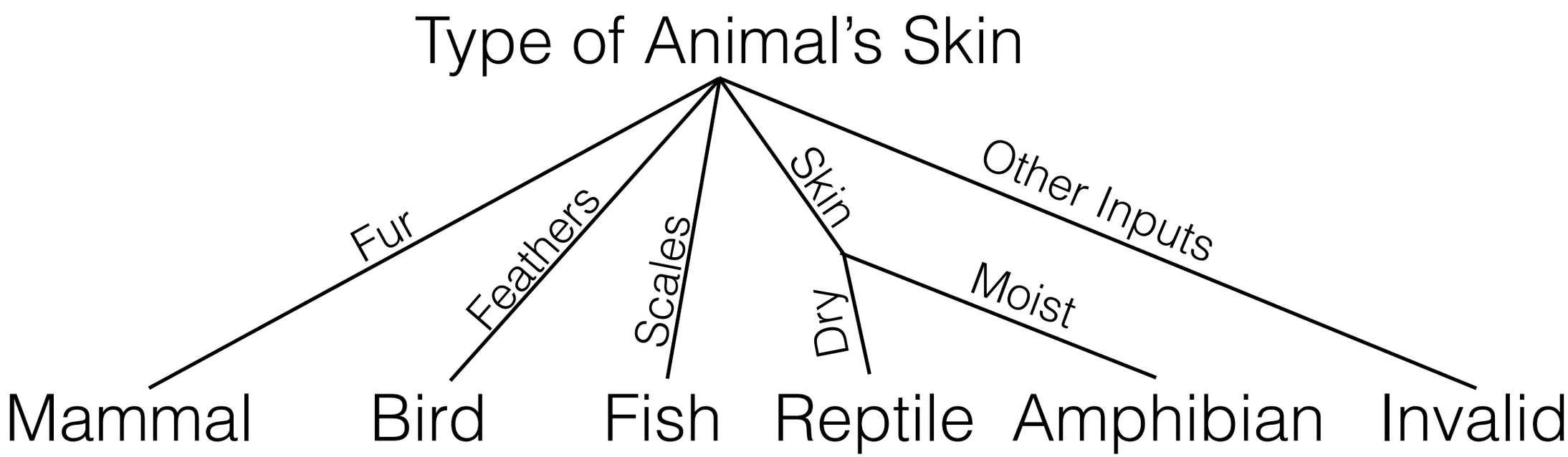
- In our **`classification_tree.py`** program, we have the user specify if the skin is “dry skin” or “moist skin”
- Instead of this, we should allow the user to simply input “skin” and then ask them if the skin is “moist” or “dry” to complete the classification
- Modify your **`classification_tree.py`** program so that when a user enters “skin”, they are prompted for a type of skin and then the classification continues
- You can accomplish this by nesting if statements

- Remember we started with this classification tree



- What would the program we just wrote look like as a classification tree?

- It should look something like this



- You can see the nested if statement for skin creates another branch of moist or dry after the skin choice

The Lazy Physics Student

- Lets go back to our example using Newton's Universal Law of Gravity

F = G*(m1*m2/d**2) G = 6.673 * (10**(-11))

- Open your `law_of_gravity.py` file

- We want you to write a program that can calculate and print the force due to gravity exerted between the sun and another planet at either its highest or lowest point

- Prompt the user to enter a planet name, and save their answer into the variable **`planet`**

- Second, prompt the user for if the planets are **`"Far"`** or **`"Close"`**, and save their answer to a variable **`distance`**

- Now, write blocks of if, else if, and else statements that will assign variable **`m_2`** the mass of the planet that the user entered

- Also, write if, else if, and else statements that assign variable **`d`** the distance between the chosen planet and the sun depending on if distance is set to **`"Far"`** or **`"Close"`**

- After you have assigned these values, calculate and print the gravitational force between the two planets **`F`**

Planet Name	Mass (kg)	Closest Distance from the Sun (m)	Furthest Distance from the Sun (m)
Jupiter	1.8986*(10**27)	741*(10**9)	817*(10**9)
Saturn	5.6846*(10**26)	1.35*(10**12)	1.51*(10**12)
Neptune	10.243*(10**25)	4.45*(10**12)	4.55*(10**12)
Uranus	8.68*(10**25)	2.75*(10**12)	3*(10**12)
Earth	5.9736*(10**24)	147*(10**9)	152*(10**9)
Venus	4.8685*(10**24)	107*(10**9)	109*(10**9)
Mars	6.4185*(10**23)	205*(10**9)	249*(10**9)
Mercury	3.3022*(10**23)	46*(10**9)	70*(10**9)
Sun	1.9891*(10**30)	—	—

Hints: You can save some lines of code by using nested statements. You can check if values are equal using the == symbol

F = G*(m1*m2/d**2) G = 6.673 * (10**(-11))

- As you can see, prompting for a simple user input (like the planet name) makes it easier to run this program multiple times

- Also using a simple user input (instead of something like the exact values) make it much less likely that the user will type in something incorrectly

- User input does, however, allow for the introduction of even more bugs into your code

- Run your universal law of gravity program again, but this time choose Pluto as your planet, what happens?

Pluto is not a Planet

- Pluto is no longer a planet, so it was not included in the chart you used to create your program
- If you ended your if block with an else statement, then you output the incorrect value
- If you ended your block with an if or an else if, then you likely got a runtime error
- Neither of these are desired effects, it would be best if when the user put in invalid input we ended the program telling them they used the program incorrectly

Input Validation

- We know what improper inputs for this program are
  - Anything that is not the name of a planet that will be assigned to the **`planet`** variable
  - Anything other than "**`Close`**" or "**`Far`**" for the **`distance`** variable
- In order to avoid bugs or errors in our code, we should make sure that what the user input into our program will not cause our program to crash, this is called input validation

- What are some ways that we can validate our user’s input for the planet using control flow?

