Lesson 3: Iteration

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called iteration.

Because iteration is so common, Python provides language features to make it easier.

The while Statement

The `while` statement allows you to repeatedly execute a block of statements so long as a condition is true. It is considered to be a condition-controlled statement.

The simplest form of the `while` statement is this:

```
while [condition]:
    [do something]
```

Just like with the `if` statement, the Boolean expression after `while` is called the **condition**. If the condition is true, the body of the `while` statement executes. Once the body is completed, the condition is re-evaluated. If it is found to still be true, then the body executes again. This repeats until the condition is found to be false. At that point, the `while` statement stops and the rest of the program continues to run.

More formally, here is the flow of execution for a `while` statement:

1. Determine whether the condition is true or false.
2. If false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top.

Pretest Loop

The `while` statement is considered to be a pretest loop – this means it tests the condition before it decided to execute. So if you need to run the body of the `while` statement at least once, you may need to prime the `while` statement by setting the condition to be true ahead of time.

Example: Bad Joke

Here is a program which tells a knock knock joke using a `while` statement:

```
knock_knock = True

while knock_knock:
    print("\nKnock, knock!")
    print("  Who's there?")
    print("Banana.")
    print("  Banana who?\n")

    again = input("Continue the joke? y/n ")

    if again!="y":
        knock_knock = False
```

```
        print("\nKnock, knock!")
        print("  Who's there?")
        print("Orange.")
        print("  Orange who?")
        print("Orange you glad I didn't say 'banana'?")
```

Output

```
Knock, knock!
  Who's there?
Banana.
  Banana who?

Continue the joke? y/n y

Knock, knock!
  Who's there?
Banana.
  Banana who?

Continue the joke? y/n n

Knock, knock!
  Who's there?
Orange.
  Orange who?
Orange you glad I didn't say 'banana'?
```

How it Works

We've set a condition control `knock_knock` equal to True. The while statement checks that `knock_knock` is true, then performs the actions in the body of the statement. In this case, the user is prompted to enter a value, the value is evaluated using a conditional statement.

Code	Output	Explanation
`knock_knock = True`	*None*	First, we assign the Boolean value `True` to the variable `knock_knock`.
`while knock_knock:`	*None*	We begin the `while` statement with the condition of `knock_knock`. This is a shortcut way of writing `knock_knock == True`. We first check this condition. Since `knock_knock` IS True, the condition evaluates as true. We move on to execute the body.

`print("\nKnock, knock!")`
`print("  Who's there?")`
`print("Banana.")`
	`Knock, knock!`
	`  Who's there?`
	`Banana.`
		We display the beginning of a bad knock knock joke.

`print("  Banana who?\n")`	`  Banana who?`	
`again = input("Continue the joke? y/n ")`	`Continue the joke? y/n`	We accept input from the user.
`if again!="y":` `    knock_knock = False`	*None*	We evaluate the input. If the user enters something other than 'y' we set the value of `knock_knock` to `False`. If the user enters 'y' then nothing happens and the `while` loop continues. In our example, we entered 'y' so the `while` loop repeats.

`print("\nKnock, knock!")`
`print("  Who's there?")`
`print("Orange.")`
`print("  Orange who?")`
`print("Orange you glad I didn't say 'banana'?")`
	`Knock, knock!`
	`  Who's there?`
	`Orange.`
	`  Orange who?`
	`Orange you glad I didn't say 'banana'?`
		After we set the value of `knock_knock` to `False`, the program continues. In this case, we display the punchline.

Go online to complete the
Overview of the while loop Challenge

Example: Countdown

Here is a program which counts down to 0 using a `while` statement:

```
n = 5

while n > 0:
    print(n)
    n = n - 1

print('Blastoff!')
```

Output

```
5
4
3
2
1
Blastoff!
```

How it Works

You can almost read the `while` statement as if it were English. It means, "While n is greater than 0, display the value of n and then decrement n. When you get to 0, display the word *Blastoff!*"

Code	Output	Explanation
`n = 5`	*None*	First, we assign the literal constant value `5` to the variable `n`.
`while n > 0:`	*None*	We begin the `while` statement with the condition of `n > 0`. We first check this condition. Since `n = 5` and `5 > 0`, the condition evaluates as true. We move on to execute the body.
`    print(n)` `    n = n - 1`	`5`	We display the value of n, then reassign n to be one less.
`while n > 0:`	*None*	We re-evaluate the condition again. `n = 4` and `4 > 0`, so the condition again evaluates as true.
`    print(n)` `    n = n - 1`	`4`	We display the value of n, then reassign n to be one less.
`while n > 0:`	*None*	We re-evaluate the condition again. `n = 3` and `3 > 0`, so the condition again evaluates as true.
`    print(n)` `    n = n - 1`	`3`	We display the value of n, then reassign n to be one less.
`while n > 0:`	*None*	We re-evaluate the condition again. `n = 2` and `2 > 0`, so the condition again evaluates as true.
`    print(n)` `    n = n - 1`	`2`	We display the value of n, then reassign n to be one less.
`while n > 0:`	*None*	We re-evaluate the condition again. `n = 1` and `1 > 0`, so the condition again evaluates as true.
`    print(n)` `    n = n - 1`	`1`	We display the value of n, then reassign n to be one less.
`while n > 0:`	*None*	We re-evaluate the condition again. `n = 0`, but `0` is not greater than 0, so the condition evaluates as false. The `while` statement is finished and the program moves on.
`print('Blastoff!')`	`Blastoff!`	We display the word *Blastoff!* The program is finished.

Go online to complete the
Using while to Countdown Challenge

Infinite Loops

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat", are an infinite loop.

In the case of the countdown, we can prove that the loop terminates: if n is zero or negative, the loop never runs. Otherwise, n gets smaller by one each time through the loop, so eventually we have to get to 0.

For some other loops, it is not so easy to tell. For example:

```
while n != 1:
      print(n)
      if n % 2 == 0:        # n is even
            n = n / 2
      else:                 # n is odd
            n = n*3 + 1
```

The condition for this loop is `n != 1`, so the loop will continue until n is 1, which makes the condition false.

Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, n is divided by 2. If it is odd, the value of n is replaced with `n*3 + 1`. For example, if the argument passed to sequence is 3, the resulting values of n are 3, 10, 5, 16, 8, 4, 2, 1.

Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, n will be even every time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for all positive values of n. So far, no one has been able to prove it or disprove it! (See http://en.wikipedia.org/wiki/Collatz_conjecture .)

Go online to complete the
Preventing Infinite Loops Challenge

else: Alternative Execution

A `while` statement can have an optional `else` clause. This code will execute once the condition is no longer true. We can re-write the Countdown program using the `else` clause like this:

```
n = 5

while n > 0:
    print(n)
    n = n - 1
```

```
else:
    print('Blastoff!')
```

We will get the exact same output.

Example: Number Guessing, Part 2

In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly run the program for each guess, as we have done in the previous section. This aptly demonstrates the use of the `while` statement.

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer: '))

    if guess == number:
        print('You guessed it!\n')
        running = False

    elif guess < number:
        print('No, you are too low!\n')

    else:
        print('No, you are too high!\n')

else:
    print('The while loop is over.')

print('Done')
```

Output:

```
Enter an integer: 50
No, you are too high!

Enter an integer: 22
No, you are too low!

Enter an integer: 23
You guessed it!

The while loop is over.
Done
```

How It Works

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

		`True` to the variable `running`.
`while running:` **`    guess = int(input('Enter an integer: '))`** **`    if guess == number:`** **`        print('You guessed it!')`** **`        running = False`** **`    elif guess < number:`** **`        print('No, you are too low!')`** **`    else:`** **`        print('No, you are too high!')`**	*Varies depending on the user's input*	We move the `input` and `if` statements to inside the `while` loop and set the variable `running` to `True` before the while loop. First, we check if the variable `running` is `True` and then proceed to execute the corresponding *while-block*. After this block is executed, the condition is again checked which in this case is the `running` variable. If it is true, we execute the while-block again, otherwise we continue to execute the optional else-block and then continue to the next statement. Notice that when the correct number is guessed, we change the value of `running` to `False`. This will stop the while loop and continue in our program.
`else:` **`    print('The while loop is over.')`**	`The while loop is over.`	The `else` block is executed when the `while` loop condition becomes `False` - this may even be the first time that the condition is checked. If there is an `else` clause for a `while` loop, it is always executed unless you break out of the loop with a `break` statement.
`print('Done')`	`Done`	The `while` statement is finished. The program continues and the word `Done` is displayed.

Go online to complete the
Using else in a while Statement Challenge

The for Statement

Sometimes, we may prefer to loop a specific number of times, rather than checking whether a condition is true or false. Fortunately, we have the `for` statement which does exactly that.

The `for` statement is another looping statement which *iterates* over a sequence of objects i.e. go through each item in a sequence. We will see more about sequences in detail in later lessons. What you need to know right now is that a sequence is just an ordered collection of items.

The `for` statement is considered to be a count-controlled statement.

The simplest form of the `for` statement is this:

```
for [variable] in [item1, item2, item3, etc]:
    [do something]
```

Example: Countdown, Part 2

Here is how we can re-write the countdown program using a `for` statement:

```
for n in [5, 4, 3, 2, 1]:
    print(n)

print('Blastoff!')
```

Output

```
5
4
3
2
1
Blastoff!
```

How it Works

Code	Output	Explanation
`for n in [5, 4, 3, 2, 1]:`	*None*	We begin the `for` statement by selecting a variable – and we can call it anything. n is a traditional variable to use to represent a number, but we could really call it almost anything. Then we create our list. We separate each item using a comma and we enclose the entire list in bracket symbols [ ]. We then move on to execute the body.
`print(n)`	5	The first time we go through the loop, the variable n represents the first element in the list. We display the value of n. And that's it for the body of the loop.
`for n in [5, 4, 3, 2, 1]:`		The second time we go through the

`    print(n)`	`4`	loop, the variable n represents the second element in the list. We display the value of n.
`for n in [5, 4, 3, 2, 1]:` `    print(n)`	`3`	The third time we go through the loop, the variable n represents the third element in the list. We display the value of n.
`for n in [5, 4, 3, 2, 1]:` `    print(n)`	`2`	The fourth time we go through the loop, the variable n represents the fourth element in the list. We display the value of n.
`for n in [5, 4, 3, 2, 1]:` `    print(n)`	`1`	The fifth time we go through the loop, the variable n represents the fifth element in the list. We display the value of n. We're out of elements in the list now, so the `for` statement is finished and the program moves on.
`print('Blastoff!')`	`Blastoff!`	We display the word *Blastoff!* The program is finished.

A list doesn't have to contain numbers. It can contain strings and other objects as well:

Code	Output

`for day in ['Mon', 'Tues', 'Weds']:`
`    print(day)`
	`Mon`
	`Tues`
	`Weds`

`for stuff in [0, 'hi', 2.75, 'bye']:`
`    print(stuff)`
	`0`
	`hi`
	`2.75`
	`bye`

We'll learn more about lists in a later lesson.

Go online to complete the
for Loop with a Numeric List Challenge

Go online to complete the
for Loop with a List of Strings Challenge

else: Alternative Execution

A `for` statement can also have an optional `else` clause. This code will execute once the list is exhausted. We can re-write the Countdown program using the `else` clause like this:

```
for n in [5, 4, 3, 2, 1]:
    print(n)
else:
    print('Blastoff!')
```

Again, we will get the exact same output.

Go online to complete the
Rewrite Fibonacci Challenge

Loop Patterns

Often we use a `for` or `while` loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

Counting and Summing Loop Examples

For example, to count the number of items in a list, we would write the following for loop:

```
count = 0

for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1

print 'Count: ', count
```

We set the variable `count` to zero before the loop starts, then we write a `for` loop to run through the list of numbers. Our iteration variable is named `itervar` and while we do not use `itervar` in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add 1 to the current value of `count` for each of the values in the list. While the loop is executing, the value of `count` is the number of values we have seen "so far".

Once the loop completes, the value of `count` is the total number of items. The total number "falls in our lap" at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Go online to complete the Counting Using a for Loop Challenge

Another similar loop that computes the total of a set of numbers is as follows:

```
total = 0

for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar

print 'Total: ', total
```

In this loop we do use the iteration variable. Instead of simply adding one to the `count` as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running `total` during each loop iteration. If you think about the variable `total`, it contains the "running total of the values so far". So before the loop starts `total` is zero because we have not yet seen any values, during the loop total is the running total, and at the end of the loop total is the overall total of all the values in the list.

As the loop executes, `total` accumulates the sum of the elements; a variable used this way is sometimes called an accumulator.

Go online to complete the Fibonacci Sum Challenge

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions `len()` and `sum()` that compute the number of items in a list and the total of the items in the list respectively.

Code	Output
`total_count = len([3, 41, 12, 9, 74, 15])` `print(total_count)`	6
`total_sum = sum([3, 41, 12, 9, 74, 15])` `print(total_sum)`	154

Maximum and Minimum Loop Examples

To find the largest value in a list or sequence, we construct the following loop:

```
largest = None

print 'Before:', largest
```

```
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print 'Loop:', itervar, largest

print 'Largest:', largest
```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable `largest` is best thought of as the "largest value we have seen so far". Before the loop, we set `largest` to the constant `None`. `None` is a special constant value which we can store in a variable to mark the variable as "empty".

Before the loop starts, the largest value we have seen so far is `None` since we have not yet seen any values. While the loop is executing, if `largest` is `None` then we take the first value we see as the largest so far. You can see in the first iteration when the value of `itervar` is 3, since `largest` is `None`, we immediately set `largest` to be 3.

After the first iteration, `largest` is no longer `None`, so the second part of the compound logical expression that checks `itervar > largest` triggers only when we see a value that is larger than the "largest so far". When we see a new "even larger" value we take that new value for `largest`. You can see in the program output that `largest` progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable `largest` now does contain the largest value in the list.

Go online to complete the Maximum Value Challenge

To compute the smallest number, the code is very similar with one small change:

```
smallest = None

print 'Before:', smallest

for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print 'Loop:', itervar, smallest

print 'Smallest:', smallest
```

Again, smallest is the "smallest so far" before, during, and after the loop executes. When the loop has completed, smallest contains the minimum value in the list.

Go online to complete the
Minimum Value Challenge

Again as in counting and summing, the built-in functions `max()` and `min()` make writing these exact loops unnecessary.

Code	Output
`maximum = max([3, 41, 12, 9, 74, 15])` `print(maximum)`	`74`
`minimum = min([3, 41, 12, 9, 74, 15])` `print(minimum)`	`3`

Input Validation Loop Example

Program end users are incompetent. They rarely read directions and they will come up with an infinite number of ways to break your program. We'll learn more about ways of trapping many types of errors, but for now, we can use an input validation loop to steer the end users in the right direction.

Example: Candy Bar Ordering System

This programs accepts the number of candy bars a user wishes to order.

```
candy_bars = int(input("Enter number of candy bars desired: "))

while candy_bars < 0:
    print("You can\'t order negative candy bars!")
    candy_bars = int(input("Enter number of candy bars desired: "))

if candy_bars == 0:
    print("I'm sorry you don't like candy bars")

elif candy_bars > 20:
    print("That's a big order!")

elif candy_bars > 50:
    print("You sure like candy!")

print("Thank you for ordering", candy_bars, "candy bars!\n")
```

Output:

```
Enter number of candy bars desired: -1
You can't order negative candy bars!
Enter number of candy bars desired: 2
Thank you for ordering 2 candy bars!
```

```
Enter number of candy bars desired: 22
That's a big order!
Thank you for ordering 22 candy bars!

Enter number of candy bars desired: 122
You sure like candy!
Thank you for ordering 122 candy bars!
```

How It Works

We'll focus on the input validation loop.

Code	Output
`candy_bars = int(input("Enter number of candy bars desired: "))`	`Enter number of candy bars desired:`

We assign the variable `candy_bars` to accept the input from the end user.

Code	Output

`while candy_bars < 0:`
`  print("You can\'t order negative candy bars!")`
`  candy_bars = int(input("Enter number of candy bars desired: "))`
	Varies depending on the user's input

This is our input validation loop. Our condition is that `candy_bars` is less than 0 (or a negative number). So if our end user turns out to be a wise guy and plugs in `-1`, this while loop will trigger, yell at the end user, and prompt them for another entry. If the end user continues to enter negative numbers, the loop will continue until the condition (negative number) is no longer satisfied. Then the rest of the code continues.

If the users does the right thing and enters a number greater than or equal to 0 from the beginning, then the `while` loop is skipped altogether.

Pretty neat, huh?

Go online to complete the
Jelly Bean Input Validation Challenge

The range function

Let's say you wanted to count to 100. It would be a pain to have list each number out. The `range` function can make it much easier to create and manage a `for` loop.

The range function is a built-in function in Python. We will learn more about functions in the next lesson. For now, functions are simply reusable bits of code that can be called when needed.

The range function helps us to create a sequence of integers. The creation is based on information, called an **argument**, which you pass into the range function.

Here's an example:

Code	Sequence	Notes
`range(3)`	`0, 1, 2`	Start at zero and go up to, but not including, 3.
`range(5)`	`0, 1, 2, 3, 4`	Start at zero and go up to, but not including, 5.

Here we have passed the integer 3 into the `range` function. When we call `range(3)`, we get the numbers `0, 1, 2`.

Note that the first item in the sequence is a zero. And notice that the sequence goes up to, but does not include, 3.

We can use `range(3)` in a `for` loop like this:

Code	Output

`for n in range(3):`
`    print(n)`
	`0`
	`1`
	`2`

Go online to complete the
for Loop using a Range Challenge

Go online to complete the
Calculations in a for Loop Challenge

Of course, we don't want every sequence to begin with a zero. The `range` function has other options as well, and you pass these into the function too. Adding a second integer will allow you to control where you start and where you stop.

Code	Sequence	Notes
`range(1, 3)`	`1, 2`	Start at 1 and go up to, but not including, 3.

`range(2, 5)`	`2, 3, 4`	Start at 2 and go up to, but not including, 5.

When we use `range(2, 5)` in a `for` statement, we get:

Code	Output
`for n in range(2, 5):` `  print(n)`	2 3 4

Notice that the first integer passed into the `range` function controls the starting integer in the sequence. The second integer passed into the `range` function controls the ending integer - but again, it is up to an not including that integer.

Go online to complete the
Fix the for Loop using a Range Challenge

Finally, we might want to be able to skip count. Remember as a kid? Counting to 100 by 10s? That was skip counting. Python calls this the step. The third argument passed into the `range` function controls the step in which the sequence is incremented or decremented (because you can also go backwards!). Here are some examples:

Code	Sequence	Notes
`range(1, 10, 2)`	`1, 3, 5, 7, 9`	Start at 1 and go up to, but not including, 10. Go up in steps of 2.
`range(5, 35, 4)`	`5, 9, 13, 17, 21, 25, 29, 33`	Start at 5 and go up to, but not including, 35. Go up in steps of 4.

When we use `range(1, 10, 2)` in a `for` statement, we get:

Code	Output
`for n in range(1, 10, 2):` `  print(n)`	1 3 5 7 9

Go online to complete the
Skip Counting Challenge

Remember how you can go backwards? Just start at a higher number, stop at a lower number, and use a negative step.

Code	Sequence	Notes
`range(10, 0, -2)`	`10, 8, 6, 4, 2`	Start at 10 and go **down** to, but not including, 0. Go down in steps of 2.

Example: Countdown, Part 3

Here is how we can re-write the countdown program using the `range` function:

```
for n in range(5, 0, -1):
    print(n)

print('Blastoff!')
```

Output

```
5
4
3
2
1
Blastoff!
```

How it Works

Code	Output	Explanation
`for n in range(5, 0, -1):` **`    print(n)`**	5 4 3 2 1	Range creates a sequence. It starts at 5. The step argument (the last one) tells us to down by 1 until, but not including, the stop argument of 0. In other words: `[5, 4, 3, 2, 1]` The `for` statement then iterated through the generated sequence like normal.
`print('Blastoff!')`	`Blastoff!`	We display the word *Blastoff!* The program is finished.

Go online to complete the Countdown by 10s Challenge

The arguments which are passed into the range function can be variables as well - just so long as they are integers. This allows more control over your program.

Code	Sequence	Notes

`start = 5`
`stop = 10`
`step = 2`
 	
`range(start, stop, step)`
	`[5, 7, 9]`	Start at 5 and go up to, but not including, 10. Go up in steps of 2.

`x = 5`
`y = 35`
`z = 4`
 	
`range(x, y, z)`
	`[5, 9, 13, 17, 21, 25, 29, 33]`	Start at 5 and go up to, but not including, 35. Go up in steps of 4.

This means that you have your user control the countdown program as well.

The break Statement

The `break` statement is used to *break* out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become `False` or the sequence of items has not been completely iterated over.

For example, suppose you want to take input from the user until they type done. You could write:

```
while True:
    line = input('Are you done? ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

The loop condition is `True`, which is always true, so the loop runs until it hits the `break` statement.

Each time through, it prompts the user with the question *Are you done?* If the user types *done*, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
Are you done? not done
not done

Are you done? done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

An important note is that if you *break* out of a `for` or `while` loop, any corresponding loop `else` block is **not** executed.

Note that you can use `break` with the `for` statement as well.

Go online to complete the
Using break Challenge

The continue Statement

The `continue` statement is used to tell Python to skip the rest of the statements in the current loop block and to *continue* to the next iteration of the loop.

For example:

```
for n in [1, 2, 3]:
    if n==2:
        continue
    print(n)
```

When we execute this code, we get:

```
1
3
```

So even though there is clearly a 2 in our list of 1, 2, 3, the 2 is skipped. Why? In the body block of the `for` statement, there is a conditional statement. The conditional is looking at whether `n = 2`. When `n = 2`, the conditional uses the `continue` statement to skip the rest of the body block of the `for` statement and go straight to the next iteration. So, the 2 is never printed. It is skipped.

The better practice might be to leave the 2 out of your list, but it is nice to know that you can use `continue` if you need it.

Note that you can use `continue` with the `while` statement as well.

Go online to complete the
Using continue Challenge

Nested Loops

Just as with conditional statements, you can also nest loops. You can nest `for` loops in `while` loops and `while` loops in `for` loops. You can nest `for` loops in other `for` loops and `while` loops in other `while` loops. What follows are some examples and ideas for creating programs.

Example: Triangles

This code produces some pretty triangles:

```
for x in range(3):
    for y in range(3, 0, -1):
        print("|", " "*x, "*"*y)
```

Output

```
|  ***
|  **
|  *
|   ***
|   **
|   *
|    ***
|    **
|    *
|     ***
|     **
|     *
```

How It Works

Code	Output	Explanation
`for x in range(4):`	*None*	This is the outer `for` loop. It will be iterated 3 times. The variable x will take on the values of 0, 1, 2, and 3.
`    for y in range(3, 0, -1):`	*None*	This is the inner `for` loop. It will also be iterated 3 times. The variable y will take on the values of 3, 2, and 1. Because the inner `for` loop is inside of the outer `for` loop, the inner `for` loop will completely execute a total of 3 times **each** time the outer `for` loop runs.
`        print("|", " "*x, "*"*y)`		This is the statement which changes during each iteration. Let's do these one at a time. When the outer `for` loop runs, `x = 0`. Immediately, the inner `for` loop runs where `y = 4`. Let's look at the below table to see each individual iteration.

Here are our iterations. Since the outer loop will run 4 times and the inner loop will run 3 times, there will be 12 total iterations:

Iteration	x value	y value	print statement	Output
1	0	3	`print("|", " "*0, "*"*3)`	| ***
2	0	2	`print("|", " "*0, "*"*2)`	| **
3	0	1	`print("|", " "*0, "*"*1)`	| *
4	1	3	`print("|", " "*1, "*"*3)`	| ***
5	1	2	`print("|", " "*1, "*"*2)`	| **
6	1	1	`print("|", " "*1, "*"*1)`	| *
7	2	3	`print("|", " "*2, "*"*3)`	| ***
8	2	2	`print("|", " "*2, "*"*2)`	| **
9	2	1	`print("|", " "*2, "*"*1)`	| *
10	3	3	`print("|", " "*3, "*"*3)`	| ***
11	3	2	`print("|", " "*3, "*"*2)`	| **
12	3	1	`print("|", " "*3, "*"*1)`	| *

Go online to complete the
Nested for Loops Challenge

Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is “debugging by bisection”. For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.