Name: _____

Week 2:
Input, Comparisons, Loops, and If Statements

Input

When we want to read in information from the user, we use a special function, or command, called `input`. `input` will wait for the user to type some text and hit the enter key, then effectively replace itself in the code with whatever the user entered. We call the `input` function by typing `input`, an opening parenthesis '(', and then a close parenthesis ')'.

For example,

```
print("Please enter your name!")
string_foo = input()
print("Your name is " + string_foo)
```

Would first ask the user for their name, then wait. It would look like the program had simply stopped doing anything, but it's actually waiting for you to type something. If we were to type in "Oliver Otis Howard", then `string_foo` would get the value "Oliver Otis Howard". The next line would then print "Your name is Oliver Otis Howard".

Now, one nice trick with `input()` is that since it's so common to want to print out a message telling the user what to enter, we can just put that message in the parentheses of `input()`, and Python will print it out for us. So this:

```
input("Please enter your name!")
```

is the same as this:

```
print("Please enter your name!")
input()
```

with one small exception - the `print()` function will print a newline character ('\n') after the message, so the input will start on the next line. When you put the message in the `input()` function itself, the input will be collected on the same line. This won't matter very often, but it's worth noting.

Example Problems:

`` int_age = int(input()) print("Twice your age is " + str(int_age * 2)) ```	What will this code output if the user enters “23”?
`` float_cost = float(input()) float_cost_in_yen = float_cost * 111.24 print("Cost in yen: " + str(float_cost_in_yen)) ```	What will this code output if the user enters “5.05”?
Write some code that will ask the user for their school name, and then print out “Your school is $SCHOOL_NAME!” (Remember the $SCHOOL_NAME syntax is just CS shorthand to tell us that the school name should go there.)	

Comparisons

Motivation

Before we can get to the fun stuff with loops and if statements, we need to briefly touch on comparisons. Comparison operators are operators that work on non-boolean values, but give boolean results. This is useful because it lets us ask yes/no questions about things that are not already booleans, such as asking if a number is smaller than another number, or if the user input a specific string. The comparisons we have available are as follows:

==	“Is Equal To”. This comparator tells us if objects are equal. It will return `True` if the object on the left and the object on the right are equal, and `False` otherwise. Note this uses **two** equal signs! If you just use one, you’ll get an error.	Examples: ``` 10 == 11 # False 10 = 11 # SyntaxError: can't assign to literal 10 == 10 # True ```

		`"Hello" == "hello" # False` `"hello" == "hello" # True`
!=	“Is Not Equal To”. This comparator tells us if objects are not equal. It will always be the opposite of `==`.	Examples: `10 != 11 # True` `10 != 10 # False` `"Hello" != "hello" # True` `"hello" != "hello" # False`
>	“Is Greater Than”. This comparator tells us if the left object is greater than the right object.	Examples: `10 > 11 # False` `10 > 10 # False` `10 > 9 # True`
<	“Is Less Than”. This comparator tells us if the left object is smaller than the right object.	Examples: `10 < 11 # True` `10 < 10 # False` `10 < 9 # False`
>=	“Is Greater Than Or Equal To”. This comparator tells us if the left object is larger than, or the same as, the right object.	Examples: `10 >= 11 # False` `10 >= 10 # True` `10 >= 9 # True`
<=	“Is Less Than Or Equal To”. This comparator tells us if the left object is smaller than, or the same as, the right object.	Examples: `10 <= 11 # True`

		`10 <= 10 # True` `10 <= 9 # False`

There are more comparators we will encounter as we learn more about Python, but these are enough to start!

One last note: Any operations on either side of a comparator will be done **before** comparing. So `10 == 9+1` will be `True`, because we will first add `9+1`, and then check the `==`.

Example Problems:

77 > 76 -> _____

77 == 76 -> _____

77 != 76 -> _____

77 < 76 -> _____

77 >= 76 -> _____

77 <= 76 -> _____

77 > 77 -> _____

77 == 77 -> _____

77 != 77 -> _____

77 < 77 -> _____

77 >= 77 -> _____

77 <= 77 -> _____

77 > 78 -> _____

77 == 78 -> _____

77 != 78 -> _____

77 < 78 -> _____

77 >= 78 -> _____

77 <= 78 -> _____

77 == 76 + 1 -> _____

“howard” == “howard!” -> _____

“howard” == “how” + “ard” -> _____

“hahaha” == “ha” * 3 -> _____

“hahaha” != “ha” * 3 -> _____

Loops

Motivation

We’ve learned how to take input from the user, and how to print things back to the user. However, lots of programs like Messenger, What’s App, and other chat apps not only take input and write things out, but they do it over and over again until you close the app. Now, those programmers probably didn’t write out thousand of `input()` statements, so how do we tell the computer to do something repeatedly?

