Lesson 2: Conditional Statements & Boolean Logic

In the programs we have seen till now, there has always been a series of statements faithfully executed by Python in exact top-down order. What if you wanted to change the flow of how it works? For example, you want the program to take some decisions and do different things depending on different situations, such as printing 'Good Morning' or 'Good Evening' depending on the time of the day?

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability.

In order to check a condition, we need to be able to compare different values and determine whether the condition is true or false. To do these actions, we first need to know about **Boolean expressions**, **comparison operators** and **logical operators**.

Boolean Expressions

A **Boolean expression** is an expression that is either true or false. For example:

```
5 equals 5
```

This statement would evaluate as **True**. The number five *does* equal the number five. How about this one:

```
5 equals 6
```

This statement would evaluate as **False**. The number 5 is not the same value as the number 6.

When writing in code, Python uses the comparison operator == to represent the word "equals." Our two examples above may be re-written in Python as:

Code	Evaluates As
`5 == 5`	`True`
`5 == 6`	`False`

Note the capitalization of the names. **True** and **False** are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<class 'bool'>

>>> type(False)
<class 'bool'>
```

Boolean Data Types

You can also assign a variable directly to the data type of `bool`:

Code	Evaluates As
`is_enabled = True`	`True`
`my_variable = False`	`False`

Go online to complete the
Boolean Data Types Challenge

Comparison Operators

The equals operator == is one of several **comparison operators**. We use comparison operators to compare different objects – be it strings, numbers, integers, variables, or other data types. Here are the most common:

== (equal to)

This compares whether the objects are equal. `x == y` means that x is equal to y. When you use `x == y`, you are asking the question "*Is x equal to y?*".

A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an *assignment* operator and == is a *comparison* operator.

Code	Output	Notes

`x = 2`
`y = 2`
`x == y`
	`True`	Both x and y equal 2, so the return value of `x == y` is True.

`x = 'str'`
`y = 'stR'`
`x == y`
	`False`	Remember that strings are case sensitive. `str` is not the same as `stR`. So when we compare x and y, they are not equal, so `x == y` evaluates to false.

`x = 'str'`
`y = 'str'`
`x == y`
	`True`	Both values are the same, so `x == y` evaluates to true.

Go online to complete the
Equal To Challenge

!= (not equal to)

This compares whether the objects are not equal. `x != y` means that x is not equal to y. When you use `x != y`, you are asking the question "*Is x not equal to y?*".

Code	Output	Notes

`x = 2`
`y = 3`
`x != y`
	`True`	The numbers 2 and 3 are different. So the return value of `x != y` is True. Remember, we are checking whether x does NOT equal y.

`x = 'str'`
`y = 'stR'`
`x != y`
	`True`	Remember that strings are case sensitive. `str` is not the same as `stR`. So when we compare x and y, they are not equal, so `x != y` evaluates to true.

`x = 'str'`
`y = 'str'`
`x != y`
	`False`	Both values are the same, so `x != y` evaluates to false. Remember, we are checking whether x does NOT equal y.

Go online to complete the Not Equal To Challenge

< (less than)

This compares whether the first object is less than the second object. x < y means that x is less than y. When you use `x < y`, you are asking the question "*Is the value of x less than the value of y?*".

Comparisons can be chained arbitrarily: `3 < 5 < 7` gives `True` because 3 is less than 5 which is also less than 7.

You can also compare strings using less than, but it is complicated. The string characters are converted into ASCII decimal values and those numbers are compared. In most cases, you will not want to use less than to compare strings.

Code	Output	Notes
`5 < 3`	`False`	5 is not less than 3, so this evaluates to false.
`3 < 5`	`True`	3 is less than 5, so this evaluates to true.
`3 < 3`	`False`	3 is not less than itself, so this returns as false.

`'a' < 'b'`	`True`	In this case, a has an ASCII value of 97 and b has an ASCII value of 98. 97 is less than 98 so this evaluates as true.

Go online to complete the Less Than Challenge

> (greater than)

This compares if the first object is greater or larger than the second object. `x > y` means that x is greater than y. When you use `x > y`, you are asking the question "*Is the value of x more than the value of y?*".