- There are many different ways, but let's look at one of the easier ones
- In our if/else statement tree, we should have an if or else if statement for each planet
- Since those are the only valid values for planet, we can simply add an else statement at the end of the block to catch all other inputs
- If none of the correct planets were executed, then we should tell the user that they entered an invalid planet and end the program in the final else statement
- But how do we purposefully end the program?

Exit

- We can end the program intentionally using a special keyword called **`exit`**

- **`exit`** is a function keyword, however it can be called with no parameter

```
exit()
```

- This tells the program that it should now quit without running any more code

- Since it exists the program, the **`exit`** function doesn't return any value

- Now, you may be thinking “but shouldn’t we just prompt the user for another planet instead of stopping the program?”

- Yes, that would make our program more usable, and we will do exactly that later on in this course

- For now, just exit the program purposefully

- This is called a “**graceful exit**” because we intentionally close the program instead of letting it crash

- We also need to validate the **`distance`** variable

- We could change the if and else block that checks the **`distance`** nested in each planet, but that would mean 7 changes

- An easier way would be to check right after we get the user input using **`raw_input(…)`**, we can check if distance is equal to "**`Close`**", equal to "**`Far`**", or equal to anything else

- If the variable **`distance`** is equal to anything but "**`Close`**" or "**`Far`**", then we want to tell the user they typed something in wrong and end the program

Other ways to Control Flow

- Now we know how to change the control flow of the program so that it will run some code but not other code

- What happens when we want to run the same code multiple times, say to print every number between 1 and 1 million

- Or if we want to ask the user for an input over and over again until they enter something valid

- We use something called loops to accomplish this

Loops

- Loops are basically what you expect, its a set of *expressions* (remember expressions?) that repeat (or loop) a set number of times (or number of **iterations**)
- An iteration is one run of a loop, so if we loop 10 times, we say we have run 10 iterations
- One way to create loops in python is with the keyword **`while`**
- Like **`if`** and **`elif, while`** is also a keyword function
- Also like **`if`**, **`elif`**, and **`else`** statements, **`while`** will run everything after it with the same indentation

While Loops

```
i = 0
while i < 5:
    print i
    i = i+1
```

- Does anyone want to guess what this loop does?
- How about how many iterations it performs?

program

```
i = 0
while i < 5:
    print i
    i = i+1
```

iteration	output
1	0
2	1
3	2
4	3
5	4

- This loop will print each number between 0 and 4
- It performs 5 iterations
- Lets take a look at the while loop and find out why

Anatomy of a While Loop

```
i = 0
while i < 5:
    print i
    i = i+1
```

- We use the keyword **`while`**, as in *while* A is true, do B
- In this example, the A is `i < 5` and the B is to **`print i`** and increment i (`i = i+1`)
- Like with if statements, everything with the same indentation after the **`while`** and condition will be included in the loop

variable declaration

condition

```
i = 0
while i < 5:
    print i
    i = i+1
```

keyword

increment

- Variable Declaration: our condition is based on i, so we have to assign it a value of 0 before the loop starts

- Condition: we want to stop at 4, so our condition is that `i < 5`

- Increment: every time we loop (each iteration) we want to increase `i` by 1, so we write `i = i+1` at the end of the loop

```
i = 0
max_input = raw_input("Enter max: ")
max = int(max_input)
while i < max:
    print i
    i = i+1
```

Remember that raw_input returns a string, which we need to convert to an integer before we can use it in our condition

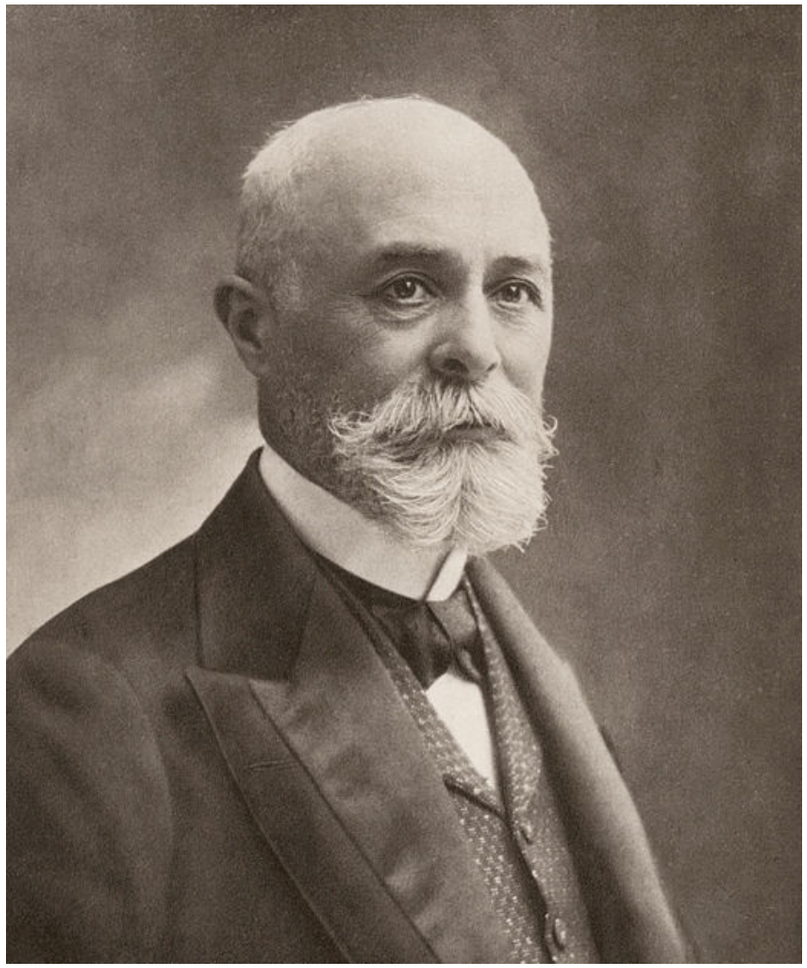
- How about this loop, how many iterations will this loop perform?

