Introduction

In a previous video we learned how to work with simple lists in Python. In this and later lessons we'll look at some more advanced ways to work with lists in Python.

Looping through a list

Looping in Python allows us to perform the same action on every item in a list. Imagine that you have a list containing several million phone numbers and you wanted to remove the plus(+) symbol that appears at the start of international numbers. Looping through the list could accomplish this task easily and in a short amount of time.

The time to use a loop in Python is when you need to perform the same action on every item in a list. If you choose to go on and learn Data Science with Python, loops play a big role in cleaning data before putting it to use. Let's take a look at some examples:

In [5]:
```
# Define a list
books_available = ['book1', 'book2', 'book3', 'book4', 'book5', 'book6', 'bo

# Define a for loop
for book in books_available:
    print(book)
```

```
book1
book2
book3
book4
book5
book6
book7
book8
book9
book10
```

In the code above we have a list containing 10 books for sale on our website. If we had to go through our list and use the **print** statement to display each book to screen this would be a long process. What happens if we have 20 or 100 books? What happens when a book is no longer available for sale?

- At line 1 we have defined a list
- Line 5 is where we define a loop. This line of code tells Python to take an element from our list of books and assign it to the variable **book**.
- Line 6 should be familiar to you by now, here we tell Python to print out the element that has just been assigned to the variable **book**.

We have ten elements in our list, and Python will repeat lines two and three until it there are no more elements left to print.

Think of it this way, for every book in the list, books_available, print the book.

As you write more Python programs you will discover that looping is one of the most common ways for a program to carry out repetitive tasks. In the example above, the loop begins at line 5. This line tells Python to get the first value from the list, **books available** and associate it with the variable **book**. At line 6, we tell Python to print out the current value of **book**, which in this example is **book1**. Python will continue this process until all of the items from the list have been printed out. When there are no more items left to print, Python will move to the next line of the program. In our example there is no next line so the program ends.

When you create **for** loops in Python keep in mind that you can choose any name for the variable. We used **book**, 'for book in books_available:' but you can use any name you like. It's best practice to try to choose a name that is meaningful as it will not always be you reading your code.

Doing more with for loops

Not only can we print out each item from a list using a **for** loop but we can do almost anything to that item. Let's take a look:

In [6]:
```
# Create a list of books
books_available = ["the everything store", "elon musk", "steve jobs"]

# Define a for loop
for book in books_available:
    print(f"{book.title()}, is in stock.")
```

```
The Everything Store, is in stock.
Elon Musk, is in stock.
Steve Jobs, is in stock.
```

In this example we have looped through each item of the list **books_available**, used the title method on each item and then combined that item with the string "..., is in stock."

We could keep adding lines of code to our loop, for example:

In [7]:
```
# Create a list of books
books_available = ["the everything store", "elon musk", "steve jobs"]

# Define a for loop
for book in books_available:
    print(f"{book.title()}, is in stock.")
    print("Would you like to order this book? Y/N\n")
```

```
The Everything Store, is in stock.
Would you like to order this book? Y/N

Elon Musk, is in stock.
Would you like to order this book? Y/N

Steve Jobs, is in stock.
Would you like to order this book? Y/N
```

Why has the second print statement printed out three times? It's because we have made it part of the **for** loop by indenting it. Every indented line of code after line 5, the creation

of the **for** loop, will be executed once for every item in the **books_available** list. You can add as many lines of code that are necessary in your **for** loops.

After your **for** loop has executed any lines of code that are note indented will run once without being repeated.

In [8]:
```
# Create a list of books
books_available = ["the everything store", "elon musk", "steve jobs"]

# Define a for loop
for book in books_available:
    print(f"{book.title()}, is in stock.")
    print("Would you like to order this book? Y/N\n")

# Print statement
print("Note: There is a 5 day lead time for all orders.")
```

```
The Everything Store, is in stock.
Would you like to order this book? Y/N

Elon Musk, is in stock.
Would you like to order this book? Y/N

Steve Jobs, is in stock.
Would you like to order this book? Y/N

Note: There is a 5 day lead time for all orders.
```

In this example, the two print statements inside the **for** loop execute for each item in the list. The final **print** statement at line 10 is executed only once as it is not indented inside the **for** loop.