Code	Output	Notes
`5 > 3`	`True`	5 is greater than 3, so this evaluates to true.
`3 > 5`	`False`	3 is not greater than 5, so this evaluates to false.
`3 > 3`	`False`	3 is not greater than itself, so this returns as false.
`'a' > 'b'`	`False`	In this case, a has an ASCII value of 97 and b has an ASCII value of 98. 97 is less than 98 so this evaluates as false.

Go online to complete the Greater Than Challenge

<= (less than or equal to)

This compares whether the first object is less than or equal to the second object. `x <= y` means that x is less than or equal to y. The "or equal to" part is important because as we saw before, `3 < 3` evaluates as false, but `3 <= 3` would evaluate as true because or the "or equal to" part. When you use `x <= y`, you are asking the question "*Is the value of x less than or the same as the value of y?*"

Remember that the equals sign comes last. There is no such thing as =<.

Code	Output	Notes

`5 <= 3`	`False`	5 is not less than 3 and it is not equal to 3, so this evaluates to false.
`3 <= 5`	`True`	3 is less than 5, so this evaluates to true.
`3 <= 3`	`True`	3 is not less than itself, but it is equal to itself so this returns as true.
`'a' <= 'b'`	`True`	In this case, a has an ASCII value of 97 and b has an ASCII value of 98. 97 is less than 98 so this evaluates as true.

`x = 3`
`y = 6`
`x <= y`
	`True`	3 is less than 6, so this evaluates as true.

Go online to complete the
Less Than or Equal To Challenge

>= (greater than or equal to)

This compares whether the first object is greater than or equal to the second object. x >= y means that x is greater than or equal to y. The "or equal to" part is important because as we saw before, 3 > 3 evaluates as false, but 3 >= 3 would evaluate as true because or the "or equal to" part. When you use x <= y, you are asking the question "*Is the value of x more than or the same as the value of y?*"

Remember that the equals sign comes last. There is no such thing as =>.

Code	Output	Notes
`5 >= 3`	`True`	5 is greater than 3, so this evaluates to true.
`3 >= 5`	`False`	3 is not greater than 5, so this evaluates to false.
`3 >= 3`	`True`	3 is not greater than itself, but it is equal to itself so this returns as false.
`'a' >= 'b'`	`False`	In this case, a has an ASCII value of 97 and b has an ASCII value of 98. 97 is less than 98 so this evaluates as false.

`x = 4`
`y = 3`
`x >= 3`
	`True`	4 is greater than 3, so it evaluates as true.

	Go online to complete the Greater Than or Equal To Challenge

Go online to complete the
Comparison Operators Challenge

if: Conditional Statements

So now we know how to use comparison operators to compare different values in order to reach a `True` or `False` conclusion. Let's put this to use:

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the `if` statement:

```
if [condition]:
    [do something]
```

The Boolean expression after `if` is called the condition. If it is true, the indented statement runs. If not, nothing happens.

Notice how the `if` statement contains a colon at the end - we are indicating to Python that a block of statements follows.

Notice also that the *`[do something]`* part is indented. The first line of the condition statement is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, indentation is always four spaces.

Code	Output	Notes

`x = 1`
`if x > 0:`
`    print('x is positive')`
	`x is positive`	The condition `x > 0` evaluates to true because 1 is greater than 0. So the condition is satisfied and the body of statements executes.

`x = -5`
`if x > 0:`
`    print('x is positive')`
	None	The condition `x > 0` evaluates to false because -5 is less than 0. So the condition is not satisfied nothing happens.

`x = 0`
`if x > 0:`
`    print('x is positive')`
	None	The condition `x > 0` evaluates to false because 0 is not greater than 0 (it is equal and we aren't looking for that). So the condition is not satisfied nothing happens.

`my_variable = True`
`if my_variable:`
`    print("It's true!")`
	`It's true!`	The variable `my_variable` is a Boolean value. It evaluates to true because its value is set to True. So the condition is satisfied and the body of statements executes.

`color = "red"`
`if color == "blue":`
`    print("It's blue!")`
	None	The variable `color` is a string assigned to the value "red". We compare it to the string "blue" to check if they are equal. They are not, so the condition evaluates as false. The condition is not satisfied so nothing happens.

Go online to complete the Conditional Statement Challenge

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing – it just "passes" by and continues the code.