- Notice that you cannot predict how many iterations this loop will run for
- It all depends on what the user inputs as the **`max`**
- Special care is needed when using user input in loops, since the user can input anything
- Lets see how we can use these loops in the real world

Radioactivity

- Radioactivity was discovered in 1896 by the French scientist Henri Becquerel, while working with phosphorescent materials

- Many radioactive materials die at a known rate, which can be modeled with computers



Radioactive Decay

- As time passes, radioactive material degrades at a known rate

$$N = N_0 \, e^{-\lambda t}$$

- Where N is the current amount of material, N0 is the starting amount, e is Euler's constant, λ is the rate of decay, and t is the current time step

- We are going to write a program called **`radioactive_decay.py`** that calculates how much radioactive material is left at every time step between 0 and 10

- We will prompt the user to enter the starting amount of radioactive material (**`starting_amount`**) and the radioactive decay constant (**`decay_constant`**), remember that the **`starting_amount`** and **`decay_constant`** can be a decimal value, so make sure to cast it to a float (remember floats are decimal numbers, integers are round numbers)

- The only thing that will be changing in the equation is the amount of material left (N) and the time step (t)

- Our while loop will represent the passage of time, and we will go for 10 time steps

- Each time step, we will calculate the amount of material left and print it to the terminal

$$N = N_0 \, e^{-\lambda t}$$

```
while current_time < 10:
    amount = start_amount * (e ** . . .
    print amount
    current_time = current_time + 1
```

- Since we only want to have 10 time steps, we have `current_time < 10` as our condition, since `current_time` will be increased during each iteration

- `current_time` is increased at the end of the loop, and the new value is used in doing the next calculation

- In this way, we calculate the amount of radioactive material left every time step from 0 to 10

- If you forgot to increment `current_time` in the while loop, then you ran into an infinite loop

Infinite Loop

- When a loop runs forever, it is called an infinite loop
- Infinite loops should not be used intentionally, and can cause your computer or program to crash
- It is easy to accidentally forget to iterate closer to a false condition in your loop (like forgetting to increment **`current_time`** in the last example)
- Make sure to check that your loops have a way of getting a false condition eventually

Radioactive Decay

- Let's go back to our `radioactive_decay.py` example, and make a small change
- Now, instead of doing 10 time steps, have the user enter how many time steps the program should run for
- You should probably check that the value is reasonable (greater than 0 and less than 1000)

Complex Conditions

- Notice that we aren't restricted to having just one simple condition in our statements
- We can create complex expressions that evaluate to a boolean value (remember boolean values?)
- One way to do this is by having an expression on one or both sides of the symbol
- This works just like in math, and the expressions on either side can be as complex as you want, as long as they can evaluate to a boolean expression

- Something like this

```
if a * b * c < d * e * f:
```

- Is a valid condition, since each side of the < will be evaluated to a number and then the < will be evaluated with just one number on each side

a = 2, b = 3, c = 4, d = 1, e = 2, f = 3

```
if 2 * 3 * 4 < 1 * 2 * 3:
```

```
if 24 < 6: (True)
```

Boolean Expressions

- In addition to being able to use complex expressions in creating boolean expressions, you can also chain your boolean expressions together

```
if a > 10 and b < 10:
```

- The final evaluation of this entire expression will be true only when `a > 10` **and** `b < 10`

- There are a few key words like this that allow you to chain expressions together, they are known as **logical operators**

- There is something important that needs to be pointed out in the last example

```
if a > 10 and b < 10:
```

- Notice that each segment needs to be evaluated to true or false, so we could not do the following

```
if a and b > 10:
```

- This would evaluate if `a` was true first, then if `b > 10` was true, then if `a` **`and`** `b >10` is true

- This is the same for strings, so while we might expect to say this

```
if planet != “earth” and “mars”:
```

- We actually need to do this

```
if planet != “earth” and planet != “mars”:
```

- One way to help clarify complex conditions for yourself and others is to wrap each boolean condition in parenthesis

```
if (a > 10) and (b < 10):
```

```
if (planet != “earth”) and (planet != “mars”):
```

- Now we can see that each condition in parenthesis must evaluate to either true or false

- It is also easier to visually determine the result, since the logical operators stand out more in the expression

Logical Operators

- Logical operators are keywords that work in the same way that boolean expression symbols like < and >= do

- The difference is that they only allow boolean values on either side of the keyword

- However, since expressions are evaluated before the logical operators, we can use entire boolean expressions as our boolean values

```
a = 5
b = 10
c = 3
if a + b > c**2 and a + c < b:
    print a + b + c
```

- This program will print `a + b + c` because `a + b > c**2` evaluates to True and `a + c < b` evaluates to True

- We intuitively know how the `and` logical operator works, but lets look at it more specifically with the other logical operators

Truth Tables

`and`

LEFT	RIGHT	RESULT
True	True	True
True	False	False
False	True	False
False	False	False

`or`

LEFT	RIGHT	RESULT
True	True	True
True	False	True
False	True	True
False	False	False

- `and` evaluates to true when its left and right side are both true, `or` evaluates to true when either its left or right side is true, or if both sides are true

Logical Operators in Action

- You may have to reference the tables, known as truth tables, for a while but using **`and`** and **`or`** will become intuitive

- Usually you will know what operators to use simply by stating the problem in your head