Indentation errors

In Python, indentation and white space are used to work out how a line, or several lines of code, are related to the rest of the program. In the examples above we've seen how the indented print statements were executed as part of the **for** loop. It is this use of indentation that helps to make Python easy to read. As you begin to write longer programs you will write lines of code which are indented to form blocks of code at several different levels.

For new programmers indentation is a common area for errors. You may forget to indent, you may indent where it is not necessary or you may incorrectly indent within a block of code.

Let's look some examples of indentation errors and hopefully this will help you avoid them in the future.

Forgetting to indent

In the example below we return to our **for** loop.

In [9]:
```
# Create a list of books
books_available = ["the everything store", "elon musk", "steve jobs"]
```

```
# Define a for loop
for book in books_available:
print(f"{book.title()}, is in stock.")
```

```
  Input In [9]
    print(f"{book.title()}, is in stock.")
    ^
IndentationError: expected an indented block
```

As you can see at line 6 I have removed the indentation. When the code is run Python returns an error, telling us that it expected an indented block at line 6. This error is easy to resolve, just add indentation to line 6. The rule is always indent the line after the **for** statement in a loop.

Forgetting to indent additional lines

In [10]:
```
# Create a list of books
books_available = ["the everything store", "elon musk", "steve jobs"]

# Define a for loop
for book in books_available:
    print(f"{book.title()}, is in stock.")
print("Would you like to order this book? Y/N\n")

# Print statement
print("Note: There is a 5 day lead time for all orders.")
```

```
The Everything Store, is in stock.
Elon Musk, is in stock.
Steve Jobs, is in stock.
Would you like to order this book? Y/N

Note: There is a 5 day lead time for all orders.
```

In the example above I have neglected to indent the print statement at line 7. The program has run and not generated any errors but the output is not what I expected.

Every time you define a **for** loop, Python expects to find at least one indented line after the **for** statement, if it does, no error is reported. At line 6 we have an indented **print** statement so this statement is executed once for each item on the list. The next **print** statement at line 7 is not indented and so is executed only once after the loop has finished running.

In Python, this is called a **logical error**. Our code is valid and runs successfully but the output is not what we wanted. We expected to see the line "Would you like to order this book? Y/N" after every book title, instead we see it only once at the end of our program. To solve this error we review our code and determine if and where an indentation might be missing and we indent one or several lines of code.

Unnecessarily indenting

If we accidentally indent a line of code Python is going to let us know.

In [11]:
```
out_of_stock_message("This title is currently unavailable.")
    print(out_of_stock_message)
```

```
  Input In [11]
    print(out_of_stock_message)
    ^
IndentationError: unexpected indent
```

In this example, we do not need to indent the **print** statement at line 2.

Unnecessarily indenting after the loop

In [14]:
```
# Create a list of books
books_available = ["the everything store", "elon musk", "steve jobs"]

# Define a for loop
for book in books_available:
    print(f"{book.title()}, is in stock.")
    print("Would you like to order this book? Y/N\n")

# Print statement
    print("Note: There is a 5 day lead time for all orders.")
```

```
The Everything Store, is in stock.
Would you like to order this book? Y/N

Note: There is a 5 day lead time for all orders.
Elon Musk, is in stock.
Would you like to order this book? Y/N

Note: There is a 5 day lead time for all orders.
Steve Jobs, is in stock.
Would you like to order this book? Y/N

Note: There is a 5 day lead time for all orders.
```

In the example above we have mistakenly indented the **print** statement at line 10. As a result, this code is repeated for each item on the list. This is another logical error. Python may report this error but most times your code will run with unexpected results.

Forgetting the colon

When you create a **for** loop you add a colon at the end of the **for** statement. This colon tells Python that the next line of code is the start of the loop. If you forget to add the colon you will get what's called a syntax error because Python is unsure of what is suppose to happen. Let's take a look:

In [15]:
```
# Create a list of books
books_available = ["the everything store", "elon musk", "steve jobs"]

# Define a for loop
for book in books_available
    print(f"{book.title()}, is in stock.")
```

```
  Input In [15]
    for book in books_available
                               ^
SyntaxError: invalid syntax
```

The solution to this error is to simply add the colon at the end of the **for** statement.