```
if x < 0:
    pass          # TODO: need to handle negative values!
```

else: Alternative Execution

A second form of the `if` statement is "alternative execution", in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

```
if [condition]:
    [do something]
else:
    [do something else]
```

If the first condition does not evaluate to true, then the code in the body of the `else` header will execute. It is the alternative action. Just like with the `if` statement, the `else` statement ends with a colon and the code block within its body is indented exactly the same ways as the `if` code block.

Look at this conditional statement:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called branches, because they are branches in the flow of execution.

Code	Output	Notes

`x = 7`
 	
`if x % 2 == 0:`
`    print('x is even')`
`else:`
`    print('x is odd')`
	`x is odd`	In this case, `7 % 2` equals 1, because when you divide 7 by 2, you get 3 with a remainder of 1. Since `x % 2 == 1`, then the first condition is not satisfied and the alternative code is executed.

`x = 6`
 	
`if x % 2 == 0:`
`    print('x is even')`
`else:`
`    print('x is odd')`
	`x is even`	In this case, `6 % 2` equals 0, because when you divide 6 by 2, you get 3 with a remainder of 0. Since `x % 2 == 0`, then the first condition is satisfied and the alternative code is not executed.

Go online to complete the
Alternate Execution Challenge

elif: Chained Conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional. It follows this format:

```
if [condition #1]:
    [do something]
elif [condition #2]:
    [do something else]
else:
    [do something even more different]
```

`elif` is an abbreviation of "else if". Again, exactly one branch will run. There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

Code	Output	Notes

`x = 1`
`y = 2`
 	
`if x < y:`
	`x is less than y`	1 is less than 2, so the very first condition of `x < y` evaluates to true, so we see

`    print('x is less than y')`
`elif x > y:`
`    print('x is greater than y')`
`else:`
`    print('x and y are equal')`
		`x is less than y`.
		
		Note: Python ignores the second condition and the alternative.

`x = 6`
`y = 5`
 	
`if x < y:`
`    print('x is less than y')`
`elif x > y:`
`    print('x is greater than y')`
`else:`
`    print('x and y are equal')`
	`x is greater than y`	6 is greater than 5, so the very first condition of `x < y` evaluates to false. Python moves to the next condition, `x > y`, which evaluates as true, so we see `x is greater than y`.
		
		Note: Python ignores the alternative.

`x = 0`
`y = 0`
 	
`if x < y:`
`    print('x is less than y')`
`elif x > y:`
`    print('x is greater than y')`
`else:`
`    print('x and y are equal')`
	`x and y are equal`	0 is equal to 0, so the very first condition of `x < y` evaluates to false. Python moves to the next condition, `x > y`, which evaluates as false. Python moves on to the alternative, so we see `x and y are equal`.

Go online to complete the
Chained Conditional Challenge

Go online to complete the
Fix this Code Challenge

Nested Conditionals

One conditional can also be nested within another. We could have written the example in the previous section like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. It is a good idea to avoid them when you can.

Logical Operators

Sometimes we need to compare multiple things to make a determination about which path to follow.

For example, let's say I want to see a movie. I have a child with me, so I can't see a PG-13 or R-rated movie. I don't have enough money for a 3D film, so that selection is out. And it is the weekend, so all the first run movies are sold out; I am limited to movies which have been in the theater for a while. This is a complicated scenario that looks at several things at once.

I can use **logical operators** to string several things together at once.

There are three logical operators in Python: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English.

and

Just as in English, and in Python looks at two scenarios. In other for the entire statement to evaluate as true, both scenarios have to true. `x and y` returns `False` if x is `False`, otherwise it returns evaluation of y.

Code	Output	Notes

`x = False`
`y = True`
`x and y`
	`False`	The only way this evaluates as true is when BOTH x and y are `True`. Since x is `False`, this entire statement evaluates to false.
		
		In this case, Python will not evaluate y since it knows that the left hand side of the 'and' expression is `False` which implies that the whole expression will be `False` irrespective of the other values. This is called short-circuit evaluation.

`x = 3`
`x > 0 and x < 10`
	`True`	We know `x = 3`. 3 is greater than 0 and 3 is less than 10, so this evaluates to true.

`x = 15`
`x > 0 and x < 10`
	`False`	While it is true that 15 is greater than 0, 15 is not less than 10. Since `x < 10` is false, then the entire statement evaluates as false.

How would this look in a conditional statement?

Code	Output

`x = False`
`y = True`
 	
`if x and y:`
`    print("true")`
`else:`
`    print("false")`
	`false`

`x = 3`
`if x > 0 and x < 10:`
`    print("true")`
`else:`
`    print("false")`
	`true`

`x = 15`
`if x > 0 and x < 10:`
`    print("true")`
`else:`
`    print("false")`
	`false`

Go online to complete the
Logical Operator - and Challenge

or

Once again Python looks at two scenarios but this one acts a little differently. If *one or both* of the scenarios is true, then the entire statement will evaluate to true. If x is `True`, it returns True, else it returns evaluation of y.

Code	Output	Notes

`x = False`
`y = True`
`x or y`
	`True`	This evaluates as true is when EITHER x and y are `True`. Since x is `False`, then Python looks at y. Since y is True, then this entire statement evaluates to true.
		
		Short-circuit evaluation applies here as well. If x had evaluated as true, then Python would not have bothered to evalute y. and the entire statement would evaluate as true.

`x = 3`
`x > 0 or x < 10`
	`True`	We know x = 3. 3 is greater than 0, so Python would stop right there and evaluate this to true.

`x = -10`
`x > 0 or x < 10`
	`True`	Negative 10 is not greater than 0, so Python looks at the second scenario. -10 is less than 10, so the entire statement evaluates as true.

`x = 3`
`x <= 2 or x > 5`
	`False`	3 is not less than or equal to 2, so the first scenario evaluates as false. 3 is also not greater than 5. Since both scenarios evaluate as false, the entire statement is false.

How would this look in a conditional statement?

Code	Output

`x = False`
`y = True`
`if x or y:`
`    print("true")`
`else:`
`    print("false")`
	`true`

`x = 3`
`if x > 0 or x < 10:`
`    print("true")`
`else:`
`    print("false")`
	`true`

`x = -10`
`if x > 0 or x < 10:`
`    print("true")`
`else:`
`    print("false")`
	`true`

`x = 3`
`if x <= 2 or x > 5:`
`    print("true")`
`else:`
`    print("false")`
	`false`

Go online to complete the
Logical Operator - or Challenge

not

The `not` operator negates a boolean expression – or does the opposite, so not (x > y) is True if x > y is False, that is, if x is less than or equal to y.

Code	Output	Notes

`x = True`
`not x`
	`False`	If `x = True`, then the opposite of `True` is `False`. So `not x` is the same as saying `not True`, which is `False`. So `not x` evaluates to `False`.

`x = 4`
`y = 5`
`not x > y`
	`True`	Look from the inside, out. Inside, we have `x > y`. 4 is not greater than 5, so `x > y` evaluates as false. But then we have `not (False)`. The opposite of `False` is `True`, so the entire statement evaluates as `True`.

How would this look in a conditional statement?

Code	Output

`x = True`
`if not x:`
`    print("true")`
`else:`
`    print("false")`
	`false`

`x = 4`
`y = 5`
`if not x > y:`
`    print("true")`
`else:`
`    print("false")`
	`true`

Go online to complete the
Logical Operator - not Challenge

Simplifying Nested Conditionals

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

The print statement runs only if we make it past both conditionals, so we can get the same effect with the and operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

For this kind of condition, Python provides a more concise option:

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

Go online to complete the
Nested Conditional Statement Challenge

Using Logical Operators with non-Boolean Objects

Strictly speaking, the operands of the logical operators should be Boolean expressions, but Python is not very strict. Any nonzero number is interpreted as True:

```
>>> 42 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

Using Parenthesis in Conditional Statements

When you use multiple logical operators, it can be helpful – more readable and less prone to error – to group them together using parenthesis:

Example
`if (x and y):`
`if (x > 0) and (x < 10):`
`if (color1=="blue") and (color2=="green" or color2=="purple"):`
`if not (color1=="blue" or color1=="red"):`

Go online to complete the
Fix this Code Challenge

Example: Number Guessing

In this program, we take guesses from the user and check if it is the number that we have.

```
number = 23
guess = int(input('Enter an integer: '))

if guess == number:
    # New block starts here
    print('You guessed it!')
    # New block ends here

elif guess < number:
    # Another block
    print('No, you are too low!')

else:
    # you must have guessed > number to reach here
    print('No, you are too high!')


print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

Output:

First Execution:

```
Enter an integer : 50
No, it is a little lower than that
Done
```

Second Execution:

```
Enter an integer : 22
No, it is a little higher than that
Done
```

Third Execution:

```
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

How It Works

Code	Output	Explanation
`number = 23`	*None*	First, we assign the literal constant value 23 to the variable number using the assignment operator (=).

`guess = int(input('Enter an`
`integer: '))`
	`Enter an integer:`	Next, we use the `input` statement to ask the user to enter a number. We also use the `int` statement to convert the input to an integer. We assign this integer to the variable `guess` and continues.
		
		Note: if the user enters non-numerals, such as the word "Hi" then the program will throw a Runtime error when trying to convert it to an integer.

`if guess == number:`
`    # New block starts here`
`    print('You guessed it!')`
`    # New block ends here`
	`You guessed it!`	Next, we compare the guess of the user with the number we have chosen.
		
		If they are equal, we print a success message. Notice that we use indentation levels to tell Python which statements belong to which block. This is why indentation is so important in Python. I hope you are sticking to the "consistent indentation" rule. Are you?
		
		Notice how the `if` statement contains a colon at the end - we are indicating to Python that a block of statements follows.

`elif guess < number:`
`    # Another block`
`    print('No, you are too low!')`
	`No, you are too low!`	Then, we check if the guess is less than the number, and if so, we inform the user that they must guess a little higher than that. What we have used here is the `elif` clause which actually combines two related `if else-if else` statements into one combined `if-elif-else` statement. This makes the program easier and reduces the amount of indentation required.

`else:`
`    # you must have guessed >`
`    # number to reach here`
`    print('No, you are too high!')`
	`No, you are too high!`	Finally, if `guess` is not equal to `number` and `guess` is not less than `number`, then we use the `else` clause to tell them they picked a number which was too large.
		
		We could also have written this as

		`elif guess > number:` but using `else` accomplished the same thing.

`print('Done')`
`# This last statement is always executed,`
`# after the if statement is executed.`
	`Done`	After Python has finished executing the complete `if` statement along with the associated `elif` and `else` clauses, it moves on to the next statement in the block containing the `if` statement. In this case, it is the main block (where execution of the program starts), and the next statement is the `print('Done')` statement. After this, Python sees the ends of the program and simply finishes up.

Even though this is a very simple program, I have been pointing out a lot of things that you should notice. All these are pretty straightforward (and surprisingly simple for those of you from C/C++ backgrounds). You will need to become aware of all these things initially, but after some practice you will become comfortable with them, and it will all feel 'natural' to you.