- "I need this if statement to execute when a + b is less than c and b + c is less than d and a is greater than e"

```
if (a + b < c) and (b + c < d) and (a > e):
```

Not Operator

- The last logical operator is the **`not`** keyword
- **`not`** comes before any logical expression and inverts it, so True becomes False and False becomes True

```
while not “istanbul” == “constantinople”
```

- The above is equivalent to

```
while “istanbul” != “constantinople”
```

Guess the Number

- Now, let's build a simple game, create a new file called **`number_guess.py`**
- Create a variable called **`secret_number`**, and assign it an integer value between 1 and 10
- Now, use while loops and **`raw_input`** to ask the user to guess the number, and have them keep guessing until they get it right
- Make sure to output if they got the answer correct or incorrect

- Now, let's change the game a bit, make your secret number between 1 and 100
- When the user inputs their number…
  - Validate that their guess is between 1 and 100
  - If their guess is lower than the correct number, tell them their guess is too low
  - If their guess is higher than the correct number, tell them their guess is too high
  - If they are correct, tell them congratulations and end the game

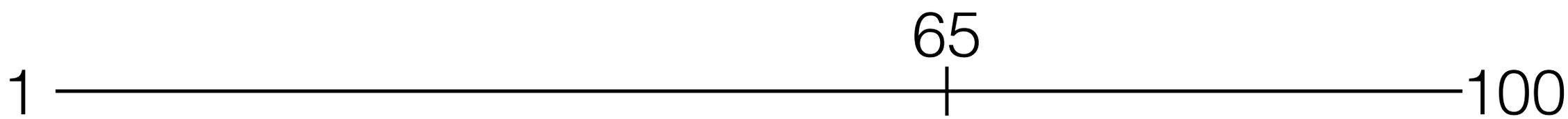
- Let's stop and take a look at that last line

```
elif user_input > secret_number:
    print "Too high, guess again"
```

- Should this last statement be either
  - `elif user_input > secret_number:`
    - or
  - `else`

- Let's take a look at a number line to help us decide

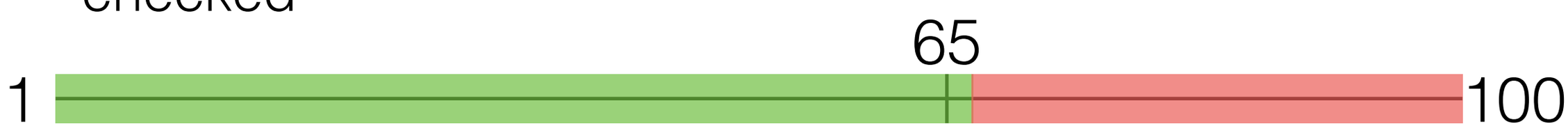
- The first thing we do is check that the input is between 1 and 100, so we can restrict our number line to this

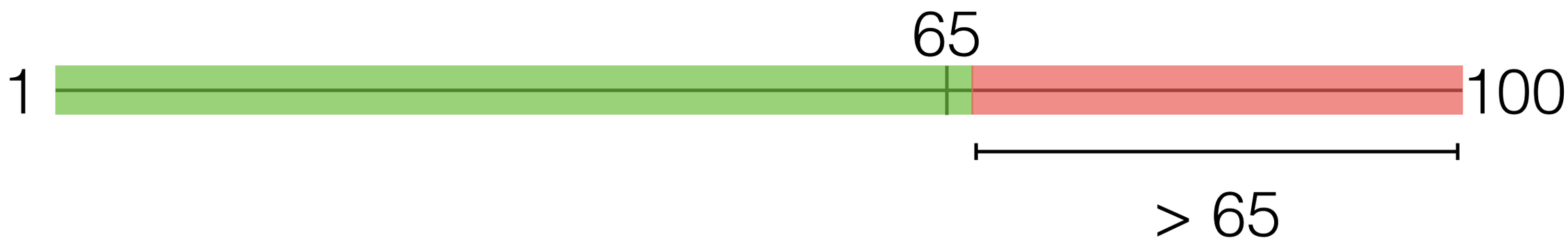


- Next, we check that the user input is not the answer

- Then we check if the user input is below the secret number

- This leaves us with the red section that has not yet been checked





- We know that this is any number greater than secret_number, so elif user_input > secret_number must be correct

- However, since we've already check all the other possibilities on the number line, else covers the exact same numbers that the elif does

- So both answers are correct!

- Now let's expand out **`number_guess.py`** game
- When the user guesses a number that is too low, they should stop guessing lower numbers because those cannot be the answer
- When the user guesses a number that is too high, any number higher than that can also not be the answer

- Give your program two new variables **`minimum`** and **`maximum`**, which start out as 1 and 100 respectively

- As the user guesses, the **`minimum`** and **`maximum`** variables should be updated to be the minimum number and the maximum number guessed by the user

- Now, instead of validating that the users guess is between 1 and 100, you should validate that their guess is between the minimum number they have guessed so far, and the maximum number they have guessed so far

- Let's go back to our **`radioactive_decay.py`** example and make a similar addition

- Now, instead of going through 10 time steps, ask the user for the number of time steps that should be calculated

- Make sure to validate the user input as something reasonable (more than 0 and less than 100), and if they enter an invalid number keep asking the user for new input until a valid input is given

Flags

- Often, we will want to end a loop only when multiple criteria is met
- The easiest way to do this is to check if each criteria is met within the loop, and if it is then set a boolean variable to True and if it is not met set that same boolean variable to False
- Do this every loop iteration, and then create a complex condition based on these boolean variables that can only be true or false

