Python 3 Fundamentals

Diego López Tamayo [*]

Contents

[*]El Colegio de México, diego.lopez@colmex.mx

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." — Martin Fowler.

Phyton Sintax

Introduction

Python is a programming language. Like other languages, it gives us a way to communicate ideas. In the case of a programming language, these ideas are “commands” that people use to communicate with a computer!

We convey our commands to the computer by writing them in a text file using a programming language. These files are called programs. Running a program means telling a computer to read the text file, translate it to the set of operations that it understands, and perform those actions.

```
my_name = "Diego"
print("Hello and welcome " + my_name + "!")
```

```
## Hello and welcome Diego!
```

Comments

Ironically, the first thing we’re going to do is show how to tell a computer to ignore a part of a program. Text written in a program but not run by the computer is called a comment. Python interprets anything after a # as a comment.

Comments can:

- Provide context for why something is written the way it is:

```
# This variable will be used to count the number of times anyone tweets the word hello
hello_count = 0
```

- Help other people reading the code understand it faster:

```
# This code will calculate the likelihood that it will rain tomorrow
## complicated_rain_calculation_for_tomorrow()
```

- Ignore a line of code and see how a program will run without it:

```
# useful_value = old_sloppy_code()
##  useful_value = new_clean_code()
```

Print

Now what we’re going to do is teach our computer to communicate. The gift of speech is valuable: a computer can answer many questions we have about “how” or “why” or “what” it is doing. In Python, the **print()** function is used to tell a computer to talk. The message to be printed should be surrounded by quotes:

```
# from Karl Marx's "Das Kapital"
print("Just as man is governed, in religion, by the products of his own brain, so, in capitalist produc
```

```
## Just as man is governed, in religion, by the products of his own brain, so, in capitalist production
```

In the above example, we direct our program to print() an excerpt from a notable book. The printed words that appear as a result of the print() function are referred to as output. The output of this example program would be:

Strings

Computer programmers refer to blocks of text as strings. In Python a string is either surrounded by double quotes (“Hello world”) or single quotes (‘Hello world’). It doesn’t matter which kind you use, just be consistent.

Variables

Programming languages offer a method of storing data for reuse. If there is a greeting we want to present, a date we need to reuse, or a user ID we need to remember we can create a variable which can store a value. In Python, we assign variables by using the equals sign (=).

```
message_string = "Hello there"
# Prints "Hello there"
print(message_string)
```

```
## Hello there
```

In the above example, we store the message "Hello there" in a variable called message_string. Variables can't have spaces or symbols in their names other than an underscore (_). **They can't begin with numbers** but they can have numbers after the first letter (e.g., cool_variable_5 is OK).

It's no coincidence we call these creatures "variables". If the context of a program changes, we can update a variable but perform the same logical process on it.

```
# Greeting
message_string = "Hello there"
print(message_string)

# Farewell
```

```
## Hello there
```

```
message_string = "Hasta la vista"
print(message_string)
```

```
## Hasta la vista
```

Above, we create the variable message_string, assign a welcome message, and print the greeting. After we greet the user, we want to wish them goodbye. We then update message_string to a departure message and print that out.

Errors

Humans are prone to making mistakes. Humans are also typically in charge of creating computer programs. To compensate, programming languages attempt to understand and explain mistakes made in their programs. Python refers to these mistakes as errors and will point to the location where an error occurred with a ˆ character. When programs throw errors that we didn't expect to encounter we call those errors bugs. Programmers call the process of updating the program so that it no longer produces unexpected errors **debugging**.

Two common errors that we encounter while writing Python are **SyntaxError** and **NameError**.

- SyntaxError means there is something wrong with the way your program is written — punctuation that does not belong, a command where it is not expected, or a missing parenthesis can all trigger a SyntaxError.
- A NameError occurs when the Python interpreter sees a word it does not recognize. Code that contains something that looks like a variable but was never defined will throw a NameError.

You might encounter a SyntaxError if you open a string with double quotes and end it with a single quote. You might encounter a NameError if you try to print a single word string but fail to put any quotes around it.

Numbers

Computers can understand much more than just strings of text. Python has a few numeric data types. It has multiple ways of storing numbers. Which one you use depends on your intended purpose for the number you are saving.

- An integer, or int, is a whole number. It has no decimal point and contains all counting numbers (1, 2, 3, . . .) as well as their negative counterparts and the number 0. If you were counting the number of people in a room, the number of jellybeans in a jar, or the number of keys on a keyboard you would likely use an integer.
- A floating-point number, or a float, is a decimal number. It can be used to represent fractional quantities as well as precise measurements. If you were measuring the length of your bedroom wall, calculating the average test score of a seventh-grade class, or storing a baseball player's batting average for the 1998 season you would likely use a float.

Numbers can be assigned to variables or used literally in a program:

```
an_int = 2
a_float = 2.1

print(an_int + 3) # prints 5
```

```
## 5
```

Above we defined an integer and a float as the variables an_int and a_float. We printed out the sum of the variable an_int with the number 3. We call the number 3 here a *"literal"*, meaning it's actually the number 3 and not a variable with the number 3 assigned to it.

Floating-point numbers can behave in some unexpected ways due to how computers store them. For more information on floating-point numbers and Python, review Python's documentation on floating-point limitations.

Calculations

Computers absolutely excel at performing calculations. The "compute" in their name comes from their historical association with providing answers to mathematical questions. Python performs addition, subtraction, multiplication, and division with +, -, *, and /.

```
# Prints "500"
print(573 - 74 + 1)

# Prints "50"
```

```
## 500
```

```
print(25 * 2)

# Prints "2.0"
```

```
## 50
```

```
print(10 / 5)
```

```
## 2.0
```

Notice that when we perform division, the result has a decimal place. This is because Python converts all ints to floats before performing division. In older versions of Python (2.7 and earlier) this conversion did not happen, and integer division would always round down to the nearest integer.

Division can throw its own special error: ZeroDivisionError. Python will raise this error when attempting to divide by 0. Mathematical operations in Python follow the standard mathematical order of operations.

Changing Numbers

Variables that are assigned numeric values can be treated the same as the numbers themselves. Two variables can be added together, divided by 2, and multiplied by a third variable without Python distinguishing between the variables and literals (like the number 2 in this example). Performing arithmetic on variables does not change the variable — you can only update a variable using the = sign.

```
coffee_price = 1.50
number_of_coffees = 4

# Prints "6.0"
print(coffee_price * number_of_coffees)
# Prints "1.5"
```

```
## 6.0
```

```
print(coffee_price)
# Prints "4"
```

```
## 1.5
```

```
print(number_of_coffees)

# Updating the price
```

```
## 4
```

```
coffee_price = 2.00

# Prints "8.0"
print(coffee_price * number_of_coffees)
# Prints "2.0"
```

```
## 8.0
```

```
print(coffee_price)
# Prints "4"
```

```
## 2.0
```

```
print(number_of_coffees)
```

```
## 4
```

We create two variables and assign numeric values to them. Then we perform a calculation on them. This doesn't update the variables! When we update the coffee_price variable and perform the calculations again, they use the updated values for the variable!

Exponents

Python can also perform exponentiation. In written math, you might see an exponent as a superscript number, but typing superscript numbers isn't always easy on modern keyboards. Since this operation is so related to multiplication, we use the notation **.

```
# 2 to the 10th power, or 1024
print(2 ** 10)

# 8 squared, or 64
```

```
## 1024
```

```
print(8 ** 2)

# 9 * 9 * 9, 9 cubed, or 729
```

```
## 64
```

```
print(9 ** 3)

# We can even perform fractional exponents
# 4 to the half power, or 2
```

```
## 729
```

```
print(4 ** 0.5)
```

```
## 2.0
```

Module operator

Python offers a companion to the division operator called the modulo operator. The modulo operator is indicated by % and gives the remainder of a division calculation. If the number is divisible, then the result of the modulo operator will be 0.

```
# Prints 4 because 29 / 5 is 5 with a remainder of 4
print(29 % 5)

# Prints 2 because 32 / 3 is 10 with a remainder of 2
```

```
## 4
```

```
print(32 % 3)

# Modulo by 2 returns 0 for even numbers and 1 for odd numbers
# Prints 0
```

```
## 2
```

```
print(44 % 2)
```

```
## 0
```

The modulo operator is useful in programming when we want to perform an action every nth-time the code is run. Can the result of a modulo operation be larger than the divisor? Why or why not?

Concatenation

The + operator doesn't just add two numbers, it can also "add" two strings! The process of combining two strings is called **string concatenation**. Performing string concatenation creates a brand new string comprised of the first string's contents followed by the second string's contents (without any added space in-between).

```
greeting_text = "Hey there!"
question_text = "How are you doing?"
full_text = greeting_text + question_text

# Prints "Hey there!How are you doing?"
print(full_text)
```

```
## Hey there!How are you doing?
```

In this sample of code, we create two variables that hold strings and then concatenate them. But we notice that the result was missing a space between the two, let's add the space in-between using the same concatenation operator!

```
full_text = greeting_text + " " + question_text

# Prints "Hey there! How are you doing?"
print(full_text)
```

```
## Hey there! How are you doing?
```

If you want to concatenate a string with a number you will need to make the number a string first, using the **str()** function. If you're trying to print() a numeric variable you can use commas to pass it as a different argument rather than converting it to a string.

```
birthday_string = "I am "
age = 10
birthday_string_2 = " years old today!"

# Concatenating an integer with strings is possible if we turn the integer into a string first
full_birthday_string = birthday_string + str(age) + birthday_string_2

# Prints "I am 10 years old today!"
print(full_birthday_string)

# If we just want to print an integer
# we can pass a variable as an argument to
# print() regardless of whether
# it is a string.

# This also prints "I am 10 years old today!"
```

```
## I am 10 years old today!
```

```
print(birthday_string, age, birthday_string_2)
```

```
## I am  10  years old today!
```

Using str() we can convert variables that are not strings to strings and then concatenate them. But we don't need to convert a number to a string for it to be an argument to a print statement.

Plus Equals

Python offers a shorthand for updating variables. When you have a number saved in a variable and want to add to the current value of the variable, you can use the += (plus-equals) operator.

```
# First we have a variable with a number saved
number_of_miles_hiked = 12

# Then we need to update that variable
# Let's say we hike another two miles today
number_of_miles_hiked += 2

# The new value is the old value
# Plus the number after the plus-equals
print(number_of_miles_hiked)
# Prints 14
```

```
## 14
```

Above, we keep a running count of the number of miles a person has gone hiking over time. Instead of recalculating from the start, we keep a grand total and update it when we've gone hiking further. The plus-equals operator also can be used for string concatenation, like so:

```
hike_tweet = "What an amazing time to walk through nature!"

# Almost forgot the hashtags!
hike_tweet += " #nofilter"
hike_tweet += " #blessed"
print(hike_tweet)
```

```
## What an amazing time to walk through nature! #nofilter #blessed
```

Another example:

```
total_price = 0
# We put some new_sneakers into our chekout
new_sneakers = 50.00
total_price += new_sneakers

# Right before we check out, we spot a nice sweater and some fun books we also want to purchase!
nice_sweater = 39.00
fun_books = 20.00
# Update total_price:
total_price += (nice_sweater+fun_books )
print("The total price is", total_price)
```

```
## The total price is 109.0
```

Multi-line Strings

Python strings are very flexible, but if we try to create a string that occupies multiple lines we find ourselves face-to-face with a SyntaxError. Python offers a solution: multi-line strings.

By using three quote-marks (""" or ”’) instead of one, we tell the program that the string doesn't end until the next triple-quote. This method is useful if the string being defined contains a lot of quotation marks and we want to be sure we don't close it prematurely.

```
leaves_of_grass = """
Poets to come! orators, singers, musicians to come!
Not to-day is to justify me and answer what I am for,
But you, a new brood, native, athletic, continental, greater than
  before known,
Arouse! for you must justify me.
"""
```

In the above example, we assign a famous poet's words to a variable. Even though the quote contains multiple linebreaks, the code works!

If a multi-line string isn't assigned a variable or used in an expression it is treated as a comment.

Review

```
my_age = 24
half_my_age = my_age/2
greeting = "Hello"
```

```
name = "Diego"
greeting_with_name = greeting +" "+name
print(greeting_with_name)
```

```
## Hello Diego
```

Functions

Introduction

A function is a collection of several lines of code. By calling a function, we can call all of these lines of code at once, without having to repeat ourselves.

So, a function is a tool that you can use over and over again to produce consistent output from different inputs.We have already learned about one function, called print. We know that we call print by using this syntax:

```
# print(something_to_print)
```

In the rest of the lesson, we'll learn how to build more functions, call them with and without inputs, and return values from them.

What is a Function?

Let's imagine that we are creating a program that greets customers as they enter a grocery store. We want a big screen at the entrance of the store to say:

"Welcome to Engrossing Grocers. Our special is mandarin oranges. Have fun shopping!"

We have learned to use print statements for this purpose:

```
print("Welcome to Engrossing Grocers.")
```

```
## Welcome to Engrossing Grocers.
```

```
print("Our special is mandarin oranges.")
```

```
## Our special is mandarin oranges.
```

```
print("Have fun shopping!")
```

```
## Have fun shopping!
```

Every time a customer enters, we call these three lines of code. Even if only 3 or 4 customers come in, that's a lot of lines of code required. In Python, we can make this process easier by assigning these lines of code to a function.

We'll name this function greet_customer. In order to call a function, we use the syntax **function_name()**. The parentheses are important! They make the code inside the function run. In this example, the function call looks like: **greet_customer()**

Having this functionality inside greet_customer() is better form, because we have isolated this behavior from the rest of our code. Once we determine that greet_customer() works the way we want, we can reuse it anywhere and be confident that it greets, without having to look at the implementation. We can get the same output, with less repeated code. Repeated code is generally more error prone and harder to understand, so it's a good goal to reduce the amount of it.

```
def greet_customer():
  print("Welcome to Engrossing Grocers.")
  print("Our special is mandarin oranges.")
  print("Have fun shopping!")

greet_customer()
```

```
## Welcome to Engrossing Grocers.
## Our special is mandarin oranges.
## Have fun shopping!
```

Write a Function

We have seen the value of simple functions for modularizing code. Now we need to understand how to write a function. To write a function, you must have a heading and an indented block of code. The heading starts with the keyword def and the name of the function, followed by parentheses, and a colon. The indented block of code performs some sort of operation. This syntax looks like:

```
# def function_name():
#  some code
```

The keyword def tells Python that we are defining a function. This function is called greet_customer. Everything that is indented after the : is what is run when greet_customer() is called. So every time we call greet_customer(), the three print statements run.

Whitespace

Consider this function:

```
def greet_customer():
  print("Welcome to Engrossing Grocers.")
  print("Our special is mandarin oranges.")
  print("Have fun shopping!")
```

The three print statements are all executed together when greet_customer() is called. This is because they have the same level of indentation. In Python, the amount of whitespace tells the computer what is part of a function and what is not part of that function. If we wanted to write another line outside of greet_customer(), we would have to unindent the new line:

```
def greet_customer():
  print("Welcome to Engrossing Grocers.")
  print("Our special is mandarin oranges.")
  print("Have fun shopping!")
print("Cleanup on Aisle 6")
```

```
## Cleanup on Aisle 6
```

When we call greet_customer, the message“Cleanup on Aisle 6” is not printed, as it is not part of the function.

Here we use tab for our default indentation. Anything other than that will throw an error when you try to run the program. Many other platforms use 4 spaces. Some people even use one! These are all fine. What is important is being consistent throughout the project.

Parameters

Let’s return to “Engrossing Grocers”. The special of the day will not always be mandarin oranges, it will change every day. What if we wanted to call these three print statements again, except with a variable special? We can use parameters, which are variables that you can pass into the function when you call it.

```
def greet_customer(special_item):
  print("Welcome to Engrossing Grocers.")
  print("Our special is " + special_item + ".")
  print("Have fun shopping!")
```

In the definition heading for greet_customer(), the special_item is referred to as a **formal parameter.** This variable name is a placeholder for the name of the item that is the grocery's special today. Now, when we call greet_customer, we have to provide a special_item. That item will get printed out in the second print statement:

```
greet_customer("peanut butter")
```

```
## Welcome to Engrossing Grocers.
## Our special is peanut butter.
## Have fun shopping!
```

The value between the parentheses when we call the function (in this case, "peanut butter") is referred to as an argument of the function call. The argument is the information that is to be used in the execution of the function.

When we then call the function, Python assigns the formal parameter name special_item with the actual parameter data, "peanut_butter". In other words, it is as if this line was included at the top of the function: *special_item = "peanut butter"*

Every time we call greet_customer() with a different value between the parentheses, special_item is assigned to hold that value.

Another example:

The function mult_two_add_three() prints a number multiplied by 2 and added to 3. As it is written right now, the number that it operates on is always 5.

```
def mult_two_add_three():
  number = 5
  print(number*2 + 3)

mult_two_add_three()
```

```
## 13
```

If we modify so that the **number** variable is a parameter of the function and then pass any number into the function call:

```
def mult_two_add_three(number):
  print(number*2 + 3)
# Call mult_two_add_three() here:
mult_two_add_three(2)
```

```
## 7
```

Multiple Parameters

Our grocery greeting system has gotten popular, and now other supermarkets want to use it. As such, we want to be able to modify both the special item and the name of the grocery store in a greeting like this:

Welcome to [grocery store]. Our special is [special item]. Have fun shopping!

We can make a function take more than one parameter by using commas:

```
def greet_customer(grocery_store, special_item):
  print("Welcome to "+ grocery_store + ".")
  print("Our special is " + special_item + ".")
  print("Have fun shopping!")
```

```
greet_customer("Stu's Staples", "papayas")
```

```
## Welcome to Stu's Staples.
## Our special is papayas.
## Have fun shopping!
```

Another example:

```
def mult_x_add_y(number,x,y):
  print(number*x + y)
mult_x_add_y (5,2,3)
```

```
## 13
```

Keyword Arguments

In our greet_customer() function from the last exercise, we had two arguments:

```
def greet_customer(grocery_store, special_item):
  print("Welcome to "+ grocery_store + ".")
  print("Our special is " + special_item + ".")
  print("Have fun shopping!")
```

Whichever value is put into greet_customer() first is assigned to grocery_store, and whichever value is put in second is assigned to special_item. These are called **positional arguments** because their assignments depend on their positions in the function call.

We can also pass these arguments as keyword arguments, where we explicitly refer to what each argument is assigned to in the function call.

```
greet_customer(special_item="chips and salsa", grocery_store="Stu's Staples")
```

```
## Welcome to Stu's Staples.
## Our special is chips and salsa.
## Have fun shopping!
```

We can use keyword arguments to make it explicit what each of our arguments to a function should refer to in the body of the function itself.

We can also define **default arguments** for a function using syntax very similar to our keyword-argument syntax, but used during the function definition. If the function is called without an argument for that parameter, it relies on the default.

```
def greet_customer(special_item, grocery_store="Engrossing Grocers"):
  print("Welcome to "+ grocery_store + ".")
  print("Our special is " + special_item + ".")
  print("Have fun shopping!")
```

In this case, grocery_store has a default value of "Engrossing Grocers". If we call the function with only one argument, the value of "Engrossing Grocers" is used for grocery_store:

```
greet_customer("bananas")
```

```
## Welcome to Engrossing Grocers.
## Our special is bananas.
## Have fun shopping!
```

Once you give an argument a default value (making it a keyword argument), no arguments that follow can be used positionally. For example:

```
  # This is not valid
#def greet_customer(special_item="bananas", grocery_store):
#  print("Welcome to "+ grocery_store + ".")
#  print("Our special is " + special_item + ".")
#  print("Have fun shopping!")
```

```
  # This is valid
def greet_customer(special_item, grocery_store="Engrossing Grocers"):
  print("Welcome to "+ grocery_store + ".")
  print("Our special is " + special_item + ".")
  print("Have fun shopping!")
```

Anothe example:

```
# Define create_spreadsheet(): note that we need to convert row_count into string.
def create_spreadsheet(title,row_count=1000):
  print("Creating a spreadsheet called "+title +" " + "with"+" " + str(row_count) +" " +"rows")

# Call create_spreadsheet() below with the required arguments:
create_spreadsheet("Applications",5)
```

```
## Creating a spreadsheet called Applications with 5 rows
```

Returns

So far, we have only seen functions that print out some result to the console. Functions can also return a value to the user so that this value can be modified or used later. When there is a result from a function that can be stored in a variable, it is called a returned function value. We use the keyword return to do this.