Controlling Output

Let's switch topics for a moment. Thus far, we have been merely displaying data using the `print` statement and its default characteristics = namely, printing a space between components and ending the statement with a line break. We can control this behavior.

Item Separator

We have seen that you can print multiple literal constants and variables using the `print` statement:

Code	Output
`print("A", "B", "C")`	`A B C`

`color = "blue"`
`print('The sky is', color)`
	`The sky is blue`

Python automatically separates each component using a space. By what if we don't want to use a space? What if we want to use a hyphen or a comma or nothing at all? Python has you covered.

The components which make up the print statement are called *arguments*. In the above examples, the string `"I see"`, the variable `number`, the variable `animal`, and the string `"running"` are all arguments.

The print statement has a special argument which controls item separator:

Code	Output	Notes
`print("A", "B", "C", sep="")`	ABC	We set the `sep` argument equal to an empty string. When the components were displayed, they ran together.

`color = "blue"`
`print('The sky is', color, sep='')`
	`The sky isblue`	We set the `sep` argument equal to an empty string. When the components were displayed, they ran together.

You can use any string value for the sep argument:

Code	Output	Notes
`print("A", "B", "C", sep="-")`	`A-B-C`	We set the `sep` argument equal to a hyphen.
`print("A", "B", "C", sep="!!")`	`A!!B!!C`	We set the `sep` argument equal to two exclamation points.

`print("A", "B", "C", sep="\n")`	`A`
	`B`
	`C`
		We set the `sep` argument equal to a line break.

When you do not specify the `sep` argument, Python defaults to a space. When you want to use it, just separate it from the other components with a comma, then set your separator to whatever string you like.

End Behavior

We have also seen that the `print` statement automatically ends each statement with a line break.

Code	Output

`print("Line 1")`
`print("Line 2")`
	`Line 1`
	`Line 2`

`print("Line 1\nLine 2")`
`print("Line 3")`
	`Line 1`
	`Line 2`
	`Line 3`

The print statement also has an argument to control the end behavior:

Code	Output	Notes

`print("Line 1", end=" ")`
`print("Line 2")`
	`Line 1 Line 2`	We set the end argument equal to a space of the first statement to a space and let the second one default. This ran the first and second statements together and separated them with a space. The second statement still ends with a line break.

`print("Line 1", end="")`
`print("Line 2")`
	`Line 1Line 2`	We set the end argument equal to an empty string. Now they run together. The second statement still ends in a line break.

You can use any string for the end argument as well:

Code	Output	Notes

`print("Line 1", end="\n\n")`
`print("Line 2")`
	`Line 1`
	
	`Line 2`
		Two line breaks

`print("Line 1", end="\n\t~")`
`print("Line 2")`
	`Line 1`
	`        ~Line 2`
		Line break, tab, and a tilde

When you do not specify the end argument, Python defaults to a line break. When you want to use it, just separate it from the other components with a comma, then set your end argument to whatever string you like.

Using Them Together

You can use the `sep` and end arguments individually as needed or together. The order does not matter, so long as they follow your list of components. Here are some examples:

Code	Output

`print("Line 1", "Line 2", sep="!", end="")`
`print("Line 3")`
	`Line 1!Line 2Line3`

`print("Line 1", "Line 2", end=" ", sep="\n")`
`print("Line 3")`
	`Line 1`
	`Line 2 Line 3`

Go online to complete the
Ending Newline and Item Separator Challenge

Formatting Numbers

We saw in the last lesson that numbers do not always display "prettily" when we want them to. For example, if we were to calculate the 4.75% tax on a $2.99 purchase we get:

Code	Output

`tax_rate = 0.0475`
`cost = 2.99`
`tax = 0.0475*2.99`
`print("You pay", tax, "in tax")`
	`You pay 0.142025 in tax`

Wait a minute! We can't pay $0.142025 in tax – we have no coin to represent fractions of a penny! It would make sense to round down to 0.14 and just display that. Fortunately, Python makes it easy for us to format numbers using the `format` function.