- For example, if we wanted to validate that the user input **`distance_input`** from our Universal Law of Gravity program as either “Far” or “Close” using flags, we might write the following

```
ask_for_distance = True
while ask_for_distance:
    distance_input = raw_input(“What distance?”)
    if distance_input == “Close” or distance_input == “Far”:
        ask_for_distance = False
    else:
        ask_for_distance = True
```

- `ask_for_distance` is the flag, and we loop as long as the flag is not set to False, and the only time it is set to false if when one of the valid inputs is input

- Note that in the last example, we started with our flag as True and changed it to False when the criteria was met

- This is simply for convenience, since we need the loop to run at least once (and the loop will only run on a true condition), we could have just as easily done this instead

```
distance_correct = False
while not distance correct:
    distance_input = raw_input("What distance?")
    if distance_input == "Close" or distance_input == "Far":
        distance_correct = True
    else:
        distance_correct = False
```

- The most important thing is to choose variable names that make sense, so when you come back the program later it still makes sense

- Let's apply the idea of using flags to validate input in our **`classification_tree.py`** program

- Before, if the user gave invalid input, we would simply tell them their input was invalid and end the program

- Now, tell the user that their input was invalid, and then prompt them again for another input

- Keep asking for input until the user gives a valid input

- Now take what you learned from the classification tree and try and edit your code for the universal law of gravity

- Your program already asks the user for a planet and distance input

- Now, when the user inputs an invalid planet or distance, the program should tell the user that their entry was invalid and ask for another input until they give a valid input

- Let's take a deeper look at the new program (remember, your program may be different)

- Most of the program is the same, we still verify the input for both the planet and the distance

- We did create a while loop and move the user input into this new while loop

- The main difference here is that instead of exiting the program when the input is invalid, we change `distance_incorrect` and/or `planet_incorrect` to True

- Now that we have these two boolean values that are changed to True whenever the user inputs something invalid, we can use that boolean value as the condition to our while loop

```
while planet_incorrect or distance_incorrect:
```

- This way, we know the loop will not end if either `planet_incorrect` **or** `distance_incorrect`

- Since the program does not finish the calculation till the while loop has concluded, we know it will keep asking the user for input until valid input is given

- Remember that these boolean variables that are used to affect a condition are called flags

- When this example was originally written, it had an **`and`** instead of an **`or`** in the condition

```
while planet_incorrect and distance_incorrect:
```

- Think about how this small change in the program would change how the program works

- The **`and`** makes it so that the program will only fail if both the **`planet_incorrect`** **and** the **`distance_incorrect`** flags are set to False

- This means that if either of the user inputs are correct, the the program will try and finish (and will crash since it will be missing a value)

- If we thoroughly test the program, this problem will become obvious pretty quickly

Testing

`while planet_incorrect and distance_incorrect:`

- To test this condition fully, there are only four options

`planet_incorrect`	`distance_incorrect`	**Condition Result**
True	True	**True**
False	True	**False**
True	False	**False**
False	False	**False**

Notice that this is every possible boolean combination

- And notice that the resulting table is simply a truth table

`planet_incorrect`	`distance_incorrect`	**Condition Result**	
True	True	**True**	(1)
False	True	**False**	(2)
True	False	**False**	(3)
False	False	**False**	(4)

- We can check column (1) by inputing a real planet (“Earth”) and a real distance (“Far”)
- Column (2) can be checked by inputting an incorrect planet (“Pluto”) and a real distance (“Far”)
- We can check column (3) by inputting a correct planet (“Earth”) and an incorrect distance (“Distant”)
- Column (4) can be checked by putting an invalid planet and an invalid distance (“Pluto” and “Distant”)

- You might think that this process would be impossible when we use other data types like integers or strings

- But we can always distill a condition down to the same table (although the size of that table may increase depending on the input)

- What would the table look like for this condition?

```
while (i < 5 and j > 3) or x <= 10:
```

```
while (i < 5 and j > 3) or x <= 10:
```

- Now there are many possible values for each variable, however remember that each each individual condition in the complex condition can only evaluate to True or False

`i < 5`	`i = 3` (True)	`i = 7` (False)
`j > 3`	`j = 5` (True)	`j = 1` (False)
`x <= 10`	`x = 10` (True)	`x = 12` (False)

`i < 5`	`i = 3` (True)	`i = 3` (True)	`i = 3` (True)	`i = 3` (True)	`i = 7` (False)	`i = 7` (False)	`i = 7` (False)	`i = 7` (False)
`j > 3`	`j = 5` (True)	`j = 5` (True)	`j = 1` (False)	`j = 1` (False)	`j = 5` (True)	`j = 5` (True)	`j = 1` (False)	`j = 1` (False)
`x <= 10`	`x = 10` (True)	`x = 12` (False)	`x = 10` (True)	`x = 12` (False)	`x = 10` (True)	`x = 12` (False)	`x = 10` (True)	`x = 12` (False)
Result	**True**	**True**	**True**	**False**	**True**	**False**	**True**	**False**

- Notice that this table makes up every combination of results for each condition
- If we run our program with each of these columns as input, then we will have fully tested this one condition
- Eventually there will be simply too many inputs for you to check them all, just check as many as you can and as you progress you'll start to learn which are more important to check

- Let's take a deeper look at some of the strings that we use in our universal law of gravity program

```
"Earth" "Venus"
```

- These two strings are actually two collections of characters (like a word is a collection of letters)

- This collection is called a list

Lists

- A list is an ordered collection of items, like a grocery list
- In python, it is an ordered set of objects, like integers, strings, booleans, etc.
- Strings are implicit lists (python does all the work to keep it in a list), but we can also make our own custom lists

Lists

- We would declare the string "**`Earth`**" as a true list instead of a string like this

```
earth = ["E", "a", "r", "t", "h"]
```