Here's an example of a function **divide_by_four** that takes an integer argument, divides it by four, and returns the result:

```
def divide_by_four(input_number):
  return input_number/4
```

The program that calls divide_by_four can then use the result later and even reuse the divide_by_four function.

```
result = divide_by_four(16)
# result now holds 4
print("16 divided by 4 is " + str(result) + "!")
```

```
## 16 divided by 4 is 4.0!
```

```
result2 = divide_by_four(result)
print(str(result) + " divided by 4 is " + str(result2) + "!")
```

```
## 4.0 divided by 4 is 1.0!
```

In this example, we returned a number, but we could also return a String:

```
def create_special_string(special_item):
  return "Our special is" + special_item + "."

special_string = create_special_string("banana yogurt")

print(special_string)
```

```
## Our special isbanana yogurt.
```

Another example:

The function **calculate_age** creates a variable called **age** that is the difference between the current year, and a birth year, both of which are inputs of the function.

```
def calculate_age(current_year, birth_year):
  age = current_year - birth_year
  return age

my_age=calculate_age (2020,1995)

dads_age=calculate_age (2020,1965)

print("I am " + str(my_age) + " years old and my dad is " + str(dads_age) + " years old.")
```

```
## I am 25 years old and my dad is 55 years old.
```

Multiple Return Values

Sometimes we may want to return more than one value from a function. We can return several values by separating them with a comma:

```
def square_point(x_value, y_value):
  x_2 = x_value * x_value
  y_2 = y_value * y_value
  return x_2, y_2
```

This function takes in an x value and a y value, and returns them both, squared. We can get those values by assigning them both to variables when we call the function:

```
x_squared, y_squared = square_point(2,3)
print(x_squared)
```

```
## 4
```

```
print(y_squared)
```

```
## 9
```

Another example:

Function get_boundaries() takes in two parameters, a number called target and a number called margin. Then we create two variables: *low_limit*: target minus the margin and *high_limit*: margin added to target

```
def get_boundaries(target, margin):
  low_limit=target-margin
  high_limit=target+margin
  return low_limit, high_limit

low, high = get_boundaries(100,20)
print(low,high)
```

```
## 80 120
```

Scope

Let's say we have our function from the last exercise that creates a string about a special item:

```
def create_special_string(special_item):
  return "Our special is " + special_item + "."
```

What if we wanted to access the variable special_item outside of the function? Could we use it?

```
#def create_special_string(special_item):
#  return "Our special is " + special_item + "."

#print("I don't like " + special_item)
```

If we try to run this code, we will get a NameError, telling us that 'special_item' is not defined. The variable special_item has only been defined inside the space of a function, so it does not exist outside the function.

We call the part of a program where special_item can be accessed its **scope**. The scope of special_item is only the create_special_string function.

Variables defined outside the scope of a function may be accessible inside the body of the function:

```
header_string = "Our special is "

def create_special_string(special_item):
  return header_string + special_item + "."
print(create_special_string("grapes"))
```

```
## Our special is grapes.
```

There is no error here. header_string can be used inside the create_special_string function because the scope of header_string is the whole file.

Review

So far you have learned:

- How to write a function
- How to give a function inputs
- How to return values from a function
- What scope means

Example: We will want to make the function **repeat_stuff** print a string with stuff repeated num_repeats amount of times. Note: Multiplying a string just makes a new string with the old one repeated! For example:

```
# num_repeats has a default value of 10.
def repeat_stuff(stuff,num_repeats=10):
  return stuff*num_repeats
# We use the function a first time into lyrics
lyrics = repeat_stuff("Row ",3) + "Your Boat. "
# We use the function a second time into song with default value
song = repeat_stuff(lyrics)

print(song)
```

```
## Row Row Row Your Boat. Row Row Row Your Boat. Row Row Row Your Boat. Row Row Row Your Boat. Row Row
```

Control Flow

Introduction

Imagine waking up in the morning.

You wake up and think,

"Ugh, is it a weekday?"

If so, you have to get up and get dressed and get ready for work or school. If not, you can sleep in a bit longer and catch a couple extra Z's. But alas, it is a weekday, so you are up and dressed and you go to look outside, "What's the weather like? Do I need an umbrella?"

These questions and decisions control the flow of your morning, each step and result is a product of the conditions of the day and your surroundings. Your computer, just like you, goes through a similar flow every time it executes code. A program will run (wake up) and start moving through its checklists, is this condition met, is that condition met, okay let's execute this code and return that value.

This is the **Control Flow** of your program. In Python, your script will execute from the top down, until there is nothing left to run. It is your job to include gateways, known as conditional statements, to tell the computer when it should execute certain blocks of code. If these conditions are met, then run this function.

We will learn how to build conditional statements using boolean expressions, and manage the control flow in your code.

Boolean Expressions

In order to build control flow into our program, we want to be able to check if something is true or not. A boolean expression is a statement that can either be True or False.

Let's go back to the 'waking up' example. The first question, "Is today a weekday?" can be written as a boolean expression:

```
# Today is a weekday.
```

This expression can be True if today is Tuesday, or it can be False if today is Saturday. There are no other options.

Consider the phrase:

```
# Friday is the best day of the week.
```

Is this a boolean expression?

No, this statement is an opinion and is not objectively True or False. Someone else might say that "Wednesday is the best weekday," and their statement would be no less True or False than the one above.

How about the phrase:

```
# Sunday starts with the letter 'C'.
```

This expression can only be True or False, which makes it a boolean expression. Even though the statement itself is false (Sunday starts with the letter 'C'), it is still a boolean expression.

Relational Operators

Now that we understand what boolean expressions are, let's learn to create them in Python. We can create a boolean expression by using relational operators. Relational operators compare two items and return either True or False. For this reason, you will sometimes hear them called *comparators*.

The two boolean operators we'll cover first are:

- Equals: ==
- Not equals: !=

These operators compare two items and return True or False if they are equal or not.

We can create boolean expressions by comparing two values using these operators:

```
1 == 1
```

```
## True
```

```
2 != 4
```

```
## True
```

```
3 == 5
```

```
## False
```

```
'7' == 7
```

```
## False
```

Why is the last statement false? The " marks in '7' make it a string, which is different from the integer value 7, so they are not equal. When using relational operators it is important to always be mindful of type.

Note that some Python consoles use >>> as the prompt when you run Python in your terminal, which you can then use to evaluate simple expressions, such as these.

Boolean Variables

Before we go any further, let's talk a little bit about **True** and **False**. You may notice that when you type them in the code editor (with uppercase T and F), they appear in a different color than variables or strings. This is because True and False are their own special type: bool.

True and False are the only bool types, and any variable that is assigned one of these values is called a boolean variable. Boolean variables can be created in several ways. The easiest way is to simply assign True or False to a variable:

```
set_to_true = True
set_to_false = False
```

You can also set a variable equal to a boolean expression.

```
bool_one = 5 != 7
bool_two = 1 + 1 != 2
bool_three = 3 * 3 == 9
```

These variables now contain boolean values, so when you reference them they will only return the True or False values of the expression they were assigned.

```
bool_one
```

```
## True
```

```
bool_two
```

```
## False
```

```
bool_three
```

```
## True
```

Example:

Setting my_baby_bool equal to "true" and checking it's type with type() function:

```
my_baby_bool= "true"
print(type(my_baby_bool))
```

```
## <class 'str'>
```

It's not a boolean variable! Boolean values True and False always need to be capitalized and do not have quotation marks.

Check this out:

```
my_baby_bool_two = True
print(type(my_baby_bool_two))
```

```
## <class 'bool'>
```

If Statements

Understanding boolean variables and expressions is essential because they are the building blocks of **conditional statements**.

Recall the waking-up example from the beginning of this lesson. The decision-making process of "Is it raining? If so, bring an umbrella" is a conditional statement. Here it is phrased in a different way: **If it is raining then bring an umbrella.**

Can you pick out the boolean expression here? If **"it is raining" == True** then the rest of the conditional statement will be executed and you will bring an umbrella.

This is the form of a conditional statement: **If** [**it is raining**] **then** [**bring an umbrella**] In Python, it looks very similar:

```
# if is_raining:
#   bring_umbrella()
```

You'll notice that instead of "then" we have a **colon, :**. That tells the computer that what's coming next is what should be executed if the condition is met. Let's take a look at another conditional statement

```
if 2 == 4 - 2:
  print("apple")
```

```
## apple
```

Will this code print apple to the terminal? Yes, because the condition of the if statement, 2 == 4 - 2 is True.

Another example: my coworker Dave kept using my computer without permission and he is a real doofus. It takes user_name as an input and if the user is Dave it tells him to stay off my computer. Dave got around my security and has been logging onto my computer using our coworker Angela's user name, *Angela*.

```
def dave_check(user_name):
  if user_name == "Diego":
    return "Get off my computer Dave!"
  if user_name == "Angela":
    return "I know it is you Diego! Go away!"


# Enter a user name here, make sure to make it a string
user_name = "Angela"

print(dave_check(user_name))
```

```
## I know it is you Diego! Go away!
```

Relational Operators II

Now that we've added conditional statements to our toolkit for building control flow, let's explore more ways to create boolean expressions. So far we know two relational operators, equals and not equals, but there are a ton (well, four) more:

- Greater than: >
- Less than: <

- Greater than or equal to: >=
- Less than or equal to: <=

Let's say we're running a movie streaming platform and we want to write a function that checks if our users are over 13 when showing them a PG-13 movie. We could write something like:

```
def age_check(age):
  if age >= 13:
    return True
age = 24
print(age_check(age))
```

```
## True
```

Another example: A function called greater_than that takes two integer inputs, x and y and returns the value that is greater. If x and y are equal, return the string "These numbers are the same"

```
def greater_than(x,y):
  if x>y:
     return x
  if x<y:
     return y
  if x==y:
     return "These numbers are the same"
print(greater_than(4,4))
```

```
## These numbers are the same
```

Boolean Operators: and

Often, the conditions you want to check in your conditional statement will require more than one boolean expression to cover. In these cases, you can build larger boolean expressions using boolean operators. These operators (also known as logical operators) combine smaller boolean expressions into larger boolean expressions.

There are three boolean operators that we will cover:

- and
- or
- not

Let's start with **and**. and combines two boolean expressions and evaluates as True if both its components are True, but False otherwise.

Consider the example: *Oranges are a fruit and carrots are a vegetable.*

This boolean expression is comprised of two smaller expressions, oranges are a fruit and carrots are a vegetable, both of which are True and connected by the boolean operator and, so the entire expression is True.

Let's look at an example of some AND statements in Python:

```
(1 + 1 == 2) and (2 + 2 == 4)
#True
```

```
## True
```

```
(1 + 1 == 2) and (2 < 1)
#False
```

```
## False
```

```
(1 > 9) and (5 != 6)
#False
```

```
## False
```

```
(0 == 10) and (1 + 1 == 1)
#False
```

```
## False
```

Notice that in the second and third examples, even though part of the expression is True, the entire expression as a whole is False because the other statement is False. The fourth statement is also False because both components are False.

Example: In a College 120 credits aren't the only graduation requirement, you also need to have a GPA of 2.0 or higher.

```
def graduation_reqs(gpa,credits):
  if credits >= 120 and gpa >= 2.0:
    return "You meet the requirements to graduate!"

print(graduation_reqs(2,120))
```

```
## You meet the requirements to graduate!
```

Boolean Operators: or

The boolean operator or combines two expressions into a larger expression that is True if either component is True.

Consider the statement *"Oranges are a fruit or apples are a vegetable."*

This statement is composed of two expressions: oranges are a fruit which is True and apples are a vegetable which is False. Because the two expressions are connected by the or operator, the entire statement is True. Only one component needs to be True for an or statement to be True.

In English, or implies that if one component is True, then the other component must be False. This is not true in Python. If an or statement has two True components, it is also True.

Let's take a look at a couple example in Python:

```
True or (3 + 4 == 7)
#True
```

```
## True
```

```
(1 - 1 == 0) or False
#True
```

```
## True
```

```
(2 < 0) or True
#True
```

```
## True
```

```
(3 == 8) or (3 > 4)
#False
```

```
## False
```

Notice that each or statement that has at least one True component is True, but the final statement has two False components, so it is False.

Boolean Operators: not

The final boolean operator we will cover is not. This operator is straightforward: when applied to any boolean expression it reverses the boolean value. So if we have a True statement and apply a not operator we get a False statement.

```
# not True == False
# not False == True
```

Consider the following statement: *"Oranges are not a fruit"*. Here, we took the True statement oranges are a fruit and added a not operator to make the False statement oranges are not a fruit.

This example in English is slightly different from the way it would appear in Python because in Python we add the not operator to the very beginning of the statement. Let's take a look at some of those:

```
not 1 + 1 == 2
# False
```

```
## False
```

```
not 7 < 0
# True
```

```
## True
```

Example:

```
def graduation_reqs(gpa, credits):
  if (gpa >= 2.0) and (credits >= 120):
    return "You meet the requirements to graduate!"
  if (gpa >= 2.0) and not (credits >= 120):
    return "You do not have enough credits to graduate."
  if not (gpa >= 2.0) and (credits >= 120):
    return "Your GPA is not high enough to graduate."
  if not (gpa >= 2.0) and not (credits >= 120):
    return  "You do not meet either requirement to graduate!"

print(graduation_reqs(3, 100))
```

```
## You do not have enough credits to graduate.
```

Else Statements

As you can tell from your work with Calvin Coolidge's Cool College, once you start including lots of if statements in a function the code becomes a little cluttered and clunky. Luckily, there are other tools we can use to build control flow. **else** statements allow us to elegantly describe what we want our code to do when certain conditions are not met.

else statements always appear in conjunction with if statements. Consider our waking-up example to see how this works:

```
#  if weekday:
#    wake_up("6:30")
#  else:
#    sleep_in()
```

In this way, we can build if statements that execute different code if conditions are or are not met. This prevents us from needing to write if statements for each possible condition, we can instead write a blanket else statement for all the times the condition is not met.

Let's return to our age_check function for our movie streaming platform. Previously, all it did was check if the user's age was over 13 and if so return True. We can use an else statement to return a message in the event the user is too young to watch the movie.

```
def age_check(age):
  if age >= 13:
    return True
  else:
    return "Sorry, you must be 13 or older to watch this movie."

print(age_check(12))
```

```
## Sorry, you must be 13 or older to watch this movie.
```

Back to the graduation example we could use else for the last scenario:

```
def graduation_reqs(gpa, credits):
  if (gpa >= 2.0) and (credits >= 120):
    return "You meet the requirements to graduate!"
  if (gpa >= 2.0) and not (credits >= 120):
    return "You do not have enough credits to graduate."
  if not (gpa >= 2.0) and (credits >= 120):
    return "Your GPA is not high enough to graduate."
  else:
    return "You do not meet the GPA or the credit requirement for graduation."

print(graduation_reqs(1, 100))
```

```
## You do not meet the GPA or the credit requirement for graduation.
```

Else If Statements

We have if statements, we have else statements, we can also have **elif** statements. It's exactly what it sounds like, "else if". An elif statement checks another condition after the previous if statements conditions aren't met. We can use elif statements to control the order we want our program to check each of our conditional statements. First, the if statement is checked, then each elif statement is checked from top to bottom, then finally the else code is executed if none of the previous conditions have been met.

Let's take a look at this in practice. The following function will display a "thank you" message after someone donates to a charity: It takes the donation amount and prints a message based on how much was donated.

```
def thank_you(donation):
  if donation >= 1000:
    print("Thank you for your donation! You have achieved platinum donation status!")
  elif donation >= 500:
    print("Thank you for your donation! You have achieved gold donation status!")
  elif donation >= 100:
    print("Thank you for your donation! You have achieved silver donation status!")
  else:
    print("Thank you for your donation! You have achieved bronze donation status!")

thank_you(600)
```

```
## Thank you for your donation! You have achieved gold donation status!
```

Take a second to think about this function. What would happen if all of the elif statements were simply if statements? If you donated $1000.00, then the first three messages would all print because each if condition had been met.

But because we used elif statements, it checks each condition sequentially and only prints one message. If I donate $600.00, the code first checks if that is over $1000.00, which it is not, then it checks if it's over $500.00, which it is, so it prints that message, then because all of the other statements are elif and else, none of them get checked and no more messages get printed.

Example:

Calvin Coolidge's Cool College has noticed that students prefer to get letter grades over GPA numbers. They want you to write a function called grade_converter that converts an inputted GPA into the appropriate letter grade. Your function should be named grade_converter, take the input gpa, and convert the following GPAs: 4.0 or higher should return "A" , 3.0 or higher should return "B" , 2.0 or higher should return "C" , 1.0 or higher should return "D" , 0.0 or higher should return "F".

```
def grade_converter(gpa):
  grade = "F"

  if gpa >= 4.0:
    grade = "A"
  elif gpa >= 3.0:
    grade = "B"
  elif gpa >= 2.0:
    grade = "C"
  elif gpa >= 1.0:
    grade = "D"
  return grade

print(grade_converter(3))
```

```
## B
```

Try and Except Statements

Notice that if, elif, and else statements aren't the only way to build a control flow into your program. You can use **try** and **except** statements to check for possible errors that a user might encounter.

The general syntax of a try and except statement is

```
#  try:
#      # some statement
#  except ErrorName:
#      # some statement
```

First, the statement under try will be executed. If at some point an exception is raised during this execution, such as a NameError or a ValueError and that exception matches the keyword in the except statement, then the try statement will terminate and the except statement will execute.

Let's take a look at this in an application. I want to write a function that takes two numbers, a and b as an input and then returns a divided by b. But, there is a possibility that b is zero, which will cause an error, so I want to include a try and except flow to catch this error.

```
def divides(a,b):
  try:
    result = a / b
    print (result)
  except ZeroDivisionError:
    print ("Can't divide by zero!")

divides(3,4)
```

```
## 0.75
```

```
divides(2,0)
```

```
## Can't divide by zero!
```

Another example: The following function is very simple and serves one purpose: it raises a ValueError. We write a try statement and an except statement around the line of code that executes the function to catch a ValueError and make the error message print You raised a ValueError!

```
def raises_value_error():
    raise ValueError
try:
  raises_value_error()
except ValueError:
  print("You raised a ValueError!")
```

```
## You raised a ValueError!
```

Review

- Boolean expressions are statements that can be either True or False
- A boolean variable is a variable that is set to either True or False.
- You can create boolean expressions using relational operators:
- Equals: ==
- Not equals: !=
- Greater than: >
- Greater than or equal to: >=
- Less than: <
- Less than or equal to: <=
- if statements can be used to create control flow in your code.
- else statements can be used to execute code when the conditions of an if statement are not met.
- elif statements can be used to build additional checks into your if statements
- try and except statements can be used to build error control into your code.

Example:

The admissions office at Calvin Coolidge's Cool College has heard about your programming prowess and wants to get a piece of it for themselves. They've been inundated with applications and need a way to automate the filtering process. They collect three pieces of information for each applicant:

1. Their high school GPA, on a 0.0 - 4.0 scale.
2. Their personal statement, which is given a score on a 1 - 100 scale.
3. The number of extracurricular activities they participate in.

The admissions office has a cutoff point for each category. They want students that have a GPA of 3.0 or higher, a personal statement with a score of 90 or higher, and who participated in 3 or more extracurricular activities. The admissions office also wants to give students who have a high GPA and a strong personal statement a chance even if they don't participate in enough extracurricular activities. For all other cases, application should be rejected.

We write a function called applicant_selector which takes three inputs, gpa, ps_score, and ec_count.

```
def applicant_selector(gpa,ps_score,ec_count):
  if gpa >= 3 and ps_score >= 90 and ec_count >= 3:
    return "This applicant should be accepted."
  elif gpa >= 3 and ps_score >= 90 and not ec_count >= 3:
    return "This applicant should be given an in-person interview."
  else:
    return "This applicant should be rejected."

applicant_selector(4,100,2)
```

```
## 'This applicant should be given an in-person interview.'
```

List

What is a list?

A list is an ordered set of objects in Python.

Suppose we want to make a list of the heights of students in a class:

- Jenny is 61 inches tall
- Alexus is 70 inches tall
- Sam is 67 inches tall
- Grace is 64 inches tall

In Python, we can create a variable called heights to store these numbers:

```
heights = [61, 70, 67, 64]
```

Notice that:

- A list begins and ends with square brackets ([and]).
- Each item (i.e., 67 or 70) is separated by a comma (,)
- It's considered good practice to insert a space () after each comma, but your code will run just fine if you forget the space.