Let’s look at some code that reads in the user’s name, and prints out a message:

```
print("What is your name?")
string_name = input()
print("Hello " + string_name)
```

Now, if we wanted to do this repeatedly, perhaps to greet a whole bunch of people, we could copy the code, so that it would look like this:

```
print("What is your name?")
string_name = input()
print("Hello " + string_name)
print("What is your name?")
string_name = input()
```

```
print("Hello " + string_name)
```

But that only works for the number of times we've copied. Instead, we want a way to make the code we've already written execute again. Conceptually, we want something like this:

```
START_OF_CODE <----------------
print("What is your name?")    |
string_name = input()          |
print("Hello " + string_name)  |
GO TO START_OF_CODE -----------|
```

The good news is that programming gives us a way to do just that! It's called the `while` loop. `while` loops will repeat any indented lines following them. They work like this:

```
while True: # This is like our START_OF_CODE line above.
    # Code is indented by four spaces to show we're in the while loop!
    print("What is your name?")
    string_name = input()
    print("Hello " + string_name)
# Code becomes un-indented here, to show the while loop is done.
# This is where our GO TO START_OF_CODE line from above would go.
print("Code here is not in the loop any more.")
```

Let's examine that in more detail.

The word `while` on the first line works like our START_OF_CODE marker in our conceptual example. It tells Python that we're going to want to repeat what follows, and how far back it should go when it reaches the end of the repeated part.

The code that we want to repeat is indented by 4 spaces. (Always! Don't forget this!) This tells Python which lines should be repeated. The last `print` statement in our example, for instance, is not indented, so it will not be repeated.

Now, why does it say `while True`? Why not just `while`? The reason is that sometimes we don't want to repeat something forever. As an example, let's try to write a password checker. We want to ask the user to enter their password, and to keep asking them until they get it right. What we want, conceptually, is something like this:

```
string_password = "hunter2"
string_user_guess = ""
START_OF_REPEATED_CODE <-------------------------------------
string_user_guess = input("Enter password:")                  |
GO TO START_OF_REPEATED_CODE IF string_user_guess IS WRONG ---
print("Access granted!")
```

In order to accomplish this, we need our while loop to be able to check if `string_user_guess` is wrong. How do we do that? Well, remember where we have `while True` in the last example? It turns out that the `True` part tells Python how long to keep repeating the indented code in the `while` loop. The code will be repeated as long as the **boolean expression** following the word `while` evaluates to `True` - in the last example, since we wanted it to repeat forever, we just used the boolean value `True`, which is always `True`.

In this case though, we want to repeat the code asking them for the password as long as they have got it wrong. As soon as they get it right, we want to stop asking them for it, and grant them access! That means we will need the value after the `while` to be `True` when the user has the password wrong, and `False` when the password is correct. We can do that like this:

```
string_password = "hunter2"
string_user_guess = ""
while string_user_guess != string_password:
    # Indent by 4 spaces
    string_user_guess = input()
print("Access granted!") # No more indent
```

Like the previous example, we see we have the `while` statement telling us we're going to repeat code, and the indent before the `string_user_guess = input()` line to tell us that line will be repeated. However, instead of `True` after our `while`, we have `string_user_guess != string_password`, so that we can stop when `string_user_guess` is correct. How does it know when to stop? Well, whenever the program reaches a `while` statement, it checks the boolean condition following it. If the condition is `True`, it will execute each of the indented lines in order, and then go back to the line with the `while` statement on it and check the boolean condition again. If the condition is `False`, it simply skips the indented lines, and executes the first non-indented line after the `while`. Let's go through this example line-by-line, to see that in action:

Line being executed:	Explanation:
`string_password = "hunter2"`	Value "super secret ultra-secure password" is stored into variable `string_password`.
`string_user_guess = ""`	The empty string value is stored into the variable `string_user_guess`. We need an initial, wrong, guess in order to enter the loop. If we set this to "hunter2", then the password would be correct on

	the first try and skip the loop!
`while string_user_guess != string_password:`	Our while statement! The computer first checks the boolean expression. The empty string “” is not equal to the value in `string_password`, “hunter2”, so `string_user_guess != string_password` is `True`. Since the boolean condition is `True`, the program executes the indented code.
`    string_user_guess = input()`	The variable `string_user_guess` stores the value the user types in. In this case, let’s say that’s “swordfish”.
`print("Access granted!") # No more indent`	This line is **not** executed, but the program sees that we’ve reached an un-indented line, which means we’ve executed all the lines we were supposed to repeat. We’ll need to see if we’re going to want to repeat again, so it jumps back to the `while` statement. Note that the first non-indented line is **not** executed. The program notes the lack of indent, and that tells it to go back to the `while` statement, but it will **not** print anything.
`while string_user_guess != string_password:`	`string_user_guess` is “swordfish”, which is not equal to the value of `string_password`, “hunter2”, so `string_user_guess != string_password` is `True`. The program goes to repeat the indented code again.
`    string_user_guess = input()`	The variable `string_user_guess` stores the value the user types in. In this case, let’s say that’s “hunter2”.
`print("Access granted!") # No more indent`	We reach the end of the indent, so the program knows to go back to the `while` statement and see if we need to repeat again. Remember, this is **not** executed! We **only** look at the lack of indent to tell us that we’re at the end of the repeated statments!
`while string_user_guess != string_password:`	`string_user_guess` is “hunter2” which **is** equal to the value in `password`, “hunter2”, so `string_user_guess != string_password` is `False`. Since the boolean expression is now `False`, we **skip** all of the indented lines, and go to execute

	the first non-indented line after the `while` statment.
`print("Access granted!") # No more indent`	This time we **do** execute this statement, because the `while` loop's boolean expression was `False`, and the loop is done repeating the indented code. We print "Access granted!" and the program terminates.

Before we move on to the problems, there's a few more things to note. First, notice how we skip the indented code as soon as the boolean expression is `False`? This means that if the expression is `False` the first time we check it, we will never execute the code in the while loop! In other words, if you write a while loop with an initially `False` condition, it's possible for the program to skip the indented code entirely, and just move on.

The other thing to note is that some `while` loops can sometimes go on forever. For instance, if we write:

```
int_counter = 0
while int_counter < 10:
    print(str(int_counter))
int_counter += 1
```

the loop will never end! We accidentally put the increase to our counter outside the loop, so the counter will remain 0 forever, and the loop will just keep outputting 0 over and over. When this happens, hit control-C to stop your program. (Note this is always control-C, even on Macs. Command-C won't work!)

Example Problems:

`int bool_should_continue = True` `while bool_should_continue:` `  bool_should_continue = False` `  print("Loop ran!")`	What does this code print?
`int_counter = 0` `while int_counter < 10:` `  print(str(int_counter))` `  int_counter = int_counter + 1`	What does this code print?

How would we repeat until the variable `string_user_guess` was equal to the variable `string_password`? (Hint: It's in the example above!)	`while _____:`
How would we repeat as long as the variable `int_num` was less than 10?	`while _____:`
How would we repeat until the variable `string_name` was "Howard" **and** the variable `string_location` was "DC"?	`while _____:`
How would we repeat as long as the variable `int_foo` was either greater than ten, **or** equal to zero?	`while _____:`
How would we print "Incorrect!" until the user enters the value in `int_secret_number`? (Hint: You'll need to create a variable for the user guess, like in our example above!)	`int_secret_number = 12` `_____` `while _____:` `    print(_____)` `    _____ = input()`
How would we print the numbers from 0 to 9 using the variable `int_count`? (Don't forget to convert to string before printing!)	`int_count = 0` `while _____:` `    print(_____)` `    int_count = int_count + 1`
How would we print the numbers from 10 to 1, counting down, using the variable `int_backwards_count`? (Hint: It's a lot like the last one!)	`int_backwards_count = 10` `# Your code below!`
How would we print "Hello, world!" five times, and then print out "Done!"?	`int_times_printed = 0` `while _____:` `    print(_____)`