Exercises

Here are three products, lamp, chair, and candle. Store these products in a list, and then use a **for** loop to print the name of each product.

- Update your **for** loop to add a message to each product. For example "...available in other colors." or "I would like to order a PRODUCT NAME"
- Add a message to the end of your program that will execute only once.

Creating numerical lists

As you expand your Python programs you won't be just creating lists that contain strings, you'll also be creating numerical lists. For example, you may need to track daily, weekly or monthly sales figures, if you are creating a program related to the weather you might want to record temperatures. There are countless examples.

In Python lists can be used to store sets of numbers. If you go on to work in the areas of data analysts or data science you will be working with lists that contain millions of items.

Range() function

Let's kick off our exploration of numerical lists in Python with the **range** function.

In [16]:
```
for value in range(1, 10):
    print(value)
```

```
1
2
3
4
5
6
7
8
9
```

These two lines of code contain a **for** loop and a **print** statement to print out the range of numbers 1 - 9. Why not 1 - 10? The **range()** function instructs Python to start counting at the first value provided to it, which in this case is 1. The **range()** function stops when it reaches the second value provided, which in this example is 10. Because it stops at the second value, the output never contains the end value.

To print the numbers 1 - 10 we would use range(1, 11):

In [17]:
```
for value in range(1, 11):
    print(value)
```

```
1
2
3
4
5
6
7
8
9
10
```

When using the **range()** function and you find that your output is not what you expected try adjusting your end value by 1.

You can also pass **range()** one argument and it will start counting at 0. For example:

In [18]:
```
for value in range(11):
    print(value)
```

```
0
1
2
3
4
5
6
7
8
9
10
```

Making a list of numbers

We've just been using Python's **range()** function to generate a range of numbers, but these are not lists. To make a list we must convert the results of **range()** into a list with the **list()** function. Let's take a look:

In [19]:
```
months = list(range(1, 13))
print(months)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

In this example we place the **range()** function inside the **list()** function, which outputs for us a list of numbers. How do you know that the output is a list? It's surrounded by square brackets and separated by commas.

The **range()** function can also be used to tell Python to skip numbers in a given range. If you pass a third argument to **range()**, Python will use that value as a step size when generating numbers. For example:

In [20]:
```
odd_numbers = list(range(1, 11, 2))
print(odd_numbers)
```

```
[1, 3, 5, 7, 9]
```

Here the **range()** function starts with the value 1 and then adds 2 to that value. It will continue to add 2 until it reaches or passes the end value .

The **range()** function can be used to create almost any set of numbers that you can think of. For example:

In [21]:
```
squares = []
for value in range(1, 21):
    square = value ** 2
    squares.append(square)
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 32
4, 361, 400]
```

In this example we've created a list of square numbers in the range 1 to 20. In Python two asterisks(`**`) represents exponents.

- At line one we create an empty list called squares
- At line two we tell Python to loop through each value from 1 to 21 using the **range()** function
- Inside the loop the current value is raised to the second power and then assigned to the variable **square**
- At line four each new value of **square** is appended to the list **squares**
- At line 6, when the loop has completed we tell Python to print out the list of squares

This code could also be written as:

In [22]:
```
squares = []
for value in range(1, 21):
    squares.append(value**2)
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 32
4, 361, 400]
```

Simple statistics

Let's look at some other Python functions that we can use to work with lists of numbers.

In [23]:
```
# Create a list of numbers
numbers = [1 ,2, 3, 4, 5, 6, 7, 8, 9, 10]
print(min(numbers))
```

```
1
```

In [24]:
```
# Create a list of numbers
numbers = [1 ,2, 3, 4, 5, 6, 7, 8, 9, 10]
print(max(numbers))
```

```
10
```

In [25]:
```
# Create a list of numbers
numbers = [1 ,2, 3, 4, 5, 6, 7, 8, 9, 10]
print(sum(numbers))
```

```
55
```

List comprehension

In a previous example we created a list of squares with four lines of code. With list comprehension we can achieve the same task in only one line of code.

List comprehension combines a for loop and the creation of new elements into one line of code, and automatically appends each new element. Let's write the code to create a list of squares as we did in a previous example but this time using list comprehension.

In [26]:
```
squares = [value**2 for value in range(1, 21)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 32
4, 361, 400]
```

We start by creating a variable **squares**. We then open a set of square brackets and create the expression for the values we want in the new list, in this example **value**2**. We then write a **for** loop on the same line and close the square brackets.

List comprehension takes practice.

Exercises

- Use a **for** loop to print the numbers one to fifty, inclusive
- create a list of numbers from one to 50,000
- Using the list of 50,000 use the functions, **max()** and **sum()**
- Create a list of even numbers by adding a third argument to the **range()** function
- Create a list of cubed numbers (hint: the cube of 2 in Python is `2**3`). Use a **for** loop to display the results

How to work with part of a list

We've learned how to access single elements from a list, and we've just seen how to work through all the elements of a list. Now let's learn how to work with individual elements or groups within a list. This is what Python calls a **slice**.

Slicing a list

To create a slice, you tell Python the index of the first and last elements you want to work with. Just like the **range()** function, Python will not return the last item or index you specify. Let's take a look:

In [27]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
print(employees[0:3])
```

```
['tony', 'robert', 'mary']
```

As you can see to output the first three elements of the list **employees** we requested indices 0 to 3, which return the elements at positions 0, 1, 2.

In the code example above we created a new list and then printed out a slice of that list. This slice has the first three employees. Also, note that the output is itself a list.

We can create any slice of a list that fits our purposes. For example:

In [28]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
print(employees[1:4])
```

```
['robert', 'mary', 'john']
```

Here we have returned the second, third and fourth items from our list.

If we leave out the first index when creating a slice, Python will automatically start at the beginning of the list:

In [29]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
print(employees[:3])
```

```
['tony', 'robert', 'mary']
```

Similarly if we want to create a slice to the end of a list we can do write the following:

In [30]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
print(employees[1:])
```

```
['robert', 'mary', 'john', 'anne']
```

This code is very helpful when you want to output from any point of a list to the end of a list but do not know the lists length.

You might remember that in a previous video we seen how a negative index returns an element from the end of a list? Using this syntax we can create a slice from the end of a list. Again, this is very helpful when you do not know the length of a list. Let's take a look:

In [31]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
print(employees[-1:])
```

```
['anne']
```

In [32]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
print(employees[-3:])
```

```
['mary', 'john', 'anne']
```

Looping through a slice

We've see **for** loops and we can use them here again to loop through a slice. Let's take a look:

In [33]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
for employee in employees[:3]:
    print(employee.title())
```

```
Tony
Robert
Mary
```

Here, we have looped through a slice or subset of our employees list. Here is another example:

In [34]:
```
employees = ['tony', 'robert', 'mary', 'john', 'anne']
```

```
print("The employees working today are:\n")
for employee in employees[0:4]:
    print(employee.title())
```

```
The employees working today are:

Tony
Robert
Mary
John
```

Slices are a very powerful feature of Python and also very helpful. For example, recently I was working on a dataset that contained over 250,000 customers. Rather than work on the entire set I created a slice of a small subset of customers. This way I could write and test code using only a small number of customers rather than the full 250,000.

Copying a list

Imagine you are at work, and your boss emails you a list of customers that she needs some data manipulation performed on. You don't want to work on the original list as this is the source of truth, which you may need to refer to in the future, like when you mess up! The best thing to do here is to create a copy of the list.

To create a copy of a list we can make a slice of the entire list. We do this with the slice syntax, [:]. This tells Python to make a slice that starts at the first item and ends with the last item, the full list. Let's take a look:

In [35]:
```
customers = ['phil', 'claire', 'alex', 'luke', 'haley', 'jay', 'gloria', 'jo
customer_orders = customers[:]

print("Here are our customer names:")
print(customers)

print("\nCustomers who have made recent orders are:")
print(customer_orders)
```

```
Here are our customer names:
['phil', 'claire', 'alex', 'luke', 'haley', 'jay', 'gloria', 'joe']

Customers who have made recent orders are:
['phil', 'claire', 'alex', 'luke', 'haley', 'jay', 'gloria', 'joe']
```

Let's add some new customers to our list:

In [36]:
```
customers = ['phil', 'claire', 'alex', 'luke', 'haley', 'jay', 'gloria', 'jo
new_customers = customers[:]

customers.append('cam')
new_customers.append('mitchell')

print("Here is our customer list:")
print(customers)

print("\nHere is a list of new customers")
print(new_customers)
```

```
Here is our customer list:
['phil', 'claire', 'alex', 'luke', 'haley', 'jay', 'gloria', 'joe', 'cam']

Here is a list of new customers
['phil', 'claire', 'alex', 'luke', 'haley', 'jay', 'gloria', 'joe', 'mitchel
l']
```

In the example above we have our original list of customers which we copy to create a new list called **new_customers**. We then add a customer to the copy using Python's **append()** method. We print out both lists to display the updates.