Lists can contain more than just numbers. Let's revisit our height example. We can make a list of strings that contain the students' names:

```
names = ['Jenny', 'Alexus', 'Sam', 'Grace']
```

We can also combine multiple data types in one list. For example, this list contains both a string and an integer:

```
mixed_list = ['Jenny', 61]
```

List of Lists

We've seen that the items in a list can be numbers or strings. They can also be other lists! Previously, we saw that we could create a list representing both Jenny's name and height:

```
jenny = ['Jenny', 61]
```

We can put several of these lists into one list, such that each entry in the list represents a student and their height:

```
heights = [['Jenny', 61], ['Alexus', 70], ['Sam', 67], ['Grace', 64]]
```

Zip

Again, let's return to our class height example. Suppose that we already had a list of names and a list of heights:

```
names = ['Jenny', 'Alexus', 'Sam', 'Grace']
heights = [61, 70, 67, 65]
```

If we wanted to create a list of lists that paired each name with a height, we could use the command **zip.** zip takes two (or more) lists as inputs and returns an object that contains a list of pairs. Each pair contains one element from each of the inputs. You won't be able to see much about this object from just printing it, because it will return the location of this object in memory. Output would look something like this:

```
names_and_heights = zip(names, heights)
print(names_and_heights)
```

```
## <zip object at 0x7ff53999b440>
```

To see the nested lists, you can convert the zip object to a list first:

```
print(list(names_and_heights))
```

```
## [('Jenny', 61), ('Alexus', 70), ('Sam', 67), ('Grace', 65)]
```

Empty Lists

A list doesn't have to contain anything! You can create an empty list like this:

```
empty_list = []
```

Why would we create an empty list?

Usually, it's because we're planning on filling it later based on some other input. We'll talk about two ways of filling up a list in the next exercise.

Growing a List: Append

We can add a single element to a list using **.append()**. For example, suppose we have an empty list called empty_list:

```
empty_list = []
```

We can add the element 1 using the following commands:

```
empty_list.append(1)
```

If we examine empty_list, we see that it now contains 1:

```
print(empty_list)
```

```
## [1]
```

When we use .append() on a list that already has elements, our new element is added to the end of the list:

```
# Create a list
my_list = [1, 2, 3]

# Append a number
my_list.append(5)
print(my_list) # check the result
```

```
## [1, 2, 3, 5]
```

Note: It's important to remember that .append() comes after the list. This is different from functions like print, which come before. Also note that .append() takes exactly one argument: a string, a number or another list.

Growing a List: Plus (+)

When we want to add multiple items to a list, we can use + to combine two lists. Below, we have a list of items sold at a bakery called items_sold:

```
items_sold = ['cake', 'cookie', 'bread']
```

Suppose the bakery wants to start selling 'biscuit' and 'tart':

```
items_sold_new = items_sold + ['biscuit', 'tart']
print(items_sold_new)
```

```
## ['cake', 'cookie', 'bread', 'biscuit', 'tart']
```

In this example, we created a new variable, items_sold_new, which contained both the original items sold, and the new ones. We can inspect the original items_sold and see that it did not change. We can only use + with other lists. If we type in this code we will get the following error:

```
my_list = [1, 2, 3]
# my_list + 4
## TypeError: can only concatenate list (not "int") to list
print(my_list)
```

```
## [1, 2, 3]
```

If we want to add a single element using +, we have to put it into a list with brackets ([]):

```
new_list = my_list + [4]
print(new_list)
```

```
## [1, 2, 3, 4]
```

Range

Often, we want to create a list of consecutive numbers. For example, suppose we want a list containing the numbers 0 through 9:

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Typing out all of those numbers takes time and the more numbers we type, the more likely it is that we have a typo.

Python gives us an easy way of creating these lists using a function called range. The function range takes a single input, and generates numbers starting at 0 and ending at the number before the input. So, if we want the numbers from 0 through 9, we use range(10) because 10 is 1 greater than 9:

```
my_range = range(10)
```

Just like with zip, the range function returns an object that we can convert into a list:

```
print(my_list)
```

```
## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(list(my_range))
```

```
## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can use range to generate more interesting lists. By default, range creates a list starting at 0. However, if we call range with two arguments, we can create a list that starts at a different number. For example, range(2, 9) would generate numbers starting at 2 and ending at 8 (just before 9):

```
my_list = range(2, 9)
print(list(my_list))
```

```
## [2, 3, 4, 5, 6, 7, 8]
```

With one or two arguments, range will create a list of consecutive numbers (i.e., each number is one greater than the previous number). If we use a third argument, we can create a list that "skips" numbers. For example, range(2, 9, 2) will give us a list where each number is 2 greater than the previous number:

```
my_range2 = range(2, 9, 2)
print(list(my_range2))
```

```
## [2, 4, 6, 8]
```

```
my_range3 = range(1, 100, 10)
print(list(my_range3))
```

```
## [1, 11, 21, 31, 41, 51, 61, 71, 81, 91]
```

Our list stops at 91 because the next number in the sequence would be 101, which is greater than 100 (our stopping point).

Review

Now we know:

- How to create a list
- How to create a list of lists using zip
- How to add elements to a list using either .append() or +
- How to use range to create lists of integers

Working lists

Operations on Lists

Now that we know how to create a list, we can start working with existing lists of data.

In this section, we'll learn how to:

- Get the length of a list
- Select subsets of a list (called slicing)
- Count the number of times that an element appears in a list
- Sort a list of items

Length of a List

Often, we'll need to find the number of items in a list, usually called its length. We can do this using the function len. When we apply len to a list, we get the number of elements in that list:

```
my_list = [1, 2, 3, 4, 5]
print(len(my_list))
```

```
## 5
```

Selecting List Elements

Chris is interviewing candidates for a job. He will call each candidate in order, represented by a Python list:

```
calls = ['Ali', 'Bob', 'Cam', 'Doug', 'Ellie']
```

First, he'll call 'Ali', then 'Bob', etc. In Python, we call the order of an element in a list its **index**. Python lists are **zero-indexed**. This means that the **first element in a list has index 0**, rather than 1.

Here are the index numbers for that list:

Element	Index
'Ali'	0
'Bob'	1
'Cam'	2
'Doug'	3
'Ellie'	4

In this example, the element with index 2 is 'Cam'. We can select a single element from a list by using square brackets ([]) and the index of the list item. For example, if we wanted to select the third element from the list, we'd use calls[2]:

```
print(calls[2])
```

```
## Cam
```

Selecting an element that does not exist produces an "IndexError: list index out of range".

What if we want to select the last element of a list?

We can use the index -1 to select the last item of a list, even when we don't know how many elements are in a list.

Consider the following list with 5 elements:

```
list1 = ['a', 'b', 'c', 'd', 'e']
```

If we select the -1 element, we get the final element, 'e'. This is the same as selecting the element with index 4:

```
print(list1[-1])
```

```
## e
```

Slicing Lists

Suppose we have a list of letters:

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Suppose we want to select from b through f. We can do this using the following syntax: **letters[start:end]**, where: - start is the index of the first element that we want to include in our selection. In this case, we want to start at b, which has index 1. - end is the index of one more than the last index that we want to include. The last element we want is f, which has index 5, so end needs to be 6.

```
sublist = letters[1:6]
print(sublist)
```

```
## ['b', 'c', 'd', 'e', 'f']
```

Notice that the element at index 6 (which is g) is not included in our selection. Creating a selection from a list is called slicing.

Example:

We have the following list:

```
suitcase = ['shirt', 'shirt', 'pants', 'pants', 'pajamas', 'books']
```

beginning selects the first 4 elements of suitcase.

```
beginning = suitcase[0:4]
print(beginning)
```

```
## ['shirt', 'shirt', 'pants', 'pants']
```

middle that contains the middle two items from suitcase.

```
middle = suitcase[2:4]
print(middle)
```

```
## ['pants', 'pants']
```

If we want to select the first 3 elements of a list, we could use the following code:

```
fruits = ['apple', 'banana', 'cherry', 'date']
print(fruits[0:3])
```

```
## ['apple', 'banana', 'cherry']
```

When starting at the beginning of the list, it is also valid to omit the 0:

```
print(fruits[:3])
```

```
## ['apple', 'banana', 'cherry']
```

We can do something similar when selecting the last few items of a list. We can omit the final index when selecting the final elements from a list.

```
print(fruits[2:])
```

```
## ['cherry', 'date']
```

If we want to select the last 3 elements of fruits, we can also use this syntax:

```
print(fruits[-3:])
```

```
## ['banana', 'cherry', 'date']
```

We can use negative indexes to count backward from the last element.

Counting elements in a list

Suppose we have a list called letters that represents the letters in the word “Mississippi”:

```
letters = ['m', 'i', 's', 's', 'i', 's', 's', 'i', 'p', 'p', 'i']
```

If we want to know how many times i appears in this word, we can use the function **.count** that goes after the list name.

```
num_i = letters.count('i')
print(num_i)
```

```
## 4
```

Another example:

Mrs. WIlson’s class is voting for class president. She has saved each student’s vote into the list votes. How many votes does Jake have?

```
votes = ['Jake', 'Jake', 'Laurie', 'Laurie', 'Laurie', 'Jake', 'Jake', 'Jake', 'Laurie', 'Cassie', 'Cas
jake_votes = votes.count('Jake')
print(jake_votes)
```

```
## 9
```

Sorting lists 1

Sometimes, we want to sort a list in either numerical (1, 2, 3, . . .) or alphabetical (a, b, c, . . .) order. We can sort a list *in place* using **.sort()**. Suppose that we have a list of names:

```
names = ['Xander', 'Buffy', 'Angel', 'Willow', 'Giles']
print(names)
```

```
## ['Xander', 'Buffy', 'Angel', 'Willow', 'Giles']
```

Now we apply .sort():

```
names.sort()
print(names)
```

```
## ['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']
```

Notice that sort goes after our list, names. If we try sort(names), we will get a NameError.

sort does not return anything. So, if we try to assign names.sort() to a variable, our new variable would be None:

```
sorted_names = names.sort()
print(sorted_names)
```

```
## None
```

Although sorted_names is None, the line sorted_names = names.sort() still edited names:

```
print(names)
```

```
## ['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']
```

More examples:

```
addresses = ['221 B Baker St.', '42 Wallaby Way', '12 Grimmauld Place', '742 Evergreen Terrace', '1600 
# Sort addresses:
addresses.sort()
print(addresses)
```

```
## ['10 Downing St.', '12 Grimmauld Place', '1600 Pennsylvania Ave', '221 B Baker St.', '42 Wallaby Way
```

```
names = ['Ron', 'Hermione', 'Harry', 'Albus', 'Sirius']
names.sort()
print(names)
```

```
## ['Albus', 'Harry', 'Hermione', 'Ron', 'Sirius']
```

Sorting lists 2

A second way of sorting a list is to use **sorted.** sorted is different from **.sort()** in several ways:

- It comes before a list, instead of after.
- It generates a new list.

Let's return to our list of names:

```
names = ['Xander', 'Buffy', 'Angel', 'Willow', 'Giles']
```

Using sorted, we can create a new list, called sorted_names, notice the difference with .sort()

```
sorted_names = sorted(names)
print(sorted_names)
```

```
## ['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']
```

```
print(names)
```

```
## ['Xander', 'Buffy', 'Angel', 'Willow', 'Giles']
```

```
names.sort()
print(names)
```

```
## ['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']
```

Review

We learned how to:

- Get the length of a list
- Select subsets of a list (called slicing)
- Count the number of times that an element appears in a list
- Sort a list of items

Example: inventory is a list of items that are in the warehouse for Bob's Furniture.

```
inventory = ['twin bed', 'twin bed', 'headboard', 'queen bed', 'king bed', 'dresser', 'dresser', 'table

#How many items
inventory_len = len(inventory)
#First element
first = inventory[0]
#Last element
last = inventory [-1]
# Items indexed 2-5
inventory_2_6 = inventory [2:6]
#First 3 items
first_3 = inventory[:3]
# How many 'twin bed's are?
twin_beds = inventory.count('twin bed')
# Sort
inventory.sort()
```

Loops

Introduction

Suppose we want to print() **each** item from a list of dog_breeds. (Notice that we don't want to print the string, but each element. We would need to use the following code snippet:

```
dog_breeds = ['french_bulldog', 'dalmatian', 'shihtzu', 'poodle', 'collie']

print(dog_breeds[0])
```

```
## french_bulldog
```

```
print(dog_breeds[1])
```

```
## dalmatian
```

```
print(dog_breeds[2])
```

```
## shihtzu
```

```
print(dog_breeds[3])
```

```
## poodle
```

```
print(dog_breeds[4])
```

```
## collie
```

This seems inefficient. Luckily, Python (and most other programming languages) gives us an easier way of using, or iterating through, every item in a list. We can use **loops!** A loop is a way of repeating a set of code many times.

```
for breed in dog_breeds:
    print(breed)
```

```
## french_bulldog
## dalmatian
## shihtzu
## poodle
## collie
```

In this section, we'll be learning about:

- Loops that let us move through each item in a list, called **for loops**.
- Loops that keep going until we tell them to stop, called **while loops**.
- Loops that create new lists, called **list comprehensions**.

Create a For Loop

In the previous exercise, we saw that we can print each item in a list using a for loop. A for loop lets us perform an action on each item in a list. Using each element of a list is known as **iterating**.

The general way of writing a for loop is:

```
#    for <temporary variable> in <list variable>:
#        <action>
```

Example:

```
for breed in dog_breeds:
    print(breed)
```

```
## french_bulldog
## dalmatian
## shihtzu
## poodle
## collie
```

In our dog breeds example, *breed* was the temporary variable, *dog_breeds* was the list variable, and *print(breed)* was the action performed on every item in the list.

Our temporary variable can be named whatever we want and does not need to be defined beforehand. Each of the following code snippets does the exact same thing as our example:

```
for i in dog_breeds:
    print(i)
```

```
## french_bulldog
## dalmatian
## shihtzu
## poodle
## collie
```

```
for dog in dog_breeds:
    print(dog)
```

```
## french_bulldog
## dalmatian
## shihtzu
## poodle
## collie
```

Notice that in all of these examples the print statement is indented. Everything in the same level of indentation after the for loop declaration is included in the for loop, and run every iteration. If we forget to indent, we'll get an *IndentationError*.

Using Range in Loops

Previously, we iterated through an existing list. Often we won't be iterating through a specific list, we'll just want to do a certain action multiple times. For example, if we wanted to print out a "WARNING!" message three times, we would want to say something like:

```
#  for i in <a list of length 3>:
#    print("WARNING!")
```

Notice that we need to iterate through a list of length 3, but we don't care what's in the list.

To create these lists of length n, we can use the range function. range takes in a number n as input, and returns a list from 0 to n-1. For example:

```
zero_thru_five = range(6)
# zero_thru_five is now [0, 1, 2, 3, 4, 5]
zero_thru_one = range(2)
# zero_thru_one is now [0, 1]
```

So, an easy way to accomplish our "WARNING!" example would be:

```
for i in range(3):
  print("WARNING!")
```

```
## WARNING!
## WARNING!
## WARNING!
```

Remember our dog_breeds list? It had 5 elements, so we could use it to print 5 "WARNING!s" (not really usefull but is for you to understand how the for loop works)

```
for i in dog_breeds:
  print("WARNING!")
```

```
## WARNING!
## WARNING!
## WARNING!
## WARNING!
## WARNING!
```

Infinite Loops

We've iterated through lists that have a discrete beginning and end. However, let's consider this example:

```
#### DON'T RUN ####
#  my_favorite_numbers = [4, 8, 15, 16, 42]
#
#  for number in my_favorite_numbers:
#    my_favorite_numbers.append(1)
```

What happens here? Every time we enter the loop, we add a 1 to the end of the list that we are iterating through. As a result, we never make it to the end of the list! It keeps growing! A loop that never terminates is called an infinite loop. **These are very dangerous for your code!**

A program that hits an infinite loop often becomes completely unusable. The best course of action is to never write an infinite loop.

Note: If you accidentally stumble into an infinite loop while developing on your own machine, you can end the loop by using **control + c** (cmd + c in Mac) to terminate the program. If you're writing code in our online editor, you'll need to refresh the page to get out of an infinite loop!

Example of using for loops:

Suppose we have two lists of students, students_period_A and students_period_B. We want to combine all students into students_period_B.

```
students_period_A = ["Alex", "Briana", "Cheri", "Daniele"]
students_period_B = ["Dora", "Minerva", "Alexa", "Obie"]
```

We want to combine all students at the end of students_period_B.

```
for student in students_period_A:
  students_period_B.append(student)

print(students_period_B)
```

```
## ['Dora', 'Minerva', 'Alexa', 'Obie', 'Alex', 'Briana', 'Cheri', 'Daniele']
```

Notice that every time you run the for loop, it will append again *students_period_A* at the end of *students_period_B*.

Break

We often want to use a for loop to search through a list for some value:

```
items_on_sale = ["blue_shirt", "striped_socks", "knit_dress", "red_headband", "dinosaur_onesie"]
# we want to check if the item with ID "knit_dress" is on sale:
for item in items_on_sale:
  if item == "knit_dress":
    print("Yes, there's a Knit Dress on sale!")
```

```
## Yes, there's a Knit Dress on sale!
```

This code goes through each item in items_on_sale and checks for a match. After we find that "knit_dress" is in the list items_on_sale, we don't need to go through the rest of the items_on_sale list. Since it's only 5 elements long, iterating through the entire list is not a big deal in this case. But what if items_on_sale had 1000 items after "knit_dress"? What if it had 100,000 items after "knit_dress"?

You can stop a for loop from inside the loop by using break. When the program hits a break statement, control returns to the code outside of the for loop. For example:

```
items_on_sale = ["blue_shirt", "striped_socks", "knit_dress", "red_headband", "dinosaur_onesie"]

print("Checking the sale list!")
```

```
## Checking the sale list!
```

```
for item in items_on_sale:
  print(item)
  if item == "knit_dress":
    break
```

```
## blue_shirt
## striped_socks
## knit_dress
```

```
print("End of search!")
```

```
## End of search!
```

We didn't need to check "red_headband" or "dinosaur_onesie" at all!

Another example:

You have a list of dog breeds you can adopt, dog_breeds_available_for_adoption. We check if the dog_breed_I_want is available. If so, we "They have the dog I want!" and stop the loop.

```
dog_breeds_available_for_adoption = ['french_bulldog', 'dalmatian', 'shihtzu', 'poodle', 'collie']
dog_breed_I_want = 'dalmatian'

for dog in dog_breeds_available_for_adoption:
  if dog == dog_breed_I_want:
    print("They have the dog I want!")
    break
```

```
## They have the dog I want!
```

Continue

When we're iterating through lists, we may want to skip some values. Let's say we want to print out all of the numbers in a list, unless they're negative. We can use **continue** to move to the next **i** in the list:

```
big_number_list = [1, 2, -1, 4, -5, 5, 2, -9]
for i in big_number_list:
  if i < 0:
    continue
  print(i)
```

```
## 1
## 2
## 4
## 5
## 2
```

Every time there was a negative number, the **continue** keyword moved the index to the next value in the list, without executing the code in the rest of the for loop.

While Loops

We now have seen and used a lot of examples of for loops. There is another type of loop we can also use, called a **while loop**. The **while loop** performs a set of code until some condition is reached.

While loops can be used to iterate through lists, just like **for loops**:

```
dog_breeds = ['bulldog', 'dalmation', 'shihtzu', 'poodle', 'collie']
index = 0
while index < len(dog_breeds):
  print(dog_breeds[index])
  index += 1
```

```
## bulldog
## dalmation
## shihtzu
## poodle
## collie
```

Every time the condition of the while loop (in this case, index < len(dog_breeds)) is satisfied, the code inside the while loop runs. While loops can be useful when you don't know how many iterations it will take to satisfy a condition.

Here's another example:

We are adding students to a Poetry class, the size of which is capped at 6. While the length of the students_in_poetry list is less than 6, we .pop() to take a student off the all_students list and add it to the students_in_poetry list.

First we look at the **.pop()** method will take an item off of the end of a list:

```
my_list = [1, 4, 10, 15]
number = my_list.pop()
print(number)
```

```
## 15
```

```
print(my_list)
```

```
## [1, 4, 10]
```

Then we proceed to fill our Poetry Class

```
all_students = ["Alex", "Briana", "Cheri", "Daniele", "Dora", "Minerva", "Alexa", "Obie", "Arius", "Lok
students_in_poetry = []

while len(students_in_poetry) < 6:
  student = all_students.pop()
  students_in_poetry.append(student)

print(students_in_poetry)
```

```
## ['Loki', 'Arius', 'Obie', 'Alexa', 'Minerva', 'Dora']
```

Nested Loops

We have seen how we can go through the elements of a list. What if we have a list made up of multiple lists? How can we loop through all of the individual elements?

Suppose we are in charge of a science class, that is split into three project teams:

```
project_teams = [["Ava", "Samantha", "James"], ["Lucille", "Zed"], ["Edgar", "Gabriel"]]
```

If we want to go through each student, we have to put one loop inside another:

```
for team in project_teams:
  for student in team:
    print(student)
```

```
## Ava
## Samantha
## James
## Lucille
## Zed
## Edgar
## Gabriel
```

Example: We have the list sales_data that shows the numbers of different flavors of ice cream sold at three different locations of the fictional shop, Gilbert and Ilbert's Scoop Shop. We want to sum up the total number of scoops sold.