	`    int_times_printed = int_times_printed + 1` `print(_____)`
How would we get the user's name, then print it five times?	`string_user_name = _____` `# Your code below!`
How would we get a number from the user, add it to the variable `total` five times, then print out only the final sum? (Yes, we could use multiplication, good catch! But only use addition here, for practice.)	`string_to_add = input()` `int_total = 0` `int_times_added = 0` `while _____:` `    _____` `    _____` `print(_____)`
How would we ask the user for a number, square it twice, and then print out the result? (Don't forget your value type conversions!)	`# Your code below!`
How would we ask the user for five numbers, and print out the total?	`int_total = 0` `_____ = _____` `while _____:` `    _____ = _____` `    int_total = _____`

	`_____`
Bonus: How would ask the user to enter a word, then to keep entering more words until they enter the first word again?	`# Your code below!`

If Statements

The `if` statement

Let's expand on the password example from above. Suppose we want to print out "ACCESS DENIED" whenever a user enters the wrong password. Conceptually, we would want something like this:

```
string_password = "hunter2"
string_user_guess = ""
START_OF_REPEATED_CODE ←-------------------------------------
string_user_guess = input()                                  |
IF string_user_guess IS WRONG print("ACCESS DENIED")         |
GO TO START_OF_REPEATED_CODE IF string_user_guess IS WRONG ---
print("Access granted!")
print("Now showing you secret information...")
```

We already know how to make the code repeat, but what about making different code execute based off of a conditional? Well, the tool we use for that is called the `if` statement. It consists of the word `if`, then a boolean expression, then a colon ':'.

The lines we want to execute conditionally are indented by 4 spaces than the `if` statement, to make sure the program can tell which lines we want to conditionally execute. This means that if the `if` statement is not indented at all, the lines should be indented 4 spaces. On the other hand, if the `if` is already indented by 4 spaces (due to already being inside the indented code for a while loop, for instance), the lines should be indented by 8, and so on and so forth. This is something that will remain consistent in Python - when you use a statement that requires

indentation to know what lines it should affect, those lines should always be indented by 4 more spaces than the line affecting them.

Let's use the password example again to demonstrate this. Suppose we only wanted to give the user one chance to get the password right, so we didn't want to use the `while` loop. We could instead use an `if` statement like this:

```
string_password = "hunter2"
string_user_guess = input("What's the password?")
if string_user_guess == string_password:
    print("Access granted!")
```

This code will print "Access granted!" if the user enters the correct password, and do nothing if they enter the wrong password.

We can also nest `if` statements, so it's completely valid to write:

```
string_password_1 = "hunter2"
string_user_guess_1 = input("What's the first password?")
if string_user_guess_1 = string_password_1:
    string_password_2 = "hunter1"
    string_user_guess_2 = input("What's the second password?")
    if string_user_guess_2 == string_password_2:
        print("Access granted to system 2!) # Indent 8 spaces
    print("Access granted to system 1!") # Indent 4 spaces - we're out of the inner
if.
print("Goodbye!")
```

This will ask the user for the first password. If they get it wrong, the program will skip everything in the if statements, and go straight to printing "Goodbye!". If they get it right, it will then ask them for the second password. If they get that one right too, it will print out "Access granted to system 2!". After asking for the second password, and possibly printing out the system 2 message, it will print out "Access granted to system 1!" Note that this will be printed out regardless of if they get password 2 correct - the system 1 print statement is indented by 4 spaces, which means it's not in the second `if` statement any more, and is only dependent on the result of the first `if` statement. After that, the program will print "Goodbye!", and terminate.

Note that when we nest `if` statements, we have to add 4 spaces of indentation for each layer of `if`s that we have! We show 4 and 8 above, but you can nest if statements as deep as you want.

Example Problems:

`int_age = 40` `if int_age < 90:`	What does this code print?

`    print("Not too old!")`	
`string_name = "Oliver"` `if name == "oliver":` `    print("hello oliver")` `print("Goodbye!)`	What does this code print? (Hint: Capitalization!)
`int_chair_weight = 10` `int_couch_weight = 50` `if chair_weight < 25:` `    if int_couch_weight < 100:` `        print("That will fit in my truck!")`	What does this code print?
`int_chair_weight = 10` `int_couch_weight = 500` `if chair_weight < 25:` `    if int_couch_weight < 100:` `        print("That will fit in my truck!")`	What does this code print?
`int_chair_weight = 50` `int_couch_weight = 50` `if chair_weight < 25:` `    if int_couch_weight < 100:` `        print("That will fit in my truck!")`	What does this code print?
Write code using an `if` statement that will ask the user for their age, and print out “You are an adult!” if they are at least 18. (Hint: Remember the difference between > and >=)	int_user_age = _____ if _____: _____