Why not simply write, new_customers = customers?

This line of code would tell Python to associate the variable **new_customers** with the list that is in **customers**, meaning that both variables would point to the same list. This means that any time we update a list with a new customer the other list would update with that customer as well.

Exercises

Create a new list using a code from an example above. With your new list try the following:

- Print the message "The first three elements of this list are:". Use a slice to print the first three elements from the list.
- Print the message "The middle two elements of this list are:". Use a slice to print the middle two elements from the list.
- Print the message "The last element of this list is:". Use a slice to print the last element on the list.
- Create a new list of your family or friends and create a copy that list
- Add a new person to the copy of the list

Tuples

First off, what is a **tuple**? A tuple is a list of items that cannot change. Lists are great for storing items that need to change, customers, sales, employees, web site users. There will be times when you need to create lists that cannot change and thats what tuples are for. In Python a value that cannot change is immutable, and an immutable list is called a tuple.

A tuple is defined with opening and closing parentheses (). You can access individual elements of a tuple using its index just as we did with lists.

Let's take a look at an example. The floor size of your bedroom, living room, kitchen or garage are not going to change unless you do some construction work.

In [37]:
```
# Defining a tuple
kitchen_size_meters = (5, 12)
print(kitchen_size_meters[0])
print(kitchen_size_meters[1])
```

```
5
12
```

In this code example we define a tuple at line 2 using parentheses. At lines 3 and 4 we print out each size of our kitchen using the same method we've been using to access items in a list.

What happens if we try to change an element in **kitchen_size_meters**.

In [38]:
```
# Defining a tuple
kitchen_size_metres = (5, 12)
kitchen_size_metres[0] = 6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [38], in <cell line: 3>()
      1 # Defining a tuple
      2 kitchen_size_metres = (5, 12)
----> 3 kitchen_size_metres[0] = 6

TypeError: 'tuple' object does not support item assignment
```

In this code example we have tried to change the first element of the tuple **kitchen_size_meters** from 5 to 6. Python has returned an error telling us that we can't assign a new value to an item in a tuple. This is exactly what should happen.

> Note: Tuples are defined by the presence of a comma. The parentheses help to make them look neater and more readable. If you ever need to create a tuple with just one element you will need to include a trailing comma, size = (100,)

Looping through a tuple

Similar to lists we can loop over all of the items in a tuple using a **for** loop.

In [39]:
```
# Defining a tuple
kitchen_size_meters = (5, 12)
for size in kitchen_size_meters:
    print(size, "meters")
```

```
5 meters
12 meters
```

Writing over a tuple

As mentioned you cannot change a tuple once it has been defined. You can assign a new value to a variable that represents a tuple. Let's take a look at how we could change the size of our kitchen:

In [40]:
```
kitchen_size_meters = (5, 12)
print("The current kitchen size is: ")
for size in kitchen_size_meters:
    print(size, "meters")

kitchen_size_meters = (10, 14)
print("\nThe new kitchen size will be: ")
```

```
for size in kitchen_size_meters:
    print(size, "meters")
```

```
The current kitchen size is:
5 meters
12 meters

The new kitchen size will be:
10 meters
14 meters
```

In this code example we have done two things:

1. Defined the original tuple and print out the size of our kitchen
2. At line 6 we assign a new tuple to the variable __kitchen_size_meters
3. We then print out the new tuple without any errors

Exercises

Think of items that could be use to create a tuple, for example, the size of rooms in your home or workplace, or coordinates such as longitude and latitude.

- Create a tuple from your items
- Use a **for_loop** to print each item
- Try and modify your tuple and examine the output from Python
- Rewrite your tuple to add new items and then print your new tuple out