- This makes sense, as a string is an ordered set of characters, Python simply lets us declared it simply using quotes because strings are used so often in programming

- We can create lists of other types as well, such as strings

```
cities = [“Los Angeles”, “San Francisco”]
```

- integers

```
sequence = [4,8,15,16,23,42]
```

- and lists of other lists

```
lists = [cities, sequence]
```

Lists with Multiple Types

- We can even create lists that contain multiple data types

```
crazy_list = [“LA”, 4, True]
```

- Lists with multiple data types can get tricky when we go to access them

Also remember, True and False (capitalized) equate to their boolean values

Accessing Lists

- One way we access elements in a list is by using two square brackets with the index we want to access in the middle

```
x = cities[1]
```

- This is like saying give me the first item in the list

Indexing

- However, computer scientists don't start counting at 1, they start counting at 0
- So this list

```
sequence = [4,8,15,16,23,42]
```

- Would have these indices

Value	4	8	15	16	23	42
Index	0	1	2	3	4	5

- So if we wanted to iterate through a whole list, we would do the following

```
sequence = [4,8,15,16,23,42]
index = 0
while index < len(sequence):
    print sequence[index]
    index = index + 1
```

- This will print out every value in sequence one at a time, since **`sequence[index]`** returns the **`sequence`** list element at the **`index`** position and **`index`** increases from 0 to the length of **`sequence`**

- create an example called **`list.py`** and try it for yourself

- Lets start with **`len`**, **`len`** is a keyword function similar to **`raw_input`**, it returns the length of a list

- It requires a list as its only parameter, and returns the length of the list an an integer

```
s = “hello world”
length = len(s)
```

Would return 11
(10 letters + a space)

- Since strings are really arrays of characters, we can call **`len`** on a string to get the length of the word

- Now, we have already discussed how a while loop works, so lets look at the **`index`** variable

- We see that **`index`** is initially declared to 0 and the while loop will continue while the **`index`** variable is less than the length of the sequence

- Also notice that the index increases by one each iteration of the while loop (**`index = index + 1`**)

- Finally, the **`index`** variable is used to access into the sequence and print each value from beginning to end (using the square brackets)

```
sequence = [4,8,15,16,23,42]
index = 0
while index < len(sequence):
    print sequence[index]
    index = index + 1
```

- At the beginning `index` is 0 and `len(sequence)` is 6
- The while condition evaluates to true, so we execute the indented code
- We print sequence at location `index`, where `index` is 0, giving us the value `4` from `sequence` to print
- `index` is incremented by 1 (`index = index + 1`) and we go back to the while again to re-evaluate the condition

```
sequence = [4,8,15,16,23,42]
index = 0
while index < len(sequence):
    print sequence[index]
    index = index + 1
```

- This will repeat, with index increasing each time, until the while condition fails, giving us the following

iteration	index	condition evaluation	printed value
1	0	True	4
2	1	True	8
3	2	True	15
4	3	True	16
5	4	True	23
6	5	True	42
7	6	False	

Nothing prints since the condition failed

```
sequence = [4,8,15,16,23,42]
index = 1
while index < len(sequence):
    print sequence[index]
    index = index + 1
```

- Now, what would happen if we changed the starting value of index to 1 like above?

```
sequence = [4,8,15,16,23,42]
index = 1
while index < len(sequence):
    print sequence[index]
    index = index + 1
```

- Our program is still very similar to the original
- It will still end at the same point (when **`index`** is 5), however since it starts at 1 we will not print the first value in **`sequence`**
- Since the first value in **`sequence`** is indexed at position 0, and there is no point in the code where **`sequence[0]`** will be called, since index never equals 0

```
sequence = [4,8,15,16,23,42]
index = 0
while index < len(sequence)+1:
    print sequence[index]
    index = index + 1
```

- How about if we changed **`len(sequence)`** to **`len(sequence)+1`**?

```
sequence = [4,8,15,16,23,42]
index = 0
while index < len(sequence)+1:
    print sequence[index]
    index = index + 1
```

- Since there are only **`len(sequence)`** entries in the sequence, then when we try to access an entry past the end we will get an error

- This will cause a "**`list index out of range`**" error, since our index (the variable **`index`** in this case) is trying to access an element that is not in the values available in the list **`sequence`**

```
sequence = [4,8,15,16,23,42]
index = -1
while index < len(sequence):
    print sequence[index]
    index = index + 1
```

- What if we started our **`index`** at -1 instead of 0 or 1?

```
sequence = [4,8,15,16,23,42]
index = -1
while index < len(sequence):
    print sequence[index]
    index = index + 1
```

- In python, negative numbers wrap around to the back of a list instead of being an out of range error

- So in this case, starting with an index of `-1` will cause the last element (`42`) to print first

- Then as we increment **`index`** (it now has a value of 0), it will print the rest of the elements from beginning to end, finishing by printing `42` again at the end

Printing Lists Backwards

- Now you know that negative numbers wrap around to the back of lists instead of causing an error
- Use this information to take the list from the last example

```
sequence = [4,8,15,16,23,42]
```

and print it out backwards from end to beginning

```
sequence = [4,8,15,16,23,42]
index = 0
while len(sequence) - index >= 0:
  print sequence[len(sequence) - index]
  index = index + 1
```

Strange Condition

- Let's take a second to look at the condition in that last problem

```
while len(sequence) - index >= 0:
```

- Remember, we are printing backwards, so the value on the left of the >= must go from high to low

- Since length is always 0 or larger, as **`index`** becomes a larger number **`len(sequence) - index`** moves closer to 0

- You'll also notice that we change the way we index into **`sequence`**

```
sequence[len(sequence) - index]
```

- We have set it up so our condition moves from the length of **`sequence`** down to 0, and we also want **`sequence`** to be accessed in that order