Write code using an `if` statement that will ask the user for the number of pets they have, and print “Wow, that’s a lot of pets!” if it’s over 3.	# Your code here:

The `else` statement

Let’s think about our password example again. Suppose we only wanted to give the user one chance to guess the password, then simply print out either “Access granted!” or “ACCESS DENIED.”, based on if they got it right? Conceptually, this is what we want:

```
string_password = "hunter2"
string_user_guess = input()
IF string_user_guess IS RIGHT print("Access granted!")
OTHERWISE print("ACCESS_DENIED.")
```

Python uses the `else` statement to let us do this. The `else` statement must always come after the indented code following an `if` statement, and the `else` is always at the same level of indentation as the `if` it corresponds to. Additionally, there must be **no** un-indented code in between the end of the indented `if` statements and the `else`. `else` is very easy once you’ve put it in the right place though - it consists of just the word `else`, and a colon ‘:’. We then indent the code we want in the `else` by 4 spaces, just like we did with the `if` and `while`.
The Python code for our above example, then, looks like this:

```
string_password = "hunter2"
string_user_guess = input()
if string_user_guess == string_password:
    print("Access granted!")
# No un-indented statements can go here! (Except comments, they don't count.)
else: # Same level of indent as the if above.
    print("ACCESS DENIED.") # Indented by 4 spaces, just like with if.
```

This code works just like our conceptual code, it will let the user enter one guess, and then print “Access granted!” if their guess is correct, and “ACCESS DENIED.” if it is wrong.

We will never print both statements, because if we run the `if`, we will skip the `else`, and if we skip the `if`, we will run the `else`. Whenever you see an `if` followed by an `else`, you can be sure that exactly **one** of those indented blocks of code will be run.

Example Problems:

`` bool_should_run_if = True if bool_should_run_if: print("If ran!") else: print("Else ran!") ```	What does this code print?
`` string_name = "Alexander" if string_name == "Hamilton": print("Your name is Hamilton!") else: print("Your name is not Hamilton.") ```	What does this code print?
Write some code that uses `if` and `else` to ask the user for a number, and print out “IT’S OVER 9000” if the number is at least 9001, and “That’s a nice number” otherwise.	int_number = _____ if _____: _____ else: _____
Write some code that uses `if` and `else` to ask the user for their favorite color. If it’s the same as your favorite color, print out “That’s my favorite color too.” If it’s different, print out a message telling the user what your favorite color is.	# Your code here:

The `elif` statement

Now, let's say people have been forgetting the '2' in our password a lot. Maybe we want to remind them, and give them a nicer message than "ACCESS DENIED". If they type in "hunter", we want to print out "Almost, but not quite!" How would we do that? Well, one way would be like this:

```
string_password = "hunter2"
string_user_guess = input()
if string_user_guess == string_password:
    print("Access granted!")
else:
# Indented 0 spaces.
    if string_user_guess == "hunter":                    # Indented 4 spaces.
        print("Almost, but not quite!")                  # Indented 8 spaces.
    else:                                                # Indented 4 spaces.
        print("ACCESS DENIED.")                          # Indented 8 spaces.
```

We can put `if` and `else` statements inside other `if` and `else` statements! But while this is cool, it's a little messy. Fortunately because this sort of problem comes up a lot, Python gives us a special statement called `elif` that will let us avoid nesting `if` statements sometimes.

`elif` is basically an `if` statement that follows another `if` statement, but with the catch that it will only be executed if the first `if` statement is `False`. So we could write the above code like this:

```
string_password = "hunter2"
string_user_guess = input()
if string_user_guess == string_password:                 # Indented 0 spaces.
    print("Access granted!")                             # Indented 4 spaces.
elif string_user_guess == "hunter":                      # Indented 0 spaces.
    print("Almost, but not quite!")                      # Indented 4 spaces.
else:                                                    # Indented 0 spaces.
    print("ACCESS DENIED.")                              # Indented 4 spaces.
```

This code will do just what we want! If the user enters "hunter2", it will print "Access granted!". If they enter "hunter", it will print "Almost, but not quite!". And if they enter anything else, it will just print "ACCESS DENIED."

One nice thing about `elif`s is that we can have more than one of them. So if we wanted to print another special message if they guessed, say, "hunter1", we could simply add another `elif` after the one we've already written.

Up above, I mentioned that `elif` will only be checked if the `if` is `False` - this is because, just like how we only ever run the `if` or the `else`, when we have an `if`, some `elif`s, and an `else`, we will only ever execute the statements after one of them. If we run the `if`, we will skip all the `elif`s and the else. And if one of the `elif`s runs, it will skip the rest, and also skip the else.

As an example, consider the following code:

```
int_temperature = int(input("What's the temperature outside?"))
if int_temperature < 40:
    print("Brr that's cold!")
elif int_temperature < 70:
    print("That's a bit chilly!")
elif int_temperature < 90:
    print("That's pretty nice!")
elif int_temperature < 110:
    print("That's pretty hot!")
else:
    print("That's dangerously hot!")
```

Suppose we enter the temperature “75”. (Note `input()` will always take in a String, hence the `int()` conversion on the first line.) Once that’s been converted and stored into `int_temperature`, we’ll go through the `if`/`elif`/`else` statements.

First, we’ll see if `int_temperature` is less than 40. It’s not, so we skip the indented code, and continue on to the next `elif`.
The `elif` checks if `int_temperature` is less than 70. It’s not, so we skip the indented code and continue on to the next `elif`.
This `elif` checks if `int_temperature is less than 90. It is! So we run the indented code and print out “That’s pretty nice!”. Then, we skip all the rest of the` elif `statements, as well as the` else` statement.

Since we skip the rest of the statements in an `if`/`elif`/`else` chain once one is executed, that let us do something clever above. Look at the last `elif` statement. If all, if we know that the temperature is less than 110, we can’t say for sure that the day is hot. After all, 109 is less than 110, but so is 9! However, since `elif`s skip the rest of the chain once they execute, we know that if the last `elif` is being executed, that means the second to last `elif` was `False` - and that means that the temperature must also be above 90! With that extra information, we can safely say it is hot out.

Now, notice that this all depends on the statements executing in order. If we wrote the code like this instead:

```
int_temperature = int(input("What's the temperature outside?"))
if int_temperature < 40:
    print("Brr that's cold!")
elif int_temperature < 110:
    print("That's pretty hot!")
elif int_temperature < 70:
    print("That's a bit chilly!")
elif int_temperature < 90:
    print("That's pretty nice!")
```

```
else:
    print("That's dangerously hot!")
```

we would always check if the temperature was less than 110 first. That means we could never execute the code in the other `elif`s, since any temperature that's less than 110 must also be less than 70 and less than 90! Order matters with `elif`s, and they will always be checked from the top down.

Example Problems:

What will the following code output?	`string_name = "Gabe"` `if string_name == "Howard":` `  print("That's the name of the school!")` `elif string_name == "Gabe":` `  print("That's the name of the instructor!")` `else:` `  print("That's not a name I recognize!")`
What will the following code output?	`int_age = 0` `if int_age < 12:` `  print("Still a child!")` `elif int_age < 18:` `  print("Teenager!")` `elif int_age < 40:` `  print("Young adult!")` `elif int_age < 60:` `  print("Middle-aged!")` `else:` `  print("Probably getting senior discounts!")`
What will the following code output?	`int_age = 25` `if int_age < 12:` `  print("Still a child!")` `elif int_age < 18:` `  print("Teenager!")` `elif int_age < 40:` `  print("Young adult!")`

	`` elif int_age < 60: print("Middle-aged!") else: print("Probably getting senior discounts!") ```
What will the following code output?	``` int_age = 0 if int_age < 12: print("Still a child!") elif int_age < 60: print("Middle-aged!") elif int_age < 18: print("Teenager!") elif int_age < 40: print("Young adult!") else: print("Probably getting senior discounts!") ```
What will the following code output?	``` int_temperature = 25 if int_age < 12: print("Still a child!") elif int_age < 60: print("Middle-aged!") elif int_age < 18: print("Teenager!") elif int_age < 40: print("Young adult!") else: print("Probably getting senior discounts!") ```
Write some code to take in a rating for a movie, from 1 to 5, and output “Great!”, “Good”, “Okay”, “Poor” or “Awful”, based on the rating.	int_rating = _____ if int_rating == _____: _____ elif _____:

	_____ elif _____: _____ elif _____: _____ else _____: _____
Write some code to take in a color from the user. Print out either “That’s my favorite color!”, “That’s my second favorite color.”, or “That color isn’t one of my favorites.”, inserting the names of your favorite colors into the conditionals.	# Your code here:

Combining `if` and `while`

`if` and `while` statements are useful on their own, but they’re even more useful when we combine them together. Going back to our password example, let’s say we wanted to let the user guess over and over, but also wanted to print out “ACCESS DENIED.” whenever they got the password wrong. We can accomplish this by using the `if` statement inside the `while` loop like this: (Note the extra indentation due to the `if` statement being inside the `while` loop.)

```
string_password = "hunter2"
string_user_guess = ""
while string_user_guess != string_password:
    # Indent by 4 more spaces than the while line.
    string_user_guess = input()
    if string_user_guess != password:
        # Indent by 4 more spaces than the if line.
        print("ACCESS DENIED.")
    # When we're done with the if statement we go back to the previous indentation.
    # There aren't any more lines here this time, but you can leave a comment like
    # "End of if statement" if you want to be extra clear about where they end.
print("Access granted!") # No more indent
```

This code will work just like our conceptual code. When we reach the `if` statement, if the conditional is `True`, then the indented code will be executed. Otherwise, it will be skipped, just like how we skip the code in the `while` loop when the conditional there evaluates to `False`.

Let's look over how this code will execute.

`string_password = "hunter2"`	Set our password
`string_user_guess = ""`	We need an initial wrong user guess, so that we can enter our loop, so we just set it to the empty string.
`while string_user_guess != string_password:`	Tell Python we want to repeat the following indented code as long as `string_user_guess` is not equal to `string_password`.
`    string_user_guess = input()`	Get a password guess from the user. Let's assume they enter "password1".
`    if string_user_guess != string_password:`	First, we evaluate `string_user_guess != string_password`. Since "password1" is different from "hunter2", this is `True`. That means we will execute the indented code following the `if`

	statement.
`print("ACCESS DENIED.")`	Yell at the user. Prints out “ACCESS DENIED.”
`print("Access granted!")`	Reach an un-indented line - in this case, it is indented by 8 spaces less than our last line, which tells Python we’re exiting two different blocks - 4 spaces less mean we’re not in the `if` statement any more, and 4 more mean we’re also not in the `while` loop any more. Since we’ve found the first statement outside of the `while` loop, we have to go back to the `while` statement and evaluate it again.
`while string_user_guess != string_password:`	`string_user_guess` is “password1” and `string_password` is “hunter2”, so our conditional is `True`, and we repeat the indented code in the `while` loop again.
`string_user_guess = input()`	Read in from the user. Let’s assume the user types in “hunter2” this time.
`if string_user_guess != string_password:`	`string_user_guess` is “hunter2”, and `string_password` is also “hunter2”, so the conditional is `False`. Since `if` sees a `False`, it will skip the next block of indented lines.
`print("ACCESS DENIED.")`	We skip this line, because it’s indented more than our `if`, and our `if` condition evaluated to `False`. Nothing is printed.
`print("Access granted!")`	Reach an un-indented line. Just like before, the 8 spaces less indentation tell us we’re out of the `if` statement, and also out of the `while` loop. Since we’re out of the `while` loop, we have to go back and re-evaluate the `while` line.
`while string_user_guess != string_password:`	`string_user_guess` is “hunter2” and `string_password` is “hunter2”, so this evaluates to `False`. That means we skip the whole next block of lines indented by more than the `while` statement, and go do the next line with the same indentation as the `while`.

`print("Access granted!")`	We reach this line and since the `while` loop is done, we don't have to go back this time. We simply evaluate this line, and print "Access granted!" to the user.

Example Problems:

What will the following code output?	`int_counter = 0` `while int_counter < 10:` `  if (int_counter % 2) == 0:` `  print(str(int_counter))` `  int_counter = int_counter + 1`
What will the following code output?	`int_counter = 0` `if (int_counter % 2) == 0:` `  while int_counter < 10:` `  print(str(int_counter))` `  int_counter = int_counter + 1`
Write some code that will print out the numbers from 0 to 10. Make the code also print out "Nearly done!" when there are 3 or less numbers left to print. (Hint: Don't forget - we want to go to 10, not to 9! Missing the last number is a common mistake in computer science, often caused by mixing up < and <=, and is a case of what is called an "off by one error".)	int_counter = _____ while _____: print(_____) if _____: print(_____) int_counter = _____
Write some code that will take in a number from the user and, if that number is positive, print out all the numbers from 0 up to that number.	# Your code goes here:
