**LOOPS**

**What are Loops?**

In our everyday lives, we tend to repeat a lot of processes without noticing.

For instance, if we want to cook a delicious recipe, we might have to prepare our ingredients by chopping them up. We chop and chop and chop until all of our ingredients are the right size. At this point, we stop chopping.

If we break down our chopping task into a series of three smaller steps, we have:

1. An initialization: We’re ready to cook and have a collection of ingredients we want to chop. We will start at the first ingredient.
2. A repetition: We’re chopping away. We are performing the action of chopping over and over on each of our ingredients, one ingredient at a time.
3. An end condition: We see that we have run out of ingredients to chop and so we stop.

In programming, this process of using an initialization, repetitions, and an ending condition is called a [*loop*](https://www.codecademy.com/resources/docs/python/loops?page_ref=catalog). In a loop, we perform a process of *iteration* (repeating tasks).

Programming languages like Python implement two types of iteration:

1. *Indefinite iteration*, where the number of times the loop is executed depends on how many times a condition is met.
2. *Definite iteration*, where the number of times the loop will be executed is defined in advance (usually based on the collection size).

Typically we will find loops being used to iterate a collection of items. In the above example, we can think of our ingredients we want to chop as our collection. This is a form of definite iteration since we know how long our collection is in advance and thus know how many times we need to iterate over the collection of ingredients.

Some collections might be small — like a short string, while other collections might be massive like a range of numbers from 1 to 10,000,000! But don’t worry, loops give us the ability to masterfully handle both ends of the spectrum. This simple, but powerful, concept saves us a lot of time and makes it easier for us to work with large amounts of data.

In this lesson, we’ll learn how to use Python to implement both definite and indefinite iteration in our own programs.

**Instructions**

Look over (and over) the provided diagram. Then, go to the next exercise to get looped in!



**Why Loops?**

Before we get to writing our own loops, let’s explore what programming would be like if we couldn’t use loops.

Let’s say we have a list of ingredients and we want to print every element in the list:

ingredients = ["milk", "sugar", "vanilla extract", "dough", "chocolate"]

If we only use print(), our program might look like this:

print(ingredients[0])  
print(ingredients[1])  
print(ingredients[2])  
print(ingredients[3])  
print(ingredients[4])

The output would be:

milk  
sugar  
vanilla extract  
dough  
chocolate

That’s still manageable, We’re writing 5 print() statements (or copying and pasting a few times). Now imagine if we come back to this program and our list had 10, or 24601, or … 100,000,000 elements? It would take an extremely long time and by the end, we could still end up with inconsistencies and mistakes.

Don’t dwell too long on this tedious scenario — we’ll learn how loops can help us out in the next exercise. For now, let’s gain an appreciation for loops.

**Instructions**

**1.**

Using 10 print() statements, print out: "This can be so much easier with loops!".

Checkpoint 2 Passed

Hint

If you already have an idea of how to use loops (or have searched up how to do it) we’re still expecting 10 individual print() statements.

If for some reason you still are not passing the tests, make sure all your print() statements have the same text "This can be so much easier with loops!".

Copy and paste is your best friend here 😉

**script.py**

# Write 10 print() statements below!

print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

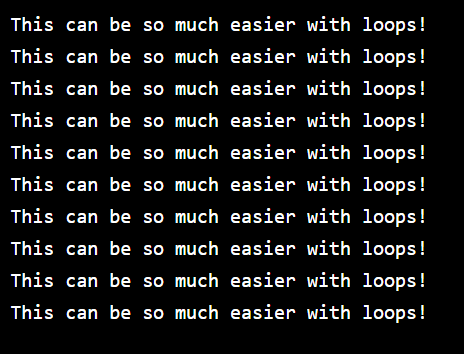
print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

print("This can be so much easier with loops!")

****

**For Loops: Introduction**

Now that we can appreciate what loops do for us, let’s start with your first type of loop, a for loop, a type of definite iteration.

In a for loop, we will know in advance how many times the loop will need to iterate because we will be working on a collection with a predefined length. In our examples, we will be using Python lists as our collection of elements.

With for loops, on each iteration, we will be able to perform an action on each element of the collection.

Before we work with any collection, let’s examine the general structure of a for loop:

for <temporary variable> in <collection>:  
  <action>

Let’s break down each of these components:

1. A for keyword indicates the start of a for loop.
2. A <temporary variable> that is used to represent the value of the element in the collection the loop is currently on.
3. An in keyword separates the temporary variable from the collection used for iteration.
4. A <collection> to loop over. In our examples, we will be using a list.
5. An <action> to do anything on each iteration of the loop.

Let’s link these concepts back to our ingredients example. This for loop prints each ingredient in ingredients:

ingredients = ["milk", "sugar", "vanilla extract", "dough", "chocolate"]  
   
for ingredient in ingredients:  
  print(ingredient)

In this example:

1. ingredient is the <temporary variable>.
2. ingredients is our <collection>.
3. print(ingredient) was the <action> performed on every iteration using the temporary variable of ingredient.

This code outputs:

milk  
sugar  
vanilla extract  
dough  
chocolate

Some things to note about for loops:

* **Temporary Variables:**

A temporary variable’s name is arbitrary and does not need to be defined beforehand. Both of the following code snippets do the exact same thing as our above example:

for i in ingredients:  
  print(i)

for item in ingredients:  
 print(item)

Programming best practices suggest we make our temporary variables as descriptive as possible. Since each iteration (step) of our loop is accessing an ingredient it makes more sense to call our temporary variable ingredient rather than i or item.

* **Indentation:**

Notice that in all of these examples the print statement is indented. Everything at the same level of indentation after the for loop declaration is included in the loop body and is run on every iteration of the loop.

for ingredient in ingredients:  
  # Any code at this level of indentation   
  # will run on each iteration of the loop  
  print(ingredient)

If we ever forget to indent, we’ll get an IndentationError or unexpected behavior.

* **Elegant loops:**

Python loves to help us write elegant code so it allows us to write simple for loops in one-line. In order to see the below example as one line, you may need to expand your narrative window. Here is the previous example in a single line:

for ingredient in ingredients: print(ingredient)

**Note**: One-line for loops are useful for simple programs. It is not recommended you write one-line loops for any loop that has to perform multiple complex actions on each iteration. Doing so will hurt the readability of your code and may ultimately lead to buggier code.

Let’s practice writing our own for loop!

**Instructions**

**1.**

Run the code.

We should get an IndentationError because the print(game) line is not indented.

Checkpoint 2 Passed

Hint

You should see the following error indicating that we forgot to indent our code:

File "script.py", line 6

print(game)

^

IndentationError: expected an indented block

**2.**

Indent (2 spaces or tab) line 6 so that we don’t get an IndentationError when you run the code.

Run the code again!

Checkpoint 3 Passed

Hint

Your code should look like this:

for game in board\_games:  
  print(game)

**3.**

Write a for loop that prints each sport in the list sport\_games.

Checkpoint 4 Passed

Hint

The general structure of a for loop is:

for <temporary variable> in <collection>:  
  <action>

In our case here are each of our components:

1. Our <temporary variable> is anything we want. Since we are accessing individual sports, it might make sense to call it sport (the singular of the word sports).
2. Our <collection> is the list sport\_games.
3. Take a stab at figuring out the action. Think about what we are trying to do on each iteration (step) of the loop.

**script.py**

board\_games = ["Settlers of Catan", "Carcassone", "Power Grid", "Agricola", "Scrabble"]

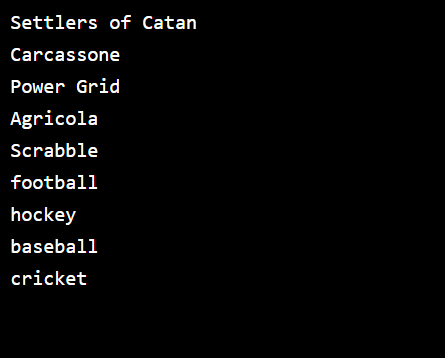
sport\_games = ["football", "hockey", "baseball", "cricket"]

for game in board\_games:

  print(game)

for sport in sport\_games:

  print(sport)

****