- Because of that, it makes sense to use the whole left side of the condition as the index into **`sequence`**

- Lets apply our list skills to our classification tree
- Create a list of the possible inputs that the user can choose from, and then print that list out before the user inputs their choice

Dictionaries

- Another type of container in python is dictionaries
- Dictionaries are used to equate one thing to another (like dictionaries equate words to definitions)
- If we were to create a word dictionary in python, it would equate a string to another string

- The item you use to do the lookup is called the key (the word in our dictionary example)

- The item you get from looking up the key in the dictionary is called the value (the definition in our example)

Key	Value
“program"	“a plan of action to accomplish a specified end”
“python”	“any of several Old World boa constrictors”
“third”	“coming after second”

- Dictionaries are defined with curly braces

```
dictionary = {“one”: “1”, “two”: “2”, “three”: “3”}
```

- They can equate any type to any other type

```
dictionary = {“one”: 1, “two”: 2, “three”: 3}
```

```
dictionary = {1: “one”, 2: “two”, 3: “three”}
```

- And can even equate multiple types

```
dictionary = {“one”: “1”, 2: “two”, 3.14: True}
```

- Notice that this last example has strings, integers, and floating point numbers as keys

- So we could create a dictionary to translate strings that represent numbers into their integer value

```
numbers = {“one”: 1, “two”: 2, “three”: 3}
```

- We could use it to keep the mass and distances of the planets from our physics example

```
mass = {“earth”: 5.9736*(10**24), …}

dist = {“earth”: 147*(10**9), …}
```

- Or we can hold the translations of words

```
spanish = {“cat”: “gato”, “dog”: “perro”}
```

- We can even create more complex dictionaries, like one that uses a word as its key and returning a list of synonyms

```
synonyms = {“good”: [“acceptable”, “great”,
“exceptional”],“bad”: [“atrocious”, “awful”,
“crummy”, “dreadful”] }
```

- Remember, you use square brackets to define a list, so here the dictionary values are lists of strings

- We access a dictionary by using its key, and getting the value back

- We do this by putting square brackets after the name of the dictionary, with the key we want to look up in the square brackets

```
synonyms[“good”]
```

- You can see the example above looks for the key “good” and returns the value from the dictionary, which in this case is the following list

```
[“acceptable”, “great”, “exceptional”]
```

- Let's take the universal law of gravity example we have already written and make a couple of changes

- Create a dictionary that uses the planets name as a key, and has the mass of that planet as the value

- Create two more dictionaries, one for the distance from the planet to the Sun when it is "Close" and one for when it is "Far"

- Use these dictionaries in your if/elif/else blocks to assign values to `m_2` and `d` using the name of the planet as the key for each

- Using the dictionary in the code makes it easier to edit, since now we can change the values at the very beginning instead of having to hunt for the correct if/elif/else statement

- You may have noticed that all the planets are already listed as the keys in the dictionary, and that all our if and elif statements all look the same

- Wouldn’t it be great if there was some way to use they keys to validate the user’s input instead of having to type the planets into the dictionary as keys and in the if statement conditions?

Membership Operators

- We can do that with the `in` keyword
- `in` is a special keyword known as a membership operator
- membership operators determine if a variable exists in a collection such as a list

- For example, if we have the following list

```
evens = [2,4,6,8]
```

- We could check if the entry 4 exists in the list by using the `in` operator

```
if 4 in evens:
```

- Which would return the boolean value True, since 4 is an element in the evens list

- This is really good for checking if a particular value is in a list, without having to use a while loop to check through each value in that list

- We can also check if a value does **not** exist in a collection by using the `not` keyword before the `in` keyword

- Remember that `not` returns the opposite boolean value, so True will become False and False will become True

- So if we have the same list as before

```
evens = [2,4,6,8]
```

- We can check if a value like `3` does not exist in the list by using the keywords **`not`** and **`in`** together

```
if 3 not in evens:
```

- Since **`3 in evens`** would return False (since 3 is not in the events list), the **`not`** makes it return True (since True is the opposite of False)

- We can use the membership operator `in` on dictionaries as well

- It will return True if the left hand operator exists as a key in the dictionary, so for a dictionary like this

```
capitols = {“CA”: “Sacramento”, “AZ”: “Phoenix”}
```

- The following expressions would evaluate to True

```
if “CA” in capitols:
if “AZ” in capitols:
```

- But this would evaluate to False

```
if “Sacramento” in capitols
```

- We can apply dictionaries to our classification tree example
- Instead of using branching if/elif/else statements to check user input, use a dictionary and the membership operator
- Have the user input either “dry skin” or “moist skin” instead of having them enter “skin” and have a second question as we did before
- Make sure you are still validating user input

- Now look at the program we just wrote, is there any way to make it even shorter?
- What if we used the membership operator to check the dictionary directly in the condition of the while loop, instead of using a flag to do it?

- Lets also apply this to our universal law of gravity program
- We already have three dictionaries that use the planets as keys, and all three should have the same keys
- Try using the in keyword to validate the user inputs with one of the dictionaries

- You may have noticed that our dictionaries now have the mass and the “Close” and “Far” distances for every planet

- What if we wanted to print the gravitational pull between the sun and every planet instead of asking the user for the planet they want to calculate?

- You could try to iterate through the dictionary, but getting values from the dictionary requires the key, so we can't iterate with a number the way we normally do in a while loop

- We do know how to iterate through lists though, so let's create a list of all the planets and iterate through that

- Since the list has all the planets, and each planet is a key, you can use the entries in the list as keys in the dictionary

- If you are really clever, you may remember that the keys in a dictionary don't have to be strings
- To calculate the value for each planet, we can use integers as the keys in our mass and distance dictionaries instead of strings
- To print the planet we're calculating for, we can create one more dictionary that equates the integer key to the name of the planet as its value

- There is really no difference between this and having a few lists that have the information like the mass, distances, and names of the planets (as long as those lists are in the same order)

For Loops

- There is an even easier way to do this without having to change our dictionaries or create additional lists

- We can use another type of loop called a for loop

- For loops use the keyword **`for`** and are used to iterate through every item in a collection, like a list or a dictionary

```
planets = [“Earth”, “Mars”, “Mercury”,…]
```

- Given the above list

```
int index = 0
while index < len(planets):
    item = planets[index]
    print item
    index = index + 1
```

while loop

```
for item in planets:
    print item
```

equivalent for loop

- Both the for loop and the while loop above will do the exact same thing, print every element in the list

```
for item in planets:
    print item
```

- We start our for loop with the keyword `for`

- We then name the variable that we want the items from our collection saved into, in this case `item` but we can name it the same way we do any other variables, this variable is known as the **iterator**

- We use the keyword `in` to link the variable we are saving items from the container into (`item`), and the container we we are saving them from (`planets`)

- We then put which container we want to iterate through, in this case `planets`

For Loops and Dictionaries

- Now lists are simply a collection of items (in the last example, a collection of strings), but dictionaries are a collection of keys with an associated value
- What happens when we iterate through a dictionary using a for loop?
- Let’s find out with an example

```
numbers = {1: “one”, 2: “two”, 3: “three”}
for x in numbers:
    print x
```

- Create a new file named “`dictionary.py`“ and write the above code in that file

- Run that code and see what prints, will it be the integer values (1, 2, 3) which are the keys in the dictionary or the string values (“one”, “two”, “three”) which are the values in the dictionary?

- We can see from this example that when we create a for loop with a dictionary as a container, our iterator holds the value of the keys in the dictionary
- This is not a problem, since we can use that key to get the value from the dictionary

```
numbers = {1: “one”, 2: “two”, 3: “three”}
for x in numbers:
    print x
    print numbers[x]
```

- We print the key, which is stored into **`x`** when we iterate using the for loop

- We also print the value of the dictionary for that key, since **`numbers[x]`** is the way we would normally get the value from a dictionary using the key, and we have the key saved into **`x`**

- Now, let's go back to our universal law of gravity example
- Use for loops to iterate through the dictionaries, with the names of the planets as the keys, and calculating and print the force between each planet and the Sun
- Also, make sure you print which planet you are calculating before you print out the result

- Did you notice that we calculated the value of the force twice, only changing the distance value
- In fact, each iteration of the for loop we only change the distance and the mass of the planet
- It would be great if we could do the calculation without having to write the mass of the sun and the gravitational constant each time
- But programmers are all about efficiency, so of course we can do this with less code

Functions

- We can use something called a function to make it easy to re-use pieces of code that we are going to want to use over and over again in our program

- We use the keyword **`def`**, and define some inputs (called parameters) that our function will use to run some code which will change depending on the parameters input

- Let's look at an example

- Let's assume you wanted to write a program to calculate and print the slope of a line

- We can use the slope-intercept equation

```
y = mx + b
```

- And do some algebra to get an equation to solve for the slope (`m`)

```
m = (y - b) / x
```

- Now we can see from this equation that we need the x value (`x`), y value (`y`), and y-intercept (`b`) to solve for the slope

- These will be our parameters for our function, since we need to know each of these things every time we want to calculate the slope

- That means our function will begin…

```
def slope(x, y, b):
```

- We know what we want to calculate as well, which will make up the body of our function

```
def slope(x, y, b):
    m = (y - b) / x
    print m
```

- Now, we can call the function **`slope`**, pass in the proper parameters, and the function will automatically do the calculation and print the result

```
slope(2.0, 4.0, 1.0)
```

We use floats here instead of integers so python doesn't round the result automatically

- So let's take a deeper look at the function we have just written

```
def slope(x, y, b):
    m = (y - b) / x
    print m
```

- Notice that the parameters match the variables in the function

- This is because whatever we input into the function parameters will replace the matching variable in the rest of the function

- If we write the following code after defining our function (it's important that we define our function before it appears in our code)

```
x = 5.0
y = 7.0
b = 2.0
slope(x, y, b)
```

- Then when the function is executed, it will be automatically rewritten to be this

```
def slope(5.0, 7.0, 2.0):
    m = (7.0 - 2.0) / 5.0
    print m
```

- Functions are great, because we can reuse them with different parameters as many times as we want

```
slope(3.0, 5.0, 2.0)
slope(6.0, 4.0, 1.0)
slope(5.0, 4.0, 3.0)
slope(9.0, 6.0, 3.0)
slope(7.0, 5.0, 2.0)
```

- And we can get different results without having to re-type our calculation with different values each time

- So now lets apply this to our universal law of gravity example
- We want to take our equation for calculating force between planets, and write it as a function so we can use it to make the two calculations
- Think about which variables we need to pass in as parameters, and which we can define within our functions

- Now what if, instead of printing the result, we wanted to save it to another variable?

- We would use the keyword **`return`**, which means that when the function completes whatever is returned will be the result of the function, which can be saved to a variable

- This works the same way that the function **`raw_input`** returns the string that the user inputs

- So for example…

```
def slope(x, y, b):
    m = (y - b) / x
    return m
```

- Would now return the value of the slope instead of printing it

- This means we could use the result in other calculations or later in our program

```
m = slope(5.0, 7.0, 2.0)
```

- Now you have learned all the basic concepts of the python programming language
- You should practice these skills by thinking of problems and trying to solve them using python, or by working through the examples of other online python tutorials
- When you are ready to move on, there are lots of great resources online you can use to continue to learn more about python