```
sales_data = [[12, 17, 22], [2, 10, 3], [5, 12, 13]]
# We define variable scoops_sold and set to 0
scoops_sold = 0
# We create the nested loop.
for location in sales_data:
  for ice_cream in location:
    scoops_sold = scoops_sold + ice_cream

print(scoops_sold)
```

```
## 96
```

List Comprehensions

Let's say we have scraped a certain website and gotten these words:

```
words = ["@coolguy35", "#nofilter", "@kewldawg54", "reply", "timestamp", "@matchamom", "follow", "#updo
```

We want to make a new list, called usernames, that has all of the strings in words with an '@' as the first character. We know we can do this with a for loop:

```
words = ["@coolguy35", "#nofilter", "@kewldawg54", "reply", "timestamp", "@matchamom", "follow", "#updo
usernames = []

for word in words:
  if word[0] == '@':
    usernames.append(word)
```

First, we created a new empty list, usernames, and as we looped through the words list, we added every word that matched our criterion. Now, the usernames list looks like this:

```
print(usernames)
```

```
## ['@coolguy35', '@kewldawg54', '@matchamom']
```

Python has a convenient shorthand to create lists like this with one line:

```
usernames = [word for word in words if word[0] == '@']
```

This is called a list comprehension. It will produce the same output as the for loop did:

```
print(usernames)
```

```
## ['@coolguy35', '@kewldawg54', '@matchamom']
```

This list comprehension:

- Takes an element in words
- Assigns that element to a variable called word
- Checks if word[0] == '@', and if so, it adds word to the new list, usernames. If not, nothing happens.
- Repeats steps 1-3 for all of the strings in words

Note: if we hadn't done any checking (let's say we had omitted if word[0] == '@') such as usernames = [word for word in words], the new list would be just a copy of words:

Example:

We have defined a list heights of visitors to a theme park. In order to ride the Topsy Turvy Tumbletron roller coaster, you need to be above 161 centimeters. Using a list comprehension, we create a new list called can_ride_coaster that has every element from heights that is greater than 161.

```
heights = [161, 164, 156, 144, 158, 170, 163, 163, 157]
# We create the list comprehension
can_ride_coaster = [person for person in heights if person > 161]

print(can_ride_coaster)
```

```
## [164, 170, 163, 163]
```

More List Comprehensions

Let's say we're working with the usernames list from the last exercise:

```
print(usernames)
```

```
## ['@coolguy35', '@kewldawg54', '@matchamom']
```

We want to create a new list with the string " please follow me!" added to the end of each username. We want to call this new list *messages*. We can use a list comprehension to make this list with one line:

```
messages = [user + " please follow me!" for user in usernames]
```

This list comprehension:

- Takes a string in usernames
- Assigns that string to a variable called user
- Adds " please follow me!" to user
- Appends that concatenation to the new list called messages
- Repeats steps 1-4 for all of the strings in usernames

Now, messages contains these values:

```
print(messages)
```

```
## ['@coolguy35 please follow me!', '@kewldawg54 please follow me!', '@matchamom please follow me!']
```

Being able to create lists with modified values is especially useful when working with numbers. Let's say we have this list:

```
my_upvotes = [192, 34, 22, 175, 75, 101, 97]
```

We want to add 100 to each value. We can accomplish this goal in one line:

```
updated_upvotes = [vote_value + 100 for vote_value in my_upvotes]
```

This list comprehension:

- Takes a number in my_upvotes
- Assigns that number to a variable called vote_value
- Adds 100 to vote_value
- Appends that sum to the new list updated_upvotes
- Repeats steps 1-4 for all of the numbers in my_upvotes

```
print(updated_upvotes)
```

```
## [292, 134, 122, 275, 175, 201, 197]
```

Another example:

We have a list of temperatures in celsius.

```
celsius = [0, 10, 15, 32, -5, 27, 3]
```

Using a list comprehension, we create a new list called fahrenheit that converts each element in the celsius list to fahrenheit. Remember the formula to convert:

$$fahrenheit^\circ = \frac{celsius^\circ \cdot 9}{5} + 32$$

```
celsius = [0, 10, 15, 32, -5, 27, 3]

fahrenheit = [temp* 9/5 + 32 for temp in celsius]

print(fahrenheit)
```

```
## [32.0, 50.0, 59.0, 89.6, 23.0, 80.6, 37.4]
```

Review

Now we know:

- how to write a for loop
- how to use range in a loop
- what infinite loops are and how to avoid them
- how to skip values in a loop
- how to write a while loop
- how to make lists with one line

Example:

```
single_digits = list(range(10))
print(single_digits)
```

```
## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
squares = []

for digit in single_digits:
  squares.append(digit**2)
print(squares)
```

```
## [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
cubes = [digit**3 for digit in single_digits]
print (cubes)
```

```
## [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Loop Challenges

In this section, we write usefull loop functions that may be helpful later.

Divisible by ten

The function divisible_by_ten()takes a list of numbers named nums as a parameter and return the count of how many numbers in the list are divisible by 10. Check the following two methods. Which one is faster?

```
#Method 1
def divisible_by_ten(nums):
  counter=[]
  for num in nums:
    if num % 10 == 0:
      counter.append(num)
  return(len(counter))

print(divisible_by_ten([20, 25, 30, 35, 40]))
```

```
## 3
```

```
#Method 2
def divisible_by_ten(nums):
  count = 0
  for number in nums:
    if (number % 10 == 0):
      count += 1
  return count

print(divisible_by_ten([20, 25, 30, 35, 40]))
```

```
## 3
```

Greetings

We create a function named add_greetings() which takes a list of strings named names as a parameter and return a list with greetings

```
#Method 1
def add_greetings(names):
  greetings=[]
  for name in names:
    greetings.append("Hello, "+ name)
  return greetings

print(add_greetings(["Owen", "Max", "Sophie"]))
```

```
## ['Hello, Owen', 'Hello, Max', 'Hello, Sophie']
```

Delete Starting Even Numbers

The delete_starting_evens() function should remove elements from the front of lst until the front of the list is not even. The function should then return lst. For example if lst started as [4, 8, 10, 11, 12, 15], then delete_starting_evens(lst) should return [11, 12, 15]. We make sure the function works even if every element in the list is even!

```
def delete_starting_evens(lst):
  while (len(lst) > 0 and lst[0] % 2 == 0):
    lst = lst[1:]
  return lst

print(delete_starting_evens([4, 8, 10, 11, 12, 15]))
```

```
## [11, 12, 15]
```

```
print(delete_starting_evens([4, 8, 10]))
```

```
## []
```

Odd Indices

The function should create a new empty list and add every element from lst that has an odd index. The function should then return this new list. For example, odd_indices([4, 3, 7, 10, 11, -2]) should return the list [3, 10, -2].

```
def odd_indices(lst):
  odd=[]
  for index in range(1, len(lst), 2):
    odd.append(lst[index])
  return odd

print(odd_indices([4, 3, 7, 10, 11, -2]))
```

```
## [3, 10, -2]
```

Exponents

A function named exponents() that takes two lists as parameters named bases and powers. Return a new list containing every number in bases raised to every number in powers.

```
def exponents(bases,powers):
  raised=[]
  for base in bases:
    for power in powers:
      raised.append(base**power)
  return raised

print(exponents([2, 3, 4], [1, 2, 3]))
```

```
## [2, 4, 8, 3, 9, 27, 4, 16, 64]
```

Larger Sum

A function named larger_sum() that takes two lists of numbers as parameters named lst1 and lst2. The function should return the list whose elements sum to the greater number. If the sum of the elements of each list are equal, return lst1.

```
def larger_sum(lst1, lst2):
  sum1=0
  sum2=0
  for num in lst1:
    sum1+=num
  for num in lst2:
    sum2+=num
  if sum1 >= sum2:
    return lst1
  else:
    return lst2

print(larger_sum([1, 9, 5], [2, 3, 11]))
```

```
## [2, 3, 11]
```

Over 9000

A function named over_nine_thousand() that takes a list of numbers named lst as a parameter.

The function should sum the elements of the list until the sum is greater than 9000. When this happens, the function should return the sum. If the sum of all of the elements is never greater than 9000, the function should return total sum of all the elements. If the list is empty, the function should return 0.

For example, if lst was [8000, 900, 120, 5000], then the function should return 9020.

```
def over_nine_thousand(lst):
  sum=0
  for num in lst:
    sum += num
    if sum > 9000:
      break
  return sum

print(over_nine_thousand([8000, 900, 120, 5000]))
```

```
## 9020
```

Max Num

A function named max_num() that takes a list of numbers named nums as a parameter. The function should return the largest number in nums

```
def max_num(nums):
  max=nums[0]
  for num in nums:
    if num >= max:
      max = num
  return max

print(max_num([50, -10, 0, 75, 20]))
```

```
## 75
```

Same values

A function named same_values() that takes two lists of numbers of equal size as parameters. The function should return a list of the indices where the values were equal in lst1 and lst2.

```
def same_values(lst1,lst2):
  equal=[]
  for index in range(len(lst1)):
    if lst1[index]==lst2[index]:
      equal.append(index)
  return equal


print(same_values([5, 1, -10, 3, 3], [5, 10, -10, 3, 5]))
```

```
## [0, 2, 3]
```

Reversed list

A function named reversed_list() that takes two lists of the same size as parameters named lst1 and lst2. The function should return True if lst1 is the same as lst2 reversed. The function should return False otherwise. For example, reversed_list([1, 2, 3], [3, 2, 1]) should return True.

```
#  You want to compare lst1[0] with lst2[4], lst1[1] with lst2[3] and so on.
def reversed_list(lst1, lst2):
  for index in range(len(lst1)):
    if lst1[index] != lst2[len(lst2) - 1 - index]:
      return False
  return True

print(reversed_list([1, 2, 3], [3, 2, 1]))
```

```
## True
```

```
print(reversed_list([1, 5, 3], [3, 2, 1]))
```

```
## False
```

Strings

Introduction

Words and sentences are fundamental to how we communicate, so it follows that we'd want our computers to be able to work with words and sentences as well. In Python, the way we store something like a word, a sentence, or even a whole paragraph is as a string. A string is a sequence of characters. It can be any length and can contain any letters, numbers, symbols, and spaces.

In this section, you will learn more about strings and how they are treated in Python. You will learn how to slice strings, select specific characters from strings, search strings for characters, iterate through strings, and use strings in conditional statements.

Strings as lists

A string can be thought of as a list of characters. Like any other list, each character in a string has an index. Consider the string

```
favorite_fruit = "blueberry"
```

We can select specific letters from this string using the index. Let's look at the first letter of the string.

```
favorite_fruit[0]
```

```
## 'b'
```

It's important to note that indices of strings must be integers. If you were to try to select a non-integer index we would get a TypeError.

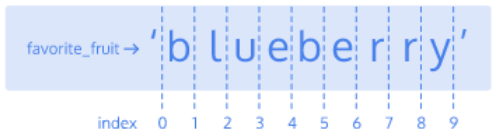
Cut Me a Slice of String

Not only can we select a single character from a string, we can select entire chunks of characters from a string. We can do this with the following syntax:

```
#  string_name[first_index:last_index]
```

This is called slicing a string. When we slice a string we are creating a new string that starts at (and includes) the first_index and ends at (but excludes) the last_index. Let's look at some examples of this. Recall our favorite fruit:

The indices of this string are shown in the diagram below.



Let's say we wanted a new string that contains the letters "eberr". We could slice favorite_fruit as follows:

```
favorite_fruit[3:8]
```

```
## 'eberr'
```

Notice how the character at the first index, e, is INCLUDED, but the character at the last index, y, is EXCLUDED. If you look for the indices 3 and 8 in the diagram, you can see how the y is outside that range.

We can also have open-ended selections. If we remove the first index, the slice starts at the beginning of the string and if we remove the second index the slice continues to the end of the string.

```
favorite_fruit[:4]
```

```
## 'blue'
```

```
favorite_fruit[4:]
```

```
## 'berry'
```

Again, notice how the b from berry is excluded from the first example and included in the second example.

Concatenating Strings

You can also concatenate two existing strings together into a new string. Consider the following two strings.

```
fruit_prefix = "blue"
fruit_suffix = "berries"
```

We can create a new string by concatenating them together as follows:

```
favorite_fruit = fruit_prefix + fruit_suffix
print(favorite_fruit)
```

```
## blueberries
```

Notice that there are no spaces added here. You have to manually add in the spaces when concatenating strings if you want to include them.

```
fruit_sentence = "My favorite fruit is " + favorite_fruit
print(fruit_sentence)
```

```
## My favorite fruit is blueberries
```

Example:

Copeland's Corporate Company has realized that their policy of using the first five letters of an employee's last name as a user name isn't ideal when they have multiple employees with the same last name. A function called account_generator that takes two inputs, first_name and last_name and concatenates the first three letters of each and then returns the new account name.

```
def account_generator(first_name,last_name):
  account=first_name[0:3]+last_name[0:3]
  return account

first_name = "Julie"
last_name = "Blevins"
new_account = account_generator(first_name,last_name)

print(new_account)
```

```
## JulBle
```

How Long is that String?

Python comes with some built-in functions for working with strings. One of the most commonly used of these functions is len(). len() returns the number of characters in a string

```
favorite_fruit = "blueberry"
len(favorite_fruit)
```

```
## 9
```

If you are taking the length of a sentence the spaces are counted as well.

```
fruit_sentence = "I love blueberries"
len(fruit_sentence)
```

```
## 18
```

len() comes in handy when we are trying to select the last character in a string. You can try to run the following code:

```
length = len(favorite_fruit)
# favorite_fruit[length]
```

But this code would generate an IndexError because, remember, the indices start at 0, so the final character in a string has the index of len(string_name) - 1.

```
favorite_fruit[length-1]
```

```
## 'y'
```

You could also slice the last several characters of a string using len():

```
favorite_fruit[length-4:]
```

```
## 'erry'
```

Using a len() statement as the starting index and omitting the final index lets you slice n characters from the end of a string where n is the amount you subtract from len().

Example:

Copeland's Corporate Company also wants to update how they generate temporary passwords for new employees.

Write a function called password_generator that takes two inputs, first_name and last_name and then concatenate the last three letters of each and returns them as a string.

```
first_name = "Reiko"
last_name = "Matsuki"

def password_generator(first_name,last_name):
  pw = first_name[len(first_name)-3:]+last_name[len(last_name)-3:]
  return pw

temp_password=password_generator(first_name,last_name)

print(temp_password)
```

```
## ikouki
```

Negative Indexes

In the previous exercise, we used len() to get a slice of characters at the end of a string. There's a much easier way to do this, we can use negative indices! Negative indices count backward from the end of the string, so string_name[-1] is the last character of the string, string_name[-2] is the second last character of the string, etc.

```
favorite_fruit = 'blueberry'
favorite_fruit[-1]
```

```
## 'y'
```

Notice that we are able to slice the last three characters of 'blueberry' by having a starting index of -3 and omitting a final index.

```
favorite_fruit[-3:]
```

```
## 'rry'
```

Strings are Immutable

So far in this lesson, we've been selecting characters from strings, slicing strings, and concatenating strings. Each time we perform one of these operations we are creating an entirely new string. This is because strings are immutable. This means that we cannot change a string once it is created. We can use it to create other strings, but we cannot change the string itself.

This property, generally, is known as mutability. Data types that are mutable can be changed, and data types, like strings, that are immutable cannot be changed.

If we try:

```
first_name = "Bob"
last_name = "Daily"
#  first_name[0] = "R"
# We get TypeError: 'str' object does not support item assignment
```

We must:

```
fixed_first_name = "R" + first_name[-2:]
print(fixed_first_name)
```

```
## Rob
```

Escape Characters

Occasionally when working with strings, you'll find that you want to include characters that already have a special meaning in python. For example let's say I create the string:

```
#  favorite_fruit_conversation = "He said, "blueberries are my favorite!""
```

We'll have accidentally ended the string before we wanted to by including the " character. The way we can do this is by introducing escape characters. By adding a backslash in front of the special character we want to escape, ", we can include it in a string.

```
favorite_fruit_conversation = "He said, \"blueberries are my favorite!\""
```

Iterating through Strings

Now you know enough about strings that we can start doing the really fun stuff!

Because strings are lists, that means we can iterate through a string using for or while loops. This opens up a whole range of possibilities of ways we can manipulate and analyze strings. Let's take a look at an example.

```
def print_each_letter(word):
  for letter in word:
    print(letter)
```

This function will iterate through each letter in a given word and will print it to the terminal.

```
print_each_letter('Diego')
```

```
## D
## i
## e
## g
## o
```

Example:

Let's **create** the len() function:

```
def get_length(string):
  len = 0
  for letter in string:
    len+=1
  return len

print(get_length('test'))
```

```
## 4
```

Strings and Conditionals I

Now that we are iterating through strings, we can really explore the potential of strings. When we iterate through a string we do something with each character. By including conditional statements inside of these iterations, we can start to do some really cool stuff.

Take a look at the following code:

```python
favorite_fruit = "blueberry"
counter = 0
for character in favorite_fruit:
  if character == "b":
    counter = counter + 1
print(counter)
```

```
## 2
```

This code will count the number of bs in the string "blueberry" (hint: it's two). Let's take a moment and break down what exactly this code is doing.

First, we define our string, favorite_fruit, and a variable called counter, which we set equal to zero. Then the for loop will iterate through each character in favorite_fruit and compare it to the letter b.

Each time a character equals b the code will increase the variable counter by one. Then, once all characters have been checked, the code will print the counter, telling us how many bs were in "blueberry". This is a great example of how iterating through a string can be used to solve a specific application, in this case counting a certain letter in a word.

Example:

A function called letter_check that takes two inputs, word and letter. This function should return True if the word contains the letter and False if it does not.

```python
#Method 1
def letter_check(word,letter):
  check=False
  for character in word:
    if character == letter:
      check=True
  return check

print(letter_check("strawberry", "x"))
```

```
## False
```

```python
print(letter_check("strawberry", "w"))
```

```
## True
```

```python
#Method 2
def letter_check(word, letter):
  for character in word:
    if character == letter:
      return True
  return False

print(letter_check("strawberry", "x"))
```

```
## False
```

```python
print(letter_check("strawberry", "w"))
```

```
## True
```

Strings and Conditionals II

There's an even easier way than iterating through the entire string to determine if a character is in a string. We can do this type of check more efficiently using in. in checks if one string is part of another string. Here is what the syntax of in looks like:

```
#  letter in word
```

Here, letter in word is a boolean expression that is True if the string letter is in the string word. Here are some examples:

Examples:

```
"e" in "blueberry"
```

```
## True
```

```
"a" in "blueberry"
```

```
## False
```

In fact, this method is more powerful than the function you wrote in the last exercise because it works not only with letters, but with entire strings as well.

```
"blue" in "blueberry"
```

```
## True
```

Example:

A function called contains that takes two arguments, big_string and little_string and returns True if big_string contains little_string.

```
def contains(big_string,little_string):
  if little_string in big_string:
    return True
  else:
    return False

print(contains("watermelon", "melon"))
```

```
## True
```

```
print(contains("watermelon", "berry"))
```

```
## False
```

Another example:

A function called common_letters that takes two arguments, string_one and string_two and then returns a list with all of the letters they have in common. The letters in the returned list should be unique.

```
def common_letters(string_one, string_two):
  common = []
  for letter in string_one:
    if (letter in string_two) and not (letter in common):
      common.append(letter)
  return common

print(common_letters('manhattan', 'san francisco'))
```

```
## ['a', 'n']
```

Review

- A string is a list of characters.
- A character can be selected from a string using its index string_name[index]. These indices start at 0.
- A 'slice' can be selected from a string. These can be between two indices or can be open-ended, selecting all of the string from a point.
- Strings can be concatenated to make larger strings.
- len() can be used to determine the number of characters in a string.
- Strings can be iterated through using for loops.
- Iterating through strings opens up a huge potential for applications, especially when combined with conditional statements.

Example:

Copeland's Corporate Company has finalized what they want to their username and temporary password creation to be and have enlisted your help, once again, to build the function to generate them. In this exercise, you will create two functions, username_generator and password_generator.

Let's start with username_generator. Create a function called username_generator take two inputs, first_name and last_name and returns a username. The username should be a slice of the first three letters of their first name and the first four letters of their last name. If their first name is less than three letters or their last name is less than four letters it should use their entire names.

For example, if the employee's name is Abe Simpson the function should generate the username AbeSimp.

```python
# Method 1
def username_generator(first_name,last_name):
  if len(first_name) >= 3 and len(last_name) >= 4:
    username=first_name[0:3]+last_name[0:4]
  elif not len(first_name) >= 3 and len(last_name) >= 4:
    username=first_name+last_name[0:4]
  else:
    username=first_name[0:3]+last_name
  return username

print(username_generator("Abe","Simpson"))
```

```
## AbeSimp
```

```python
# Method 2
def username_generator(first_name, last_name):
    if len(first_name) < 3:
        user_name = first_name
    else:
        user_name = first_name[0:3]
    if len(last_name) < 4:
        user_name += last_name
    else:
        user_name += last_name[0:4]
    return user_name

print(username_generator("Abe","Simpson"))
```

```
## AbeSimp
```

Now for the temporary password, they want the function to take the input user name and shift all of the letters by one to the right, so the last letter of the username ends up as the first letter and so forth. For example, if the username is AbeSimp, then the temporary password generated should be pAbeSim.

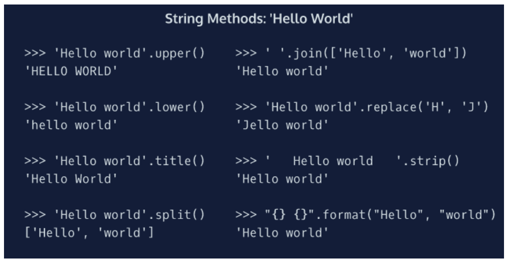
```
def password_generator(user_name):
    password = ""
    for i in range(len(user_name)):
        password += user_name[i-1]
    return password
print(password_generator("AbeSimp"))
```

```
## pAbeSim
```

String Methods

Introduction

Do you have a gigantic string that you need to parse for information? Do you need to sanitize a users input to work in a function? Do you need to be able to generate outputs with variable values? All of these things can be accomplished with string methods!


String Methods: 'Hello World'

```
>>> 'Hello world'.upper()    >>> ' '.join(['Hello', 'world'])
'HELLO WORLD'                'Hello world'

>>> 'Hello world'.lower()    >>> 'Hello world'.replace('H', 'J')
'hello world'                'Jello world'

>>> 'Hello world'.title()    >>> '   Hello world   '.strip()
'Hello World'                'Hello world'

>>> 'Hello world'.split()    >>> "{} {}".format("Hello", "world")
['Hello', 'world']           'Hello world'
```

Python comes with built-in string methods that gives you the power to perform complicated tasks on strings very quickly and efficiently. These string methods allow you to change the case of a string, split a string into many smaller strings, join many small strings together into a larger string, and allow you to neatly combine changing variables with string outputs.

In the previous lesson, you worked len(), which was a function that determined the number of characters in a string. This, while similar, was NOT a string method. String methods all have the same syntax:

```
#  string_name.string_method(arguments)
```

Unlike len(), which is called with a string as it's argument, a string method is called at the end of a string and each one has its own method specific arguments.

Formatting Methods

There are three string methods that can change the casing of a string. These are .lower(), .upper(), and .title().

- .lower() returns the string with all lowercase characters.
- .upper() returns the string with all uppercase characters.
- .title() returns the string in title case, which means the first letter of each word is capitalized.

Here's an example of .lower() in action:

```
favorite_song = 'SmOoTH'
favorite_song_lowercase = favorite_song.lower()
favorite_song_lowercase
```

```
## 'smooth'
```

Every character was changed to lowercase! It's important to remember that string methods can only create new strings, they do not change the original string. These string methods are great for sanitizing user input and standardizing the formatting of your strings.

Splitting Strings I

.upper(), .lower(), and .title() all are performed on an existing string and produce a string in return. Let's take a look at a string method that returns a different object entirely!

.split() is performed on a string, takes one argument, and returns a list of substrings found between the given argument (which in the case of .split() is known as the delimiter). The following syntax should be used:

```
#  string_name.split(delimiter)
```

If you do not provide an argument for .split() it will default to splitting at spaces. Note: if we run .split() on a string with no spaces, we will get the same string in return.

In the code below is a string of the first line of the poem Spring Storm by William Carlos Williams. We use .split() to create a list called line_one_words that contains each word in this line of poetry.

```
line_one = "The sky has given over"
line_one_words = line_one.split(' ')
print(line_one_words)
```

```
## ['The', 'sky', 'has', 'given', 'over']
```

Splitting Strings II

If we provide an argument for .split() we can dictate the character we want our string to be split on. This argument should be provided as a string itself.

Consider the following example:

```
greatest_guitarist = "santana"
greatest_guitarist.split('an')
```

```
## ['s', 't', 'a']
```

We provided 'n' as the argument for .split() so our string "santana" got split at each 'n' character into a list of three strings.

What do you think happens if we split the same string at 'a'?

```
greatest_guitarist.split('a')
```

```
## ['s', 'nt', 'n', '']
```

Notice that there is an unexpected extra ” string in this list. When you split a string on a character that it also ends with, you'll end up with an empty string at the end of the list.

You can use any string as the argument for .split(), making it a versatile and powerful tool.

Example:

Your boss at the Poetry organization sent over a bunch of author names that he wants you to prepare for importing into the database. Annoyingly, he sent them over as a long string with the names separated by commas. Create another list called author_last_names that only contains the last names of the poets in the provided string.

```
authors = "Audre Lorde,Gabriela Mistral,Jean Toomer,An Qi,Walt Whitman,Shel Silverstein,Carmen Boullosa

author_names = authors.split(',')
```

```python
author_last_names = []
for name in author_names:
  author_last_names.append(name.split()[-1])
#[-1] tell the programm to take the last item of the splitted full name (the last name)
print(author_last_names)
```

```
## ['Lorde', 'Mistral', 'Toomer', 'Qi', 'Whitman', 'Silverstein', 'Boullosa', 'Suraiyya', 'Hughes', 'Ri
```

Splitting Strings III

We can also split strings using **escape sequences.** Escape sequences are used to indicate that we want to split by something in a string that is not necessarily a character. The two escape sequences we will cover here are:

- \n Newline
- \t Horizontal Tab

Newline or \n will allow us to split a multi-line string by line breaks and \t will allow us to split a string by tabs. \t is particularly useful when dealing with certain datasets because it is not uncommon for data points to be separated by tabs.

Let's take a look at an example of splitting by an escape sequence:

```python
smooth_chorus = """And if you said, "This life ain't good enough."
I would give my world to lift you up
I could change my life to better suit your mood
Because you're so smooth"""

chorus_lines = smooth_chorus.split('\n')
print(chorus_lines)
```

```
## ['And if you said, "This life ain\'t good enough."', 'I would give my world to lift you up', 'I coul
```

This code is splitting the multi-line string at the newlines (\n) which exist at the end of each line and saving it to a new list called chorus_lines. The new list contains each line of the original string as it's own smaller string. Also, notice that Python automatically escaped the \' character when it created the new list.

Joining Strings I

Now that you've learned to break strings apart using .split(), let's learn to put them back together using **.join()**. .join() is essentially the opposite of .split(), it joins a list of strings together with a given delimiter. The syntax of .join() is:

```python
#  'delimiter'.join(list_you_want_to_join)
```

Now this may seem a little weird, because with .split() the argument was the delimiter, but now the argument is the list. This is because join is still a string method, which means it has to act on a string. The string .join() acts on is the delimiter you want to join with, therefore the list you want to join has to be the argument.

This can be a bit confusing, so let's take a look at an example:

```python
my_poem = ['My', 'Spanish', 'Harlem', 'Mona', 'Lisa']
' '.join(my_poem)
```

```
## 'My Spanish Harlem Mona Lisa'
```

We take the list of strings, my_poem, and we joined it together with our delimiter, ' ', which is a space. The space is important if you are trying to build a sentence from words, otherwise, we would have ended up with:

```
''.join(my_poem)
```

```
## 'MySpanishHarlemMonaLisa'
```

Example:

We have a list of words from the first line of Jean Toomer's poem Reapers. We use .join() to combine these words into a sentence and save that sentence as the string reapers_line_one.

```
reapers_line_one_words = ["Black", "reapers", "with", "the", "sound", "of", "steel", "on", "stones"]
' '.join(reapers_line_one_words)
```

```
## 'Black reapers with the sound of steel on stones'
```

Joining Strings II

In the last exercise, we joined together a list of words using a space as the delimiter to create a sentence. In fact, you can use any string as a delimiter to join together a list of strings. For example, if we have the list

```
santana_songs = ['Oye Como Va', 'Smooth', 'Black Magic Woman', 'Samba Pa Ti', 'Maria Maria']
```

We could join this list together with ANY string. One often used string is a comma , because then we can create a *string of comma separated variables, or CSV.*

```
santana_songs_csv = ','.join(santana_songs)
santana_songs_csv
```

```
## 'Oye Como Va,Smooth,Black Magic Woman,Samba Pa Ti,Maria Maria'
```

You'll often find data stored in CSVs because it is an efficient, simple file type used by popular programs like Excel or Google Spreadsheets.

You can also join using escape sequences as the delimiter. Consider the following example:

winter_trees_lines contains all the lines to William Carlos Williams poem, Winter Trees:

```
winter_trees_lines = ['All the complicated details', 'of the attiring and', 'the disattiring are comple

winter_trees_full ='\n'.join(winter_trees_lines)
print(winter_trees_full)
```

```
## All the complicated details
## of the attiring and
## the disattiring are completed!
## A liquid moon
## moves gently among
## the long branches.
## Thus having prepared their buds
## against a sure winter
## the wise trees
## stand sleeping in the cold.
```

.strip()

When working with strings that come from real data, you will often find that the strings aren't super clean. You'll find lots of extra whitespace, unnecessary linebreaks, and rogue tabs.

Python provides a great method for cleaning strings: **.strip()**. Stripping a string removes all whitespace characters from the beginning and end. Consider the following example:

```
featuring = "           rob thomas                 "
featuring.strip()
```

```
## 'rob thomas'
```

All the whitespace on either side of the string has been stripped, but the whitespace in the middle has been preserved. You can also use .strip() with a character argument, which will strip that character from either end of the string.

```
featuring = "!!!rob thomas       !!!!!"
featuring.strip('!')
```

```
## 'rob thomas       '
```

By including the argument '!' we are able to strip all of the ! characters from either side of the string. Notice that now that we've included an argument we are no longer stripping whitespace, we are ONLY stripping the argument.

Example:

Audre Lorde poem: *Love, Maybe.*

```
love_maybe_lines = ['Always    ',
 '     in the middle of our bloodiest battles  ', 'you lay down your arms', '           like flowering

love_maybe_lines_stripped=[]
for element in love_maybe_lines:
  love_maybe_lines_stripped.append(element.strip())
love_maybe_full='\n'.join(love_maybe_lines_stripped)

print(love_maybe_full)
```

```
## Always
## in the middle of our bloodiest battles
## you lay down your arms
## like flowering mines
##
## to conquer me home.
```

Replace

The next string method we will cover is .replace(). Replace takes two arguments and replaces all instances of the first argument in a string with the second argument. The syntax is as follows

```
# string_name.replace(character_being_replaced, new_character)
```

Great! Let's put it in context and look at an example:

```
with_spaces = "You got the kind of loving that can be so smooth"
with_underscores = with_spaces.replace(' ', '_')
with_underscores
```

```
## 'You_got_the_kind_of_loving_that_can_be_so_smooth'
```

Here we used .replace() to change every instance of a space in the string above to be an underscore instead.

Another example:

The poetry organization has sent over the bio for Jean Toomer as it currently exists on their site. Notice that there was a mistake with his last name and all instances of Toomer are lacking one "o":

```
toomer_bio = \
"""
Nathan Pinchback Tomer, who adopted the name Jean Tomer early in his literary career, was born in Washi
"""
toomer_bio.replace('Tomer','Toomer')
```

```
## '\nNathan Pinchback Toomer, who adopted the name Jean Toomer early in his literary career, was born i
```

.find()

Another interesting string method is .find(). .find() takes a string as an argument and searching the string it was run on for that string. It then returns the first index value where that string is located.

Example:

```
'smooth'.find('t')
```

```
## 4
```

We searched the string 'smooth' for the string 't' and found that it was at the fourth index spot, so .find() returned 4.

You can also search for larger strings, and .find() will return the index value of the first character of that string.

```
"smooth".find('oo')
```

```
## 2
```

Notice here that 2 is the index of the first o.

Another example: Gabriela Mistral's poem God Wills It. At what index place does the word "disown" appear?

```
god_wills_it_line_one = "The very earth will disown you"
god_wills_it_line_one.find('disown')
```

```
## 20
```

.format() I

Python also provides a handy string method for including variables in strings. This method is .format(). .format() takes variables as an argument and includes them in the string that it is run on. You include {} marks as placeholders for where those variables will be imported.

Consider the following function:

```
def favorite_song_statement(song, artist):
  return "My favorite song is {} by {}.".format(song, artist)
favorite_song_statement('DNA', 'Kendrick Lamar')
```

```
## 'My favorite song is DNA by Kendrick Lamar.'
```

The function favorite_song_statement takes two arguments, song and artist, then returns a string that includes both of the arguments and prints a sentence. Note: .format() can take as many arguments as there are {} in the string it is run on, which in this case in two.

Now you may be asking yourself, I could have written this function using string concatenation instead of .format():

```
def favorite_song_statement2(song, artist):
  statement="My favorite song is "+ song + " by " + artist +"."
  return statement
favorite_song_statement2('DNA', 'Kendrick Lamar')
```

```
## 'My favorite song is DNA by Kendrick Lamar.'
```

Why is this the first method better? The answer is legibility and reusability. It is much easier to picture the end result .format() than it is to picture the end result of string concatenation and legibility is everything. You can also reuse the same base string with different variables, allowing you to cut down on unnecessary, hard to interpret code.

Example:

```
def poem_title_card(poet,title):
  return "The poem \"{}\" is written by {}.".format(title,poet)
# Note that we need to introduce the arguments of format () in the way they need to appear.
# So it would be usefull to define the function in the same manner.
poem_title_card("Walt Whitman", "I Hear America Singing")
```

```
## 'The poem "I Hear America Singing" is written by Walt Whitman.'
```

.format() II

.format() can be made even more legible for other people reading your code by including **keywords.** Previously with .format(), you had to make sure that your variables appeared as arguments in the same order that you wanted them to appear in the string, which just added unnecessary complications when writing code.

By including keywords in the string and in the arguments, you can remove that ambiguity. Let's look at an example.

```
def favorite_song_statement(song, artist):
    return "My favorite song is {song} by {artist}.".format(song=song, artist=artist)
```

Now it is clear to anyone reading the string what it supposed to return, they don't even need to look at the arguments of .format() in order to get a clear understanding of what is supposed to happen. You can even reverse the order of artist and song in the code above and it will work the same way. This makes writing AND reading the code much easier.

Example:

```
def poem_description(publishing_date, author, title, original_work):
  poem_desc = "The poem {title} by {author} was originally published in {original_work} in {publishing_d
  return poem_desc

author = "Shel Silverstein"
title = "My Beard"
original_work = "Where the Sidewalk Ends"
publishing_date = "1974"

my_beard_description = poem_description(publishing_date, author, title, original_work)

print(my_beard_description)
```

```
## The poem My Beard by Shel Silverstein was originally published in Where the Sidewalk Ends in 1974.
```

Review

Whatever the problem you are trying to solve, if you are working with strings then string methods are likely going to be part of the solution.

- .upper(), .title(), and .lower() adjust the casing of your string.
- .split() takes a string and creates a list of substrings.
- .join() takes a list of strings and creates a string.
- .strip() cleans off whitespace, or other noise from the beginning and end of a string.
- .replace() replaces all instances of a character/string in a string with another character/string.
- .find() searches a string for a character/string and returns the index value that character/string is found at.
- .format() and f-strings allow you to interpolate a string with variables.

Example:

```
highlighted_poems = "Afterimages:Audre Lorde:1997,  The Shadow:William Carlos Williams:1915, Ecstasy:Gal

#The information for each poem is separated by commas, and within this information is the title of the 
highlighted_poems_list = highlighted_poems.split(',')

#Notice that there is inconsistent whitespace in highlighted_poems_list. Let’s clean that up.
highlighted_poems_stripped = []
for element in highlighted_poems_list:
  highlighted_poems_stripped.append(element.strip())


#We want to break up all the information for each poem into it’s own list, so we end up with a list of 
highlighted_poems_details=[]
for element in highlighted_poems_stripped:
  highlighted_poems_details.append(element.split(':'))

#Now we want to separate out all of the titles, the poets, and the publication dates into their own lis
titles=[]
poets=[]
dates=[]

for poem in highlighted_poems_details:
  titles.append(poem[0])
  poets.append(poem[1])
  dates.append(poem[2])

# We write a for loop that uses either f-strings or .format() to prints out the following string for ea

for i in range(0,len(highlighted_poems_details)):
  print ('The poem {} was published by {} in {}'.format(titles[i], poets[i],dates[i]))
```

```
## The poem Afterimages was published by Audre Lorde in 1997
## The poem The Shadow was published by William Carlos Williams in 1915
## The poem Ecstasy was published by Gabriela Mistral in 1925
## The poem Georgia Dusk was published by Jean Toomer in 1923
## The poem Parting Before Daybreak was published by An Qi in 2014
## The poem The Untold Want was published by Walt Whitman in 1871
## The poem Mr. Grumpledump's Song was published by Shel Silverstein in 2004
## The poem Angel Sound Mexico City was published by Carmen Boullosa in 2013
## The poem In Love was published by Kamala Suraiyya in 1965
## The poem Dream Variations was published by Langston Hughes in 1994
## The poem Dreamwood was published by Adrienne Rich in 1987
```

String Challenges

Count Letters

A function called unique_english_letters that takes the string word as a parameter. The function should return the total number of unique letters in the string. Uppercase and lowercase letters should be counted as different letters. A list of every uppercase and lower case letter in the English alphabet will be helpful to include in our function.

```
letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

def unique_english_letters(word):
  unique=[]
  for letter in letters:
    if (letter in word) and (letter not in unique):
      unique.append(letter)
  return len(unique)


print(unique_english_letters("mississippi"))
# should print 4
```

```
## 4
```

```
print(unique_english_letters("Apple"))
# should print 4
```

```
## 4
```

Count X

A function named count_char_x that takes a string named word and a single character named x as parameters. The function should return the number of times x appears in word:

```
def count_char_x(word,x):
  count=[]
  for letter in word:
    if letter==x:
      count.append(letter)
  return len(count)


print(count_char_x("mississippi", "s"))
# should print 4
```

```
## 4
```

```
print(count_char_x("mississippi", "m"))
# should print 1
```

```
## 1
```

Count Multi X

A function named count_multi_char_x that takes a string named word and a string named x. This function should do the same thing as the count_char_x function you just wrote - it should return the number of times x appears in word. However, this time, make sure your function works when x is multiple characters long.

```python
def count_multi_char_x(word,x):
  splitted = word.split(x)
  return (len(splitted)-1)

print(count_multi_char_x("mississippi", "iss"))
# should print 2
```

```
## 2
```

```python
print(count_multi_char_x("apple", "pp"))
# should print 1
```

```
## 1
```

Substring Between

A function named substring_between_letters that takes a string named word, a single character named start, and another character named end. This function should return the substring between the first occurrence of start and end in word. If start or end are not in word, the function should return word.

For example, substring_between_letters(“apple”, “p”, “e”) should return “pl”.

```python
def substring_between_letters(word,start,end):
  if (start in word and end in word):
    index_s=word.find(start)
    index_e=word.find(end)
    sublist=word[index_s+1:index_e]
    return(sublist)
  else:
    return(word)

print(substring_between_letters("apple", "p", "e"))
#should return pl
```

```
## pl
```

```python
print(substring_between_letters("apple", "p", "c"))
# should print "apple"
```

```
## apple
```

X Length

A function called x_length_words that takes a string named sentence and an integer named x as parameters. This function should return True if every word in sentence has a length greater than or equal to x.

```python
def x_length_words(sentence,x):
  words=sentence.split( )
  for word in words:
    if len(word)< x:
      return False
    else:
      return True
```

```
print(x_length_words("i like apples", 2))
# should print False
```

```
## False
```

```
print(x_length_words("he likes apples", 2))
# should print True
```

```
## True
```

Check Name

A function called check_for_name that takes two strings as parameters named sentence and name. The function should return True if name appears in sentence in all lowercase letters, all uppercase letters, or with any mix of uppercase and lowercase letters. The function should return False otherwise.

```
def check_for_name(sentence,name):
  if name.lower() in sentence.lower():
    return True
  else:
    return False

print(check_for_name("My name is Jamie", "Jamie"))
# should print True
```

```
## True
```

```
print(check_for_name("My name is jamie", "Jamie"))
# should print True
```

```
## True
```

```
print(check_for_name("My name is Samantha", "Jamie"))
# should print False
```

```
## False
```

Every Other Letter

A function named every_other_letter that takes a string named word as a parameter. The function should return a string containing every odd letter in word.

```
def every_other_letter(word):
  every_other = ""
  for i in range(0, len(word), 2):
    every_other += word[i]
  return every_other

print(every_other_letter("Codecademy"))
# should print Cdcdm
```

```
## Cdcdm
```

```
print(every_other_letter("Hello world!"))
# should print Hlowrd
```

```
## Hlowrd
```

```
print(every_other_letter(""))
# should print
```

Reverse

A function named reverse_string that has a string named word as a parameter. The function should return word in reverse.

```
word="Codecademy"
reverse=""
for i in range(0, len(word)):
  print(word[-i-1])
```

```
## y
## m
## e
## d
## a
## c
## e
## d
## o
## C
```

Make Spoonerism

A Spoonerism is an error in speech when the first syllables of two words are switched. For example, a Spoonerism is made when someone says “Belly Jeans” instead of “Jelly Beans”. We write a function called make_spoonerism that takes two strings as parameters named word1 and word2. Finding the first syllable of a word is a difficult task, so for our function, we’re going to switch the first letters of each word. Return the two new words as a single string separated by a space.

```
def make_spoonerism(word1,word2):
  new_word1=word2[0]+word1[1:]
  new_word2=word1[0]+word2[1:]
  spoonerism=new_word1+' '+new_word2
  return spoonerism

print(make_spoonerism("Codecademy", "Learn"))
# should print Lodecademy Cearn
```

```
## Lodecademy Cearn
```

```
print(make_spoonerism("Hello", "world!"))
# should print wello Horld!
```

```
## wello Horld!
```

```
print(make_spoonerism("a", "b"))
# should print b a
```

```
## b a
```

Add Exclamation

A function named add_exclamation that has one parameter named word. This function should add exclamation points to the end of word until word is 20 characters long. If word is already at least 20 characters long, just return word.

```
def add_exclamation(word):
  while len(word) < 20:
    word+='!'
  return word


print(add_exclamation("Codecademy"))
# should print Codecademy!!!!!!!!!!
```

```
## Codecademy!!!!!!!!!!
```

```
print(add_exclamation("Codecademy is the best place to learn"))
# should print Codecademy is the best place to learn
```

```
## Codecademy is the best place to learn
```

Modules in Python

Introduction

In the world of programming, we care a lot about making code reusable. In most cases, we write code so that it can be reusable for ourselves. But sometimes we share code that's helpful across a broad range of situations.

In this lesson, we'll explore how to use tools other people have built in Python that are not included automatically for you when you install Python. Python allows us to package code into files or sets of files called modules.

A module is a collection of Python declarations intended broadly to be used as a tool. Modules are also often referred to as "libraries" or "packages" — **a package or librarie is really a directory that holds a collection of modules.**

Usually, to use a module in a file, the basic syntax you need at the top of that file is:

```
#   from module_name import object_name
```

Often, a library will include a lot of code that you don't need that may slow down your program or conflict with existing code. Because of this, it makes sense to only import what you need.

One common library that comes as part of the Python Standard Library is **datetime**. datetime helps you work with dates and times in Python.

Let's get started by importing and using the datetime module. In this case, you'll notice that datetime is both the name of the library and the name of the object that you are importing.

```
from datetime import datetime

current_time = datetime.now()
print(current_time)
```

```
## 2020-06-24 01:35:56.877449
```

Modules: Random

datetime is just the beginning. There are hundreds of Python modules that you can use. Another one of the most commonly used is random which allows you to generate numbers or select items at random. With random, we'll be using more than one piece of the module's functionality, so the import syntax will look like:

We'll work with two common random functions:

- random.choice() which takes a list as an argument and returns a number from the list
- random.randint() which takes two numbers as arguments and generates a random number between the two numbers you passed in

Let's take randomness to a whole new level by picking a random number from a list of randomly generated numbers between 1 and 100.

```
# Import random below:
import random
#Turn the list into a list comprehension that uses random.randint() to generate
# a random integer between 1 and 100 (inclusive) for each number in range(101).
random_list = [random.randint(1,100) for i in range(101)]
# Create randomer_number below:
randomer_number=random.choice(random_list)
# Print randomer_number below:
print(randomer_number)
```

```
## 61
```

Namespaces

Notice that when we want to invoke the randint() function we call random.randint(). This is default behavior where Python offers a namespace for the module. A namespace isolates the functions, classes, and variables defined in the module from the code in the file doing the importing. Your **local namespace**, meanwhile, is where your code is run.

Python defaults to naming the namespace after the module being imported, but sometimes this name could be ambiguous or lengthy. Sometimes, the module's name could also conflict with an object you have defined within your local namespace.

Fortunately, this name can be altered by aliasing using the as keyword:

```
#   import module_name as name_you_pick_for_the_module
```

Aliasing is most often done if the name of the library is long and typing the full name every time you want to use one of its functions is laborious.

You might also occasionally encounter **import ***. The ***** is known as a "wildcard" and matches anything and everything. This syntax is considered dangerous because it could pollute our local namespace. Pollution occurs when the same name could apply to two possible things. For example, if you happen to have a function **floor()** focused on floor tiles, using from **math import*** would also import a function **floor()** that rounds down floats.

Let's combine your knowledge of the random library with another fun library called **matplotlib**, which allows you to plot your Python code in 2D.

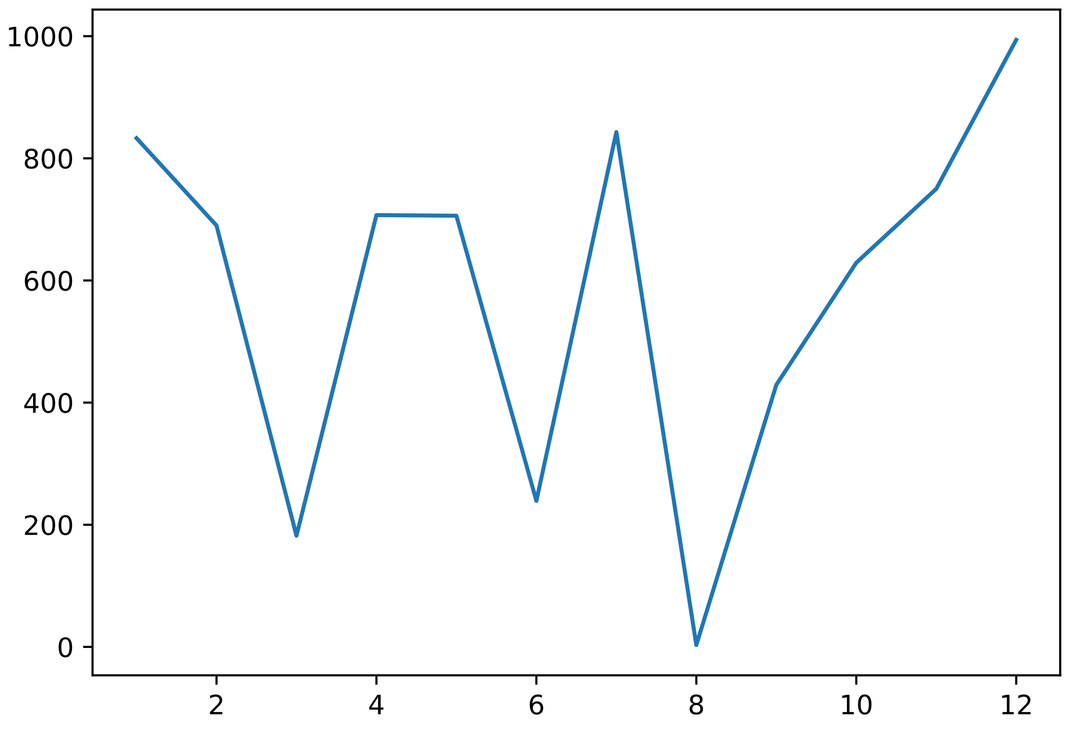
You'll use a new random function **random.sample()** that takes a range and a number as its arguments. It will return the specified number of random numbers from that range.

```
import matplotlib.pyplot as plt

#Range of numbers 1 through 12 (inclusive)
numbers_a = range(1,13)
#Random sample of twelve numbers within range(1000).
numbers_b = [random.randint(1,1000) for i in range(12)]

# Plot the two variables against each other
```

```
plt.plot(numbers_a,numbers_b)
plt.show()
```



Decimals

Let's say you are writing software that handles monetary transactions. If you used Python's built-in floating-point arithmetic to calculate a sum, it would result in a weirdly formatted number. Example:

```
cost_of_gum = 0.10
cost_of_gumdrop = 0.35

cost_of_transaction = cost_of_gum + cost_of_gumdrop
print(cost_of_transaction)
```

```
## 0.44999999999999996
```

Being familiar with rounding errors in floating-point arithmetic you want to use a data type that performs decimal arithmetic more accurately. You could do the following:

```
from decimal import Decimal

cost_of_gum = Decimal('0.10')
cost_of_gumdrop = Decimal('0.35')

cost_of_transaction = cost_of_gum + cost_of_gumdrop
print(cost_of_transaction)
```

```
## 0.45
```

Above, we use the decimal module's Decimal data type to add 0.10 with 0.35. Since we used the Decimal type the arithmetic acts much more as expected.

Usually, modules will provide functions or data types that we can then use to solve a general problem, allowing us more time to focus on the software that we are building to solve a more specific problem.

Files and Scope

You may remember the concept of scope from when you were learning about functions in Python. If a variable is defined inside of a function, it will not be accessible outside of the function.

Scope also applies to **classes** and to the **files** you are working within. Files have scope? You may be wondering.

Yes. Even files inside the same directory do not have access to each other's variables, functions, classes, or any other code. So if I have a file **sandwiches.py** and another **file hungry_people.py**, how do I give my hungry people access to all the sandwiches I defined?

Well, **files are actually modules**, so you can give a file access to another file's content using that glorious **import** statement.

With a single line of **from sandwiches import sandwiches** at the top of **hungry_people.py**, the hungry people will have all the sandwiches they could ever want.

Pandas Library

Pandas stands for "Python Data Analysis Library". According to the Wikipedia page on Pandas, "the name is derived from the term "panel data", an econometrics term for multidimensional structured data sets."

Pandas takes data (like a CSV or TSV file, or a SQL database) and creates a Python object with rows and columns called data frame that looks very similar to table in a statistical software (think Excel or SPSS for example. People who are familiar with R would see similarities to R too).

In order to use Pandas in your Python IDE (Integrated Development Environment) like Jupyter Notebook or Spyder (both of them come with Anaconda by default), you need to import the Pandas library first. Importing a library means loading it into the memory and then it's there for you to work with. In order to import Pandas all you have to do is run the following code:

```
# import pandas as pd
# import numpy as np
```

Usually you would add the second part ('as pd') so you can access Pandas with 'pd.command' instead of needing to write 'pandas.command' every time you need to use it. Also, you would import **numpy** as well, because it is very useful library for scientific computing with Python. Now Pandas is ready for use! Remember, you would need to do it every time you start a new Jupyter Notebook, Spyder file etc.

Working with Pandas

Loading and Saving Data with Pandas

When you want to use Pandas for data analysis, you'll usually use it in one of three different ways: - Convert a Python's list, dictionary or Numpy array to a Pandas data frame - Open a local file using Pandas, usually a CSV file, but could also be a delimited text file (like TSV), Excel, etc - Open a remote file or database like a CSV or a JSONon a website through a URL or read from a SQL table/database

There are different commands to each of these options, but when you open a file, they would look like this:

```
# pd.read_filetype()
```

Example:

```
# d = pd.read_csv('https://url/file.csv')
```

As I mentioned before, there are different filetypes Pandas can work with, so you would replace "filetype" with the actual, well, filetype (like CSV). You would give the path, filename etc inside the parenthesis. Inside the parenthesis you can also pass different arguments that relate to how to open the file. There are numerous arguments and in order to know all you them, you would have to read the documentation (for example, the documentation for pd.read_csv() would contain all the arguments you can pass in this Pandas command).

In order to convert a certain Python object (dictionary, lists etc) the basic command is: **pd.DataFrame()**

Inside the parenthesis you would specify the object(s) you're creating the data frame from. This command also has different arguments, take a look.

Example:

You can also save a data frame you're working with/on to different kinds of files (like CSV, Excel, JSON and SQL tables). The general code for that is:

```
# df.to_filetype(filename)
```

Review

Now you know:

- what modules are and how they can be useful
- how to use a few of the most commonly used Python libraries
- what namespaces are and how to avoid polluting your local namespace
- how scope works for files in Python

Programmers can do great things if they are not forced to constantly reinvent tools that have already been built. With the power of modules, we can import any code that someone else has shared publicly.

In this lesson, we covered some of the Python Standard Library, but you can explore all the modules that come packaged with every installation of Python at the Python Standard Library documentation.

This is just the beginning. Using a package manager (like conda or pip3), you can install any modules available on the Python Package Index.

The sky's the limit!

Dictionaries

Introduction

A dictionary is an unordered set of **key:value** pairs.

Suppose we want to store the prices of various items sold at a cafe:

- Oatmeal is 3 dollars
- Avocado Toast is 6 dollars
- Carrot Juice is 5 dollars
- Blueberry Muffin is 2 dollars

In Python, we can create a dictionary called menu to store this data:

```
menu = {"oatmeal": 3, "avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2}
print(menu)
```

```
## {'oatmeal': 3, 'avocado toast': 6, 'carrot juice': 5, 'blueberry muffin': 2}
```

Notice that:

- A dictionary begins and ends with curly braces ({ and }).
- Each item consists of a key (i.e., "oatmeal") and a value (i.e., 3)
- Each key: value pair (i.e., "oatmeal": 3 or "avocado toast": 6) is separated by a comma (,)
- It's considered good practice to insert a space () after each comma, but your code will still run without the space.

Dictionaries provide us with a way to map pieces of data to each other, so that we can quickly find values that are associated with one another.

Make a Dictionary

In the previous exercise we saw a dictionary that maps strings to numbers (i.e., "oatmeal": 3). However, the keys can be numbers as well. For example, if we were mapping restaurant bill subtotals to the bill total after tip, a dictionary could look like:

```
subtotal_to_total = {20: 24, 10: 12, 5: 6, 15: 18}
```

Values can be any type. You can use a string, a number, a list, or even another dictionary as the value associated with a key!

For example:

```
students_in_classes = {"software design": ["Aaron", "Delila", "Samson"], "cartography": ["Christopher",
```

The list ["Aaron", "Delila", "Samson"], which is the value for the key "software design", represents the students in that class.

You can also mix and match key and value types. For example:

```
person = {"name": "Shuri", "age": 18, "siblings": ["T'Chaka", "Ramonda"]}
```

Invalid Keys

We can have a list or a dictionary as a value of an item in a dictionary, but we cannot use these data types as keys of the dictionary. If we try to, we will get a TypeError. For example:

```
#  powers = {[1, 2, 4, 8, 16]: 2, [1, 3, 9, 27, 81]: 3}
# TypeError: unhashable type: 'list'
```

The word "unhashable" in this context means that this 'list' is an object that can be changed. Dictionaries in Python rely on each key having a hash value, a specific identifier for the key. If the key can change, that hash value would not be reliable. So the keys must always be unchangeable, hashable data types, like numbers or strings.

Empty Dictionary

A dictionary doesn't have to contain anything. You can create an empty dictionary:

```
empty_dict = {}
```

We can create an empty dictionary when we plan to fill it later based on some other input. We will explore ways to fill a dictionary in the next exercise.

Add a key

To add a single key : value pair to a dictionary, we can use the syntax:

```
#   my_dict["new_key"] = "new_value"
```

For example, if we had our menu object from the first exercise:

```
menu = {"oatmeal": 3, "avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2}
```

and we wanted to add a new item, “cheesecake” for 8 dollars, we could use:

```
menu["cheesecake"] = 8
print(menu)
```

```
## {'oatmeal': 3, 'avocado toast': 6, 'carrot juice': 5, 'blueberry muffin': 2, 'cheesecake': 8}
```

Another example:

```
animals_in_zoo={}
animals_in_zoo["zebras"]=8
animals_in_zoo["monkeys"]=12
animals_in_zoo["dinosaurs"]=0
print(animals_in_zoo)
```

```
## {'zebras': 8, 'monkeys': 12, 'dinosaurs': 0}
```

Add Multiple Keys

If we wanted to add multiple key : value pairs to a dictionary at once, we can use the .update() method.

Looking at our sensors object from the first exercise:

```
sensors =  {"living room": 21, "kitchen": 23, "bedroom": 20}
```

If we wanted to add 3 new rooms, we could use:

```
sensors.update({"pantry": 22, "guest room": 25, "patio": 34})
```

which would add all three items to the sensors dictionary. Now, sensors looks like:

```
print(sensors)
```

```
## {'living room': 21, 'kitchen': 23, 'bedroom': 20, 'pantry': 22, 'guest room': 25, 'patio': 34}
```

Overwrite Values

We know that we can add a key by using syntax like:

```
#  menu['avocado toast'] = 7
```

which will create a key ‘avocado toast’ and set the value to 7. But what if we already have an ‘avocado toast’ entry in the menu dictionary? In that case, our value assignment would overwrite the existing value attached to the key ‘avocado toast’.

```
menu = {"oatmeal": 3, "avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2}
menu["oatmeal"] = 5
print(menu)
```

```
## {'oatmeal': 5, 'avocado toast': 6, 'carrot juice': 5, 'blueberry muffin': 2}
```

Notice the value of “oatmeal” has now changed to 5.

List Comprehensions to Dictionaries

Let’s say we have two lists that we want to combine into a dictionary, like a list of students and a list of their heights, in inches:

```
names = ['Jenny', 'Alexus', 'Sam', 'Grace']
heights = [61, 70, 67, 64]
```

Python allows you to create a dictionary using a list comprehension, with this syntax:

```
students = {key:value for key,value in zip(names, heights)}
print(students)
```

```
## {'Jenny': 61, 'Alexus': 70, 'Sam': 67, 'Grace': 64}
```

Remember that zip() combines two lists into a zipped list of pairs.

```
names_and_heights = zip(names, heights)
print(list(names_and_heights))
```

```
## [('Jenny', 61), ('Alexus', 70), ('Sam', 67), ('Grace', 64)]
```

This list comprehension:

- Takes a pair from the zipped list of pairs from names and heights
- Names the elements in the pair key (the one originally from the names list) and value (the one originally from the heights list)
- Creates a key : value item in the students dictionary
- Repeats steps 1-3 for the entire list of pairs

Review

Now we know:

- How to create a dictionary
- How to add elements to a dictionary
- How to update elements in a dictionary
- How to use a list comprehension to create a dictionary from two lists

Example:

We are building a music streaming service. We have provided two lists, representing songs in a user's library and the amount of times each song has been played.

```
songs = ["Like a Rolling Stone", "Satisfaction", "Imagine", "What's Going On", "Respect", "Good Vibrati
playcounts = [78, 29, 44, 21, 89, 5]
#Create a dictionary named plays
plays = {key:value for key,value in zip(songs, playcounts)}
# Add the song "Purple Haze" and the playcount is 1.
plays["Purple Haze"]=1
#  Aretha Franklin fever and listened to "Respect" 5 more times.
plays["Respect"]=94
# Create a dictionary called library that has two key: value pairs
library={"The Best Songs":plays, "Sunday Feelings":{}}

print(plays)
```

```
## {'Like a Rolling Stone': 78, 'Satisfaction': 29, 'Imagine': 44, "What's Going On": 21, 'Respect': 94
```

```
print(library)
```

```
## {'The Best Songs': {'Like a Rolling Stone': 78, 'Satisfaction': 29, 'Imagine': 44, "What's Going On"
```

Using Dictionaries

Now that we know how to create a dictionary, we can start using already created dictionaries to solve problems.

Get A Key

Once you have a dictionary, you can access the values in it by providing the key. For example, let's imagine we have a dictionary that maps buildings to their heights, in meters:

```
building_heights = {"Burj Khalifa": 828, "Shanghai Tower": 632, "Abraj Al Bait": 601, "Ping An": 599, "
```

Then we can access the data in it like this:

```
building_heights["Burj Khalifa"]
```

```
## 828
```

```
building_heights["Ping An"]
```

```
## 599
```

Get an Invalid Key

Let's say we have our dictionary of building heights from the last exercise. What if we wanted to know the height of the Landmark 81 in Ho Chi Minh City? We could try:

```
# print(building_heights["Landmark 81"])
# KeyError: 'Landmark 81'
```

But "Landmark 81" does not exist as a key in the building_heights dictionary! So this will throw a KeyError: 'Landmark 81'

One way to avoid this error is to first check if the key exists in the dictionary:

```
key_to_check = "Landmark 81"

if key_to_check in building_heights:
  print(building_heights["Landmark 81"])
else:
  print("Not in the dictionary")
```

```
## Not in the dictionary
```

This will not throw an error, because key__to__check in building__heights will return False, and so we never try to access the key.

Try/Except to Get a Key

We saw that we can avoid KeyErrors by checking if a key is in a dictionary first. Another method we could use is a try/except:

```
key_to_check = "Landmark 81"
try:
  print(building_heights[key_to_check])
except KeyError:
  print("That key doesn't exist!")
```

```
## That key doesn't exist!
```

When we try to access a key that doesn't exist, the program will go into the except block and print "That key doesn't exist!".

Safely Get a Key

We saw in the last exercise that we had to add a key:value pair to a dictionary in order to avoid a KeyError. This solution is not sustainable. We can't predict every key a user may call and add all of those placeholder values to our dictionary!

Dictionaries have a .get() method to search for a value instead of the my_dict[key] notation we have been using. If the key you are trying to .get() does not exist, it will return None by default:

```
building_heights = {"Burj Khalifa": 828, "Shanghai Tower": 632, "Abraj Al Bait": 601, "Ping An": 599, "

building_heights.get("Shanghai Tower")
#this line will return 632:
```

```
## 632
```

```
building_heights.get("My House")
#this line will return None:
```

You can also specify a value to return if the key doesn't exist. For example, we might want to return a building height of 0 if our desired building is not in the dictionary:

```
building_heights.get('Shanghai Tower', 0)
```

```
## 632
```

```
building_heights.get('Mt Olympus', 0)
```

```
## 0
```

```
building_heights.get('Kilimanjaro', 'No Value')
```

```
## 'No Value'
```

Delete a Key

Sometimes we want to get a key and remove it from the dictionary. Imagine we were running a raffle, and we have this dictionary mapping ticket numbers to prizes:

```
raffle = {223842: "Teddy Bear", 872921: "Concert Tickets", 320291: "Gift Basket", 412123: "Necklace", 2
```

When we get a ticket number, we want to return the prize and also remove that pair from the dictionary, since the prize has been given away. We can use .pop() to do this. Just like with .get(), we can provide a default value to return if the key does not exist in the dictionary:

```
raffle.pop(320291, "No Prize")
```

```
## 'Gift Basket'
```

.pop() works to delete items from a dictionary, when you know the key value.

```
raffle
```

```
## {223842: 'Teddy Bear', 872921: 'Concert Tickets', 412123: 'Necklace', 298787: 'Pasta Maker'}
```

```
raffle.pop(100000, "No Prize")
```

```
## 'No Prize'
```

```
raffle
```

```
## {223842: 'Teddy Bear', 872921: 'Concert Tickets', 412123: 'Necklace', 298787: 'Pasta Maker'}
```

Example:

You are designing the video game Big Rock Adventure. We have provided a dictionary of items that are in the player's inventory which add points to their health meter. In one line, add the corresponding value of the key "stamina grains" to the health_points variable and remove the item "stamina grains" from the dictionary. If the key does not exist, add 0 to health_points.

```
available_items = {"health potion": 10, "cake of the cure": 5, "green elixir": 20, "strength sandwich":
health_points = 20

health_points += available_items.pop("stamina grains",0)
health_points += available_items.pop("power stew",0)
health_points += available_items.pop("mystic bread",0)

print(available_items)
```

```
## {'health potion': 10, 'cake of the cure': 5, 'green elixir': 20, 'strength sandwich': 25}
```

```
print(health_points)
```

```
## 65
```

Get All Keys

Sometimes we want to operate on all of the keys in a dictionary. For example, if we have a dictionary of students in a math class and their grades:

```
test_scores = {"Grace":[80, 72, 90], "Jeffrey":[88, 68, 81], "Sylvia":[80, 82, 84], "Pedro":[98, 96, 95
```

We want to get a roster of the students in the class, without including their grades. We can do this with the built-in list() function:

```
list(test_scores)
```

```
## ['Grace', 'Jeffrey', 'Sylvia', 'Pedro', 'Martin', 'Dina']
```

Dictionaries also have a .keys() method that returns a **dict_keys** object. A dict_keys object is a view object, which provides a look at the current state of the dicitonary, without the user being able to modify anything.

The dict_keys object returned by .keys() is a set of the keys in the dictionary. You cannot add or remove elements from a dict_keys object, but it can be used in the place of a list for iteration:

```
for student in test_scores.keys():
  print(student)
```

```
## Grace
## Jeffrey
## Sylvia
## Pedro
## Martin
## Dina
```

Get All Values

Dictionaries have a .values() method that returns a dict_values object (just like a dict_keys object but for values!) with all of the values in the dictionary. It can be used in the place of a list for iteration:

```
test_scores = {"Grace":[80, 72, 90], "Jeffrey":[88, 68, 81], "Sylvia":[80, 82, 84], "Pedro":[98, 96, 95

for score_list in test_scores.values():
  print(score_list)
```

```
## [80, 72, 90]
## [88, 68, 81]
## [80, 82, 84]
## [98, 96, 95]
## [78, 80, 78]
## [64, 60, 75]
```

There is no built-in function to get all of the values as a list, but if you really want to, you can use:

```
list(test_scores.values())
```

```
## [[80, 72, 90], [88, 68, 81], [80, 82, 84], [98, 96, 95], [78, 80, 78], [64, 60, 75]]
```

However, for most purposes, the dict_list object will act the way you want a list to act.

Example:

```
user_ids = {"teraCoder": 100019, "pythonGuy": 182921, "samTheJavaMaam": 123112, "lyleLoop": 102931, "ke
num_exercises = {"functions": 10, "syntax": 13, "control flow": 15, "loops": 22, "lists": 19, "classes"

users = user_ids.keys()
number_lessons = num_exercises.values()

print(users)
```

```
## dict_keys(['teraCoder', 'pythonGuy', 'samTheJavaMaam', 'lyleLoop', 'keysmithKeith'])
```

```
print(number_lessons)
```

```
## dict_values([10, 13, 15, 22, 19, 18, 18])
```

```
total_exercises=0
for values in num_exercises.values():
  total_exercises+=values
print(total_exercises)
```

```
## 115
```

Get All Items

You can get both the keys and the values with the .items() method. Like .keys() and .values(), it returns a dict_list object. Each element of the dict_list returned by .items() is a tuple consisting of: **(key, value)**

So to iterate through, you can use this syntax:

```
biggest_brands = {"Apple": 184, "Google": 141.7, "Microsoft": 80, "Coca-Cola": 69.7, "Amazon": 64.8}
#Iterate through the tuple
for company,value in biggest_brands.items():
  print(company + " has a value of " + str(value) + " billion dollars. ")
```

```
## Apple has a value of 184 billion dollars.
## Google has a value of 141.7 billion dollars.
## Microsoft has a value of 80 billion dollars.
## Coca-Cola has a value of 69.7 billion dollars.
## Amazon has a value of 64.8 billion dollars.
```

Another example:

```
pct_women_in_occupation = {"CEO": 28, "Engineering Manager": 9, "Pharmacist": 58, "Physician": 40, "Law

for occupation,value in pct_women_in_occupation.items():
  print("Women make up " + str(value) + " percent of " +occupation +"s.")
```

```
## Women make up 28 percent of CEOs.
## Women make up 9 percent of Engineering Managers.
## Women make up 58 percent of Pharmacists.
## Women make up 40 percent of Physicians.
## Women make up 37 percent of Lawyers.
## Women make up 9 percent of Aerospace Engineers.
```

Review

In this section, you've learned how to go through dictionaries and access keys and values in different ways. Specifically you have seen how to:

- Use a key to get a value from a dictionary
- Check for existence of keys
- Find the length of a dictionary
- Remove a key: value pair from a dictionary
- Iterate through keys and values in dictionaries

Example:

You have a pack of tarot cards, tarot. You are going to do a three card spread of your past, present, and future.

```
tarot = { 1:    "The Magician", 2:  "The High Priestess", 3:    "The Empress", 4:   "The Emperor", 5:

spread={}
# The first card you draw is card 13 for past
spread["past"]=tarot.pop(13)
#The second card you draw is card 22 for present
spread["present"]=tarot.pop(22)
# The third card you draw is card 10 for future.
spread["future"]=tarot.pop(10)
# Print "Your {key} is the {value} card."
for key,value in spread.items():
  print("Your " + key + " is the " + value +" card.")
```

```
## Your past is the Death card.
## Your present is the The Fool card.
## Your future is the Wheel of Fortune card.
```

Dictionary Challenges

This lesson will help you review Python functions by providing some challenge exercises involving dictionaries.

As a refresher, function syntax looks like this:

```
#  def some_function(some_input1, some_input2):
#    ... do something with the inputs ...
#    return output
```

For example, a function that counts the number of values in a dictionary that are above a given number would look like this:

```
#  def greater_than_ten(my_dictionary, number):
#    count = 0
#    for value in my_dictionary.values():
#      if value > number:
#        count += 1
#    return count

#Output would look like:
# greater_than_ten({"a":1, "b":2, "c":3}, 0)
# 3
```

Sum Values

A function named sum_values that takes a dictionary named my_dictionary as a parameter. The function should return the sum of the values of the dictionary

```
def sum_values(my_dictionary):
  sum=0
  for val in my_dictionary.values():
    sum += val
  return sum

print(sum_values({"milk":5, "eggs":2, "flour": 3}))
# should print 10
```

```
## 10
```

```
print(sum_values({10:1, 100:2, 1000:3}))
# should print 6
```

```
## 6
```

Even Keys

A function called sum_even_keys that takes a dictionary named my_dictionary, with all integer keys and values, as a parameter. This function should return the sum of the values of all even keys.

```
def sum_even_keys(my_dictionary):
  total = 0
  for key in my_dictionary.keys():
    if key%2 == 0:
      total += my_dictionary[key]
  return total

print(sum_even_keys({1:5, 2:2, 3:3}))
# should print 2
```

```
## 2
```

```
print(sum_even_keys({10:1, 100:2, 1000:3}))
# should print 6
```

```
## 6
```

Add Ten

A function named add_ten that takes a dictionary with integer values named my_dictionary as a parameter. The function should add 10 to every value in my_dictionary and return my_dictionary.

```
#Mehtod 1
def add_ten(my_dictionary):
  keys=list(my_dictionary)
  for key in keys:
    my_dictionary[key] += 10.
  return my_dictionary

print(add_ten({1:5, 2:2, 3:3}))

# should print {1:15, 2:12, 3:13}
```

```
## {1: 15.0, 2: 12.0, 3: 13.0}
```

```
print(add_ten({10:1, 100:2, 1000:3}))
# should print {10:11, 100:12, 1000:13}
```

```
## {10: 11.0, 100: 12.0, 1000: 13.0}
```

```
#Mehtod 2
def add_ten(my_dictionary):
  for key in my_dictionary.keys():
    my_dictionary[key] += 10
  return my_dictionary

print(add_ten({1:5, 2:2, 3:3}))
# should print {1:15, 2:12, 3:13}
```

```
## {1: 15, 2: 12, 3: 13}
```

```
print(add_ten({10:1, 100:2, 1000:3}))
# should print {10:11, 100:12, 1000:13}
```

```
## {10: 11, 100: 12, 1000: 13}
```

Values That Are Keys

A function named values_that_are_keys that takes a dictionary named my_dictionary as a parameter. This function should return a list of all values in the dictionary that are also keys.

```
def values_that_are_keys(my_dictionary):
  same=[]
  for key in my_dictionary.keys():
    for value in my_dictionary.values():
      if key==value:
        same.append(key)
  return same

print(values_that_are_keys({1:100, 2:1, 3:4, 4:10}))
# should print [1, 4]
```

```
## [1, 4]
```

```
print(values_that_are_keys({"a":"apple", "b":"a", "c":100}))
# should print ["a"]
```

```
## ['a']
```

Largest Value

A function named max_key that takes a dictionary named my_dictionary as a parameter. The function should return the key associated with the largest value in the dictionary.

```
def max_key(my_dictionary):
  largest_key=''
  #Smallest number possible to avoid guessing.
  largest_value=float("-inf")
  for key,value in my_dictionary.items():
    if value > largest_value:
      largest_value=value
      largest_key=key
  return largest_key

print(max_key({1:100, 2:1, 3:4, 4:10}))
# should print 1
```

```
## 1
```

```
print(max_key({"a":100, "b":10, "c":1000}))
# should print "c"
```

```
## c
```

Word Length Dict

A function named word_length_dictionary that takes a list of strings named words as a parameter. The function should return a dictionary of key/value pairs where every key is a word in words and every value is the length of that word.

```
def word_length_dictionary(words):
  dicc={}
  leng=0
  for word in words:
    leng=len(word)
    dicc[word]=leng
  return dicc

print(word_length_dictionary(["apple", "dog", "cat"]))
# should print {"apple":5, "dog": 3, "cat":3}
```

```
## {'apple': 5, 'dog': 3, 'cat': 3}
```

```
print(word_length_dictionary(["a", ""]))
# should print {"a": 1, "": 0}
```

```
## {'a': 1, '': 0}
```

Frequency Count

A function named frequency_dictionary that takes a list of elements named words as a parameter. The function should return a dictionary containing the frequency of each element in words.

```
#Method 1
def frequency_dictionary(words):
  freq = {}
  for word in words:
    if word not in freq.keys():
      freq[word]=1
    else:
      freq[word]+=1
  return freq


print(frequency_dictionary(["apple", "apple", "cat", 1]))
# should print {"apple":2, "cat":1, 1:1}
```

```
## {'apple': 2, 'cat': 1, 1: 1}
```

```
print(frequency_dictionary([0,0,0,0,0]))
# should print {0:5}
```

```
## {0: 5}
```

```
#Method 2
def frequency_dictionary(words):
  freqs = {}
  for word in words:
    if word not in freqs:
        freqs[word] = 0
    freqs[word] += 1
  return freqs

print(frequency_dictionary(["apple", "apple", "cat", 1]))
# should print {"apple":2, "cat":1, 1:1}
```

```
## {'apple': 2, 'cat': 1, 1: 1}
```

```
print(frequency_dictionary([0,0,0,0,0]))
# should print {0:5}
```

```
## {0: 5}
```

Unique Values

A function named unique_values that takes a dictionary named my_dictionary as a parameter. The function should return the number of unique values in the dictionary.

```
def unique_values(my_dictionary):
  unique=[]
  for val in my_dictionary.values():
    if val not in unique:
      unique.append(val)
  return len(unique)

print(unique_values({0:3, 1:1, 4:1, 5:3}))
# should print 2
```

```
## 2
```

```
print(unique_values({0:3, 1:3, 4:3, 5:3}))
# should print 1
```

```
## 1
```

Count First Letter

A function named count_first_letter that takes a dictionary named names as a parameter. names should be a dictionary where the key is a last name and the value is a list of first names. The function should return a new dictionary where each key is the first letter of a last name, and the value is the number of people whose last name begins with that letter.

For example, the dictionary might look like this: **names = {"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Lannister": ["Jaime", "Cersei", "Tywin"]}** should return **{"S" : 4, "L": 3}**

Notice that you can treat the dictionary as any other list but you will go through the keys as elements.

```
names = {"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Lannister": ["Jaime", "Cersei", "Tywin"]
for key in names:
  print(key)
```

```
## Stark
## Snow
## Lannister
```

```
test_dic={}
for key in names:
  first_letter = key[0]
  test_dic[first_letter] = 0
  print(len(names[key]))
```

```
## 3
## 1
## 3
```

```
print(test_dic)
```

```
## {'S': 0, 'L': 0}
```

```
names = {"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Lannister": ["Jaime", "Cersei", "Tywin"]

def count_first_letter(names):
  letters = {}
  for key in names:
    first_letter = key[0]
    if first_letter not in letters:
      letters[first_letter] = 0
    letters[first_letter] += len(names[key])
  return letters

print(count_first_letter({"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Lannister": ["Jaime", "
# should print {"S": 4, "L": 3}
```

```
## {'S': 4, 'L': 3}
```

```
print(count_first_letter({"Stark": ["Ned", "Robb", "Sansa"], "Snow" : ["Jon"], "Sannister": ["Jaime", "
# should print {"S": 7}
```

```
## {'S': 7}
```

Function Arguments

Parameters and Arguments

Python's functions offer us a very expressive syntax. We're going to look into some of the finer details of how functions in Python work and some techniques we can use to be more intuitive while writing and calling functions.

First, let's consider some definitions:

- A **parameter** is a variable in the definition of a function.
- An **argument** is the value being passed into a function call.
- A **function definition** begins with def and contains the entire following indented block.
- And **function calls** are the places a function is invoked, with parentheses, after its definition

The following block will help us to better understand:

```python
# The "def" keyword is the start of a function definition
def function_name(parameter1, parameter2):
  # The placeholder variables used inside a function definition are called parameters
  print(parameter1)
  return parameter2
# The outdent signals the end of the function definition

# "Arguments" are the values passed into a function call
argument1 = "argument 1"
argument2 = "argument 2"

# A function call uses the functions name with a pair of parentheses
# and can return a value
return_val = function_name(argument1, argument2)
```

```
## argument 1
```

In the above code we defined the function function_name that takes two parameters, parameter1 and parameter2. We then create two variables with the values "argument 1" and "argument 2" and proceed to call function_name with the two arguments.

Some of this terminology can be used inconsistently between schools, people, and businesses. Some people don't differentiate between "parameter" and "argument" when speaking. It's useful here because we're going to be looking at a lot of behavior that looks very similar in a function definition and a function call, but will be subtly different. But the distinction is sometimes unnecessary, so don't get too hung up if something is called a "parameter" that should be an "argument" or vice versa.

None: It's Nothing!

How do you define a variable that you can't assign a value to yet? You use "**None**". "None" is a special value in Python. It is unique (there can't be two different Nones) and immutable (you can't update None or assign new attributes to it).

```python
none_var = None
if none_var:
  print("Hello!")
else:
  print("Goodbye")

# Prints "Goodbye"
```

```
## Goodbye
```

None is *falsy*, meaning that it evaluates to False in an if statement, which is why the above code prints "Goodbye". None is also unique, which means that you can test if something is None using the is keyword.

```
# first we define session_id as None
session_id = None

if session_id is None:
  print("session ID is None!")
  # this prints out "session ID is None!"

# we can assign something to session_id
# if active_session:
#   session_id = active_session.id

# but if there's no active_session, we don't send sensitive data
# if session_id is not None:
#   send_sensitive_data(session_id)
```

```
## session ID is None!
```

Above we initialize our session_id to None, then set our session_id if there is an active session. Since session_id could either be None we check if session_id is None before sending our sensitive data.

Default Return

What does a function return when it doesn't return anything? This sounds like a riddle, but there is a correct answer. A Python function that does not have any explicit return statement will return None after completing. This means that all functions in Python return something, whether it's explicit or not. For example:

```
def no_return():
  print("You've hit the point of no return")
  # notice there's no return statement

here_it_is = no_return()
```

```
## You've hit the point of no return
```

```
print(here_it_is)
# Prints out "None"
```

```
## None
```

Above we defined a function called no_return() and saved the result to a variable here_it_is. When we print() the value of here_it_is we get None, referring to Python's None. It's possible to make this syntax even more explicit — a return statement without any value returns None also.

```
def fifty_percent_off(item):
  # Check if item.cost exists
  if hasattr(item, 'cost'):
    return item.cost / 2
  # If not, return None
  return
product="Some random product"
sale_price = fifty_percent_off(product)

if sale_price is None:
  print("This product is not for sale!")
```

```
## This product is not for sale!
```

Here we have implemented a function that returns the cost of a product with “50% Off” (or “half price”). We check (with *hasattr*)if the item passed to our function has a cost attribute. If it exists, we return half the cost. If not, we simply return, which returns None.

When we plug a product into this function, we can expect two possibilities: the first is that the item has a cost and this function returns half of that. The other is that item does not have a listed cost and so the function returns None. We can put error handling in the rest of our code, if we get None back from this function that means whatever we’re looking at isn’t for sale!

Another example:

Lots of everyday Python functions return None. What’s the return value of a call to print()? Since print() is a function it must return something.

```
# If we set a variable equal to a prin statement, this happens:
prints_return=print("What does this print function return?")
```

```
## What does this print function return?
```

```
# If we print this variable:
print(prints_return)
```

```
## None
```

Just one more example: We have a list, what happens if we sort it and save the sorted list into a variable, and print it:

```
sort_this_list = [14, 631, 4, 51358, 50000000]
list_sort_return=sort_this_list.sort()
print(list_sort_return)
```

```
## None
```

It might be surprising, but .sort() sorts a list in place. Python has a different function, sorted() that takes a list as an argument and returns the sorted list.

What’s in common with these two functions (print and .sort()) that return None? They both have side-effects besides returning a value.

Default Arguments

Function arguments are required in Python. So a standard function definition that defines two parameters requires two arguments passed into the function.

```
# Function definition with two required parameters
def create_user(username, is_admin):
  return str(username) + '_' + str(is_admin)

# Function call with all two required arguments
user1 = create_user('johnny_thunder', True)
print(user1)

# Raises a "TypeError: Missing 1 required positional argument"
# user2 = create_user('djohansen')
```

```
## johnny_thunder_True
```

Above we defined our function, create_user, with two parameters. We first called it with two arguments, but then tried calling it with only one argument and got an error. What if we had sensible defaults for this argument?

Not all function arguments in Python need to be required. If we give a default argument to a Python function that argument won't be required when we call the function.

```
# Function defined with one required and one optional parameter
def create_user(username, is_admin=False):
  return str(username) + '_' + str(is_admin)

# Calling with two arguments uses the default value for is_admin
user2 = create_user('djohansen')
print(user2)
```

```
## djohansen_False
```

Above we defined create_user with a default argument for is_admin, so we can call that function with only the one argument 'djohansen'. It assumes the default value for is_admin: False. We can make both of our parameters have a default value (therefore making them all optional).

```
# We can make all three parameters optional
def create_user(username=None, is_admin=False):
  if username is None:
    username = "Guest"
    return str(username) + '_' + str(is_admin)
  else:
    return str(username) + '_' + str(is_admin)


# So we can call with just one value
user3 = create_user('ssylvain')
print(user3)
# Or no value at all, which would create a Guest user
```

```
## ssylvain_False
```

```
user4 = create_user()
print(user4)
```

```
## Guest_False
```

Above we define the function with all optional parameters, if we call it with one argument that gets interpreted as username. We can call it without any arguments at all, which would only use the default values. View Functions

Using Keyword and Positional Arguments

Not all of your arguments need to have default values. But Python will only accept functions defined with their parameters in a specific order. The required parameters need to occur before any parameters with a default argument.

```
# This code raises a TypeError because the required parameters ocur after the default.

#  def create_user(is_admin=False, username, password):
#    create_email(username, password)
#    set_permissions(is_admin)
```

In the above code, we attempt to define a default argument for is_admin without defining default arguments for the later parameters: username and password. If we want to give is_admin a default argument, we need to list it last in our function definition:

```
# Works perfectly

#  def create_user(username, password, is_admin=False):
#    create_email(username, password)
#    set_permissions(is_admin)
```

Keyword Arguments

When we call a function in Python, we need to list the arguments to that function to match the order of the parameters in the function definition. We don't necessarily need to do this if we pass keyword arguments.

We use keyword arguments by passing arguments to a function with a special syntax that uses the names of the parameters. This is useful if the function has many optional default arguments or if the order of a function's parameters is hard to tell. Here's an example of a function with a lot of optional arguments.

```
# We define a function with a bunch of default arguments
def log_message(logging_style="shout", message="", font="Times", date=None):
  if logging_style == 'shout':
    # capitalize the message
    message = message.upper()
  print(message, date)

# Now call the function with keyword arguments
log_message(message="Hello from the past", date="November 20, 1693")
```

```
## HELLO FROM THE PAST November 20, 1693
```

```
log_message(date="27. juni", message="Hello from the future")
```

```
## HELLO FROM THE FUTURE 27. juni
```

Above we defined log_message(), which can take from 0 to 4 arguments. Since it's not clear which order the four arguments might be defined in, we can use the parameter names to call the function. Notice that in our function call we use this syntax: message="Hello from the past". The key word message here needs to be the name of the parameter we are trying to pass the argument to.

Don't Use Mutable Default Arguments

When writing a function with default arguments, it can be tempting to include an empty list as a default argument to that function. Let's say you have a function called populate_list that has two required arguments, but it's easy to see that we might want to give it some default arguments in case we don't have either list_to_populate or length every time. So we'd give it these defaults:

```
def populate_list(list_to_populate=[], length=1):
  for num in range(length):
    list_to_populate.append(num)
  return list_to_populate
```

It's reasonable to believe that list_to_populate will be given an empty list every time it's called. This isn't the case! list_to_populate will be given a new list once, in its definition, and all subsequent function calls will modify the same list. This will happen:

```
returned_list = populate_list(length=4)
print(returned_list)
# Prints [0, 1, 2, 3] -- this is expected
```

```
## [0, 1, 2, 3]
```

```python
returned_list = populate_list(length=6)
print(returned_list)
# Prints [0, 1, 2, 3, 0, 1, 2, 3, 4, 5] -- this is a surprise!
```

```
## [0, 1, 2, 3, 0, 1, 2, 3, 4, 5]
```

When we call populate_list a second time we'd expect the list [0, 1, 2, 3, 4, 5]. But the same list is used both times the function is called, causing some side-effects from the first function call to creep into the second. This is because a list is a mutable object.

A **mutable** object refers to various data structures in Python that are intended to be mutated, or changed. A list has append and remove operations that change the nature of a list. Sets and dictionaries are two other mutable objects in Python.

It might be helpful to note some of the objects in Python that are not mutable (and therefore OK to use as default arguments). **int, float, and other numbers can't be mutated** (arithmetic operations will return a new number). **tuples are a kind of immutable list**. **Strings** are also immutable — operations that update a string will all return a completely new string.

Example:

We've written a helper function that adds a new menu item to an order in a point-of-sale system. As you can see, we can start a new order by calling update_order without an argument for current_order. Unfortunately, there's a bug in our code causing some previous order contents to show up on other people's bills!

First, try to guess what the output of this code will be. Then, check the output.

```python
def update_order(new_item, current_order=[]):
  current_order.append(new_item)
  return current_order

# First order, burger and a soda
order1 = update_order({'item': 'burger', 'cost': '3.50'})
order1 = update_order({'item': 'soda', 'cost': '1.50'}, order1)

# Second order, just a soda
order2 = update_order({'item': 'soda', 'cost': '1.50'})

# What's in that second order again?
print(order2)
```

```
## [{'item': 'burger', 'cost': '3.50'}, {'item': 'soda', 'cost': '1.50'}, {'item': 'soda', 'cost': '1.5
```

Two sodas and a burger! And all the customer 2 wanted was a soda. You'll notice, if you print out order1 it's the same exact list as order2. Any updates to one will update the other (and will affect future calls of update_order). We'll fix this function in the next exercise.

Using None as a Sentinel

So if we can't use a list as a default argument for a function, what can we use? If we want an empty list, we can use None as a special value to indicate we did not receive anything. After we check whether an argument was provided we can instantiate a new list if it wasn't.

```python
def add_author(authors_books, current_books=None):
  if current_books is None:
    current_books = []
  current_books.extend(authors_books)
  return current_books
```

In the above function, current_books is a value expected to be a list. But we don't require it. If someone calls add_author() without giving an argument for current_books, we supply an empty list. This way multiple calls to add_author won't include data from previous calls to add_author. Recall the extend() method adds all the elements of an iterable (list, tuple, string etc.) to the end of the list.

Now we can fix our previous code:

```
def update_order(new_item, current_order=None):
  if current_order is None:
    current_order = []
  current_order.append(new_item)
  return current_order

# First order, burger and a soda
order1 = update_order({'item': 'burger', 'cost': '3.50'})
order1 = update_order({'item': 'soda', 'cost': '1.50'}, order1)

print(order1)

# Second order, just a soda
```

```
## [{'item': 'burger', 'cost': '3.50'}, {'item': 'soda', 'cost': '1.50'}]
```

```
order2 = update_order({'item': 'soda', 'cost': '1.50'})

# What's in that second order again?
print(order2)
```

```
## [{'item': 'soda', 'cost': '1.50'}]
```

Unpacking Multiple Returns

A Python function can return multiple things. This is especially useful in cases where bundling data into a different structure (a dictionary or a list, for instance) would be excessive. In Python we can return multiple pieces of data by separating each variable with a comma:

```
def multiple_returns(cool_num1, cool_num2):
  sum_nums = cool_num1 + cool_num2
  div_nums = cool_num1 / cool_num2
  return sum_nums, div_nums
```

Above we created a function that returns two results, sum_nums and div_nums. What happens when we call the function?

```
suma_and_div = multiple_returns(20, 10)

print(suma_and_div)
# Prints "(30, 2)"
```

```
## (30, 2.0)
```

```
print(suma_and_div[0])
# Prints "30"
```

```
## 30
```

So we get those two values back in what's called a **tuple**, an immutable list-like object indicated by parentheses. We can index into the tuple the same way as a list and so suma_and_div[0] will give us our sum_nums value and suma_and_div[1] will produce our div_nums value.

What if we wanted to save these two results in separate variables? Well we can by unpacking the function return. We can list our new variables, comma-separated, that correspond to the number of values returned:

```
suma, div = multiple_returns(18, 9)

print(suma)
# Prints "27"
```

```
## 27
```

```
print(div)
# Prints "2"
```

```
## 2.0
```

Above we were able to unpack the two values returned into separate variables.

Another example:

```
def scream_and_whisper(text):
    scream = text.upper()
    whisper = text.lower()
    return scream, whisper

the_scream, the_whisper = scream_and_whisper("Hello There!")

print(the_scream)
```

```
## HELLO THERE!
```

```
print(the_whisper)
```

```
## hello there!
```

Positional Argument Unpacking

We don't always know how many arguments a function is going to receive, and sometimes we want to handle any possibility that comes at us. Python gives us two methods of unpacking arguments provided to functions. The first method is called positional argument unpacking, because it unpacks arguments given by position. Here's what that looks like:

```
def shout_strings(*args):
  for argument in args:
    print(argument.upper())

shout_strings("hi", "what do we have here", "cool, thanks!")
# Prints out:
# "HI"
# "WHAT DO WE HAVE HERE"
# "COOL, THANKS!"
```

```
## HI
## WHAT DO WE HAVE HERE
## COOL, THANKS!
```

In shout_strings() we use a single asterisk (*) to indicate we'll accept any number of positional arguments passed to the function. Our parameter **args is a tuple** of all of the arguments passed. In this case args has three values inside, but it can have many more (or fewer).

Note that args is just a parameter name, and we aren't limited to that name (although it is rather standard practice). We can also have other positional parameters before our *args parameter. We can't, as we'll see, :

```
def truncate_sentences(length, *sentences):
  for sentence in sentences:
    print(sentence[:length])

truncate_sentences(8, "What's going on here", "Looks like we've been cut off")
# Prints out:
# "What's g"
# "Looks li"
```

```
## What's g
## Looks li
```

Above we defined a function truncate_sentences that takes a length parameter and also any number of sentences. The function prints out the first length many characters of each sentence in sentences.

Example:

The Python library os.path has a function called join(). join() takes an arbitrary number of paths as arguments. We use the join() function to join all three of the path segments, and print out the results! We write a function called myjoin() which takes an arbitrary number of strings and appends them all together, similar to os.path.join().

Notice that os.path.join() does more than concatenate strings, it adds file separators between those strings, but what's important here is using the * key to unpack a function argument.

```
from os.path import join
# Method 1
path_segment_1 = "/Home/User"
path_segment_2 = "Codecademy/videos"
path_segment_3 = "cat_videos/surprised_cat.mp4"

print(join(path_segment_1, path_segment_2, path_segment_3))

# Method 2
```

```
## /Home/User/Codecademy/videos/cat_videos/surprised_cat.mp4
```

```
def myjoin(*args):
  joined_string = args[0]
  for arg in args[1:]:
    joined_string += '/' + arg
  return joined_string

print(myjoin(path_segment_1, path_segment_2, path_segment_3))
```

```
## /Home/User/Codecademy/videos/cat_videos/surprised_cat.mp4
```

Keyword Argument Unpacking

Python doesn't stop at allowing us to accept unlimited positional parameters, it gives us the power to define functions with unlimited keyword parameters too. The syntax is very similar, but uses two asterisks (**)

instead of one. Instead of args, we call this kwargs — as a shorthand for keyword arguments. For .get see Safely get a key

```python
def arbitrary_keyword_args(**kwargs):
  print(type(kwargs))
  print(kwargs)
  # See if there's an "anything_goes" keyword arg
  # and print it
  print(kwargs.get('anything_goes'))
```

```python
arbitrary_keyword_args(this_arg="wowzers", anything_goes=101)
# Prints "<class 'dict'>
# Prints "{'this_arg': 'wowzers', 'anything_goes': 101}"
# Prints "101"
```

```
## <class 'dict'>
## {'this_arg': 'wowzers', 'anything_goes': 101}
## 101
```

As you can see, **kwargs gives us a dictionary with all the keyword arguments that were passed to arbitrary_keyword_args. We can access these arguments using standard dictionary methods.

Since we're not sure whether a keyword argument will exist, it's probably best to use the dictionary's .get() method to safely retrieve the keyword argument. Do you remember what .get() returns if the key is not in the dictionary? It's None!

Example:

Remember the string .format() method can utilize keyword argument syntax if you give placeholder names in curly braces in a string. For example:

```python
"{place} is {adjective} this time of year.".format(place="New York", adjective="quite cold, actually")
```

```
## 'New York is quite cold, actually this time of year.'
```

Another example:

```python
# We give the .format method our arguments.
print("My name is {name} and I'm feeling {feeling}.".format(
    name="Diego",
  feeling="good",
))


# We update the function create_products() to take arbitrary keyword arguments
# products_dict contains all the keyword arguments passed to create_products().
```

```
## My name is Diego and I'm feeling good.
```

```python
def create_products(**products_dict):
  for name, price in products_dict.items():
    print('{name} created, being sold for ${price}'.format(name=name,price=price))

# Checkpoint 3
# We pass in this dictionary as a series of keyword
create_products(
  Baseball='3',
  Shirt='14',
  Guitar='70')
```

```
## Baseball created, being sold for $3
## Shirt created, being sold for $14
## Guitar created, being sold for $70
```

Using Both Keyword and Positional Unpacking

This keyword argument unpacking syntax can be used even if the function takes other parameters. However, the parameters must be listed in this order:

1. All named positional parameters
2. An unpacked positional parameter (*args)
3. All named keyword parameters
4. An unpacked keyword parameter (**kwargs)

Here's a function with all possible types of parameter:

```
def main(filename, *args, user_list=None, **kwargs):
  if user_list is None:
    user_list = []

  if '-a' in args:
    user_list = all_users()

  if kwargs.get('active'):
    user_list = [user for user in user_list]

  with open(filename) as user_file:
    user_file.write(user_list)
```

Looking at the signature of main() we define four different kinds of parameters. The first, filename is a normal required positional parameter. The second, *args, is all positional arguments given to a function after that organized as a tuple in the parameter args. The third, user_list, is a keyword parameter with a default value. Lastly, **kwargs is all other keyword arguments assembled as a dictionary in the parameter kwargs.

A possible call to the function could look like this:

```
#  main("files/users/userslist.txt",
#       "-d",
#       "-a",
#       save_all_records=True,
#       user_list=["current_users"])
```

In the body of main() these arguments would look like this:

- filename == "files/users/userslist.txt"
- args == ('-d', '-a)
- user_list == current_users
- kwargs == {'save_all_records': True}

We can use all four of these kinds of parameters to create functions that handle a lot of possible arguments being passed to them.

Passing Containers as Arguments

Not only can we accept arbitrarily many parameters to a function in our definition, but Python also supports a syntax that makes deconstructing any data that you have on hand to feed into a function that accepts these kinds of unpacked arguments. The syntax is very similar, but is **used when a function is called, not when it's defined.**

```python
def takes_many_args(*args):
  for arg in args:
    print(arg)

long_list_of_args = [145, "Mexico City", 10.9, "85C"]

# We can use the asterisk "*" to deconstruct the container.
# This makes it so that instead of a list, a series of four different
# positional arguments are passed to the function
takes_many_args(*long_list_of_args)
# Prints "145,Mexico City,10.9,85C"
```

```
## 145
## Mexico City
## 10.9
## 85C
```

We can use the * when calling the function that takes a series of positional parameters to unwrap a list or a tuple. This makes it easy to provide a sequence of arguments to a function even if that function doesn't take a list as an argument. Similarly we can use ** to destructure a dictionary.

```python
def pour_from_sink(temperature="Warm", flow="Medium"):
  print(temperature)
  print(flow)

# Our function takes two keyword arguments
# If we make a dictionary with their parameter names...
sink_open_kwargs = {
  'temperature': 'Hot',
  'flow': "Slight",
}

# We can destructure them an pass to the function
pour_from_sink(**sink_open_kwargs)
# Equivalent to the following:
# pour_from_sink(temperature="Hot", flow="Slight")
```

```
## Hot
## Slight
```

So we can also use dictionaries and pass them into functions as keyword arguments with that syntax. Notice that our pour_from_sink() function doesn't even accept arbitrary **kwargs. We can use this destructuring syntax even when the function has a specific number of keyword or positional arguments it accepts. We just need to be careful that the object we're destructuring matches the length (and names, if a dictionary) of the signature of the function we're passing it into.

Review

We covered a lot of ground in this lesson! We learned all about how functions can accept different arguments and different styles by which we can pass those arguments in. We covered:

- The default return of a function: None
- How to create default arguments to a function
- How to make sure our default arguments work the way we expect when dealing with lists.
- How to pass keyword arguments to a function
- How to unpack multiple returns from a function

- How to unpack multiple positional arguments to a function
- How to unpack multiple keyword arguments to a function
- How to pass a list as a series of arguments to a function
- How to pass a dictionary as a series of keyword arguments to a function

This is a lot, and you should be impressed with yourself! You now should be able to read many different styles of function writing in Python and come up with ways to call those functions with style and clarity.

Hopefully this has helped you as a writer of Python functions and enabled you to overcome any problems with input and output of a Python function you might run into. Congrats!

Files

Reading a File

Computers use file systems to store and retrieve data. Each file is an individual container of related information. If you've ever saved a document, downloaded a song, or even sent an email you've created a file on some computer somewhere. Even script.py, the Python program you're editing in the learning environment, is a file. So, how do we interact with files using Python? We're going to learn how to read and write different kinds of files using code. Let's say we had a file called **real_cool_document.txt** with these contents: *Lorem ipsum!*

We could read that file like this (Notice the file is saved in the **same directory as our .py file**):

```
with open('real_cool_document.txt') as cool_doc:
 cool_contents = cool_doc.read()
 print(cool_contents)
```

```
## Lorem ipsum!
```

This opens a file object called cool_doc and creates a new indented block where you can read the contents of the opened file. We then read the contents of the file cool_doc using cool_doc.read() and save the resulting string into the variable cool_contents. Then we print cool_contents, which outputs the statement *Lorem ipsum!*.

In general we use:

```
#  with open('filename.txt') as file_object:
#    file_string = file_object.read()
```

Iterating Through Lines

When we read a file, we might want to grab the whole document in a single string, like .read() would return. But what if we wanted to store each line in a variable? We can use the .readlines() function to read a text file line by line instead of having the whole thing. Suppose we have a file:

```
"""
keats_sonnet.txt
To one who has been long in city pent,
'Tis very sweet to look into the fair
And open face of heaven,-to breathe a prayer
Full in the smile of the blue firmament.
"""
```

```
## '\nkeats_sonnet.txt\nTo one who has been long in city pent,\n'Tis very sweet to look into the fair\n
```

We import:

```
with open('keats_sonnet.txt') as keats_sonnet:
  for line in keats_sonnet.readlines():
    print(line)
```

```
## To one who has been long in city pent,
##
## 'Tis very sweet to look into the fair
##
## And open face of heaven,-to breathe a prayer
##
## Full in the smile of the blue firmament.
```

The above script creates a temporary file object called keats_sonnet that points to the file keats_sonnet.txt. It then iterates over each line in the document and prints the entire file out.

Reading a Line

Sometimes you don't want to iterate through a whole file. For that, there's a different file method, .readline(), which will only read a single line at a time. If the entire document is read line by line in this way, subsequent calls to .readline() will not throw an error but will start returning an empty string (""). Suppose we had this file:

```
"""
millay_sonnet.txt
I shall forget you presently, my dear,
So make the most of this, your little day,
Your little month, your little half a year,
Ere I forget, or die, or move away,
"""
```

```
## '\nmillay_sonnet.txt\nI shall forget you presently, my dear,\nSo make the most of this, your little
```

```
with open('millay_sonnet.txt') as sonnet_doc:
  first_line = sonnet_doc.readline()
  second_line = sonnet_doc.readline()
  print(second_line)
```

```
## So make the most of this, your little day,
```

This script also creates a file object called sonnet_doc that points to the file millay_sonnet.txt. It then reads in the first line using sonnet_doc.readline() and saves that to the variable first_line. It then saves the second line