Just like the `print` statement, the `format` function accepts arguments. At the very least, it requires two: the number being formatted, and the *format specifier*. The format specifier is a special string which instructs Python on how to present the value. It is actually a whole language unto itself, but we'll display some of the more common examples.

Formatting Floats

Code	Output

`tax_rate = 0.0475`
`cost = 2.99`
`tax = 0.0475*2.99`
`print("You pay", format(tax, ".2f"),`
`"in tax")`
	`You pay 0.14 in tax`

The code we used - `format(tax, ".2f")` – accepted the variable tax as the first argument. The second argument is the format specifier. The `.2` represents the precision we want; in this case, 2 decimal places. The `f` represents the type of formatting we are performing. There is a whole library on the different types, but in this case f is for a fixed point number.

Note that `format()` will round up or down depending on the value of the next decimal place. If the number is greater than or equal to 5, it will round up. If the decimal is less than 5, it will round down.

Let's look at an estimation of the number pi and format it.

Code	Result	
`pi = 3.14159265359`	*none*	We assign the approximate value of pi to the variable `pi`.
`format(pi, ".0f")`	`3`	`.0` means no decimal places. Notice that this rounded down. The next number after 3 was a 1. 1 is less than 5, so it rounded down.
`format(pi, ".1f")`	`3.1`	`.1` means one decimal place. Notice that this rounded down. The next number after 3.1 was a 4. 4 is less than 5, so it rounded down.
`format(pi, ".2f")`	`3.14`	`.2` means two decimal places. Notice that this rounded down. The next number after 3.14 was a 1. 1 is less than 5, so it rounded down.
`format(pi, ".3f")`	`3.142`	`.3` means three decimal places. Notice that this rounded up. The next number after 3.141 was a 5. 5 is greater than or equal to 5, so it rounded up to 3.142.
`format(pi, ".4f")`	`3.1416`	`.4` means four decimal places. Notice that this rounded up. The next number after 3.1415 was a 9. 9 is greater than or equal to 5, so it rounded up to 3.1416

When you are dealing with large numbers, you might want to use a comma to separate the thousandths place like this: 14,726,986.19. We just need to add a comma to the format specifier:

Code	Result
`big_number = 14726986.192271`	*none*

`format(big_number, ",.0f")`	`14,726,986`
`format(big_number, ",.1f")`	`14,726,986.2`
`format(big_number, ",.2f")`	`14,726,986.19`

Go online to complete the Formatting Floats Challenge

Perhaps we want to format a percentage? In this case, we drop the `f` from the format specifier and replace it with a percentage sign `%`. Then we set our precision level to whatever we like:

Code	Result
`test_score = 0.754931`	*none*
`format(test_score, ".0%")`	`75%`
`format(test_score, ".1%")`	`75.5%`
`format(test_score, ".2%")`	`75.49%`

You can even combine the comma formatting with the percentage:

Code	Result
`sales_increase = 74290.754931`	*none*
`format(sales_increase, ",.2%")`	`7,429,075.49%`

Go online to complete the Formatting Percentages Challenge

Formatting Integers

Integers don't have decimal values as they are whole numbers. You can still use `,.0f` to format a large integer, but it would be more proper to use `d` (for "decimal integer in base 10") instead of `f`. And since integers do not have decimal values, we can drop the precision. Look how these two different format specifiers produce identical results.

Code	Result
`cars = 10473`	*none*
`format(cars, ",.0f")`	`10,473`
`format(cars, ",d")`	`10,473`

Go online to complete the
Formatting Integers Challenge

Debugging

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10 \log_{10} (P_{signal} / P_{noise})$. In Python, you might write something like this:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

When you run this program, you get an exception:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, which uses floor division `//` instead of floating-point division `/`.

You should take the time to read error messages carefully, but don't assume that everything they say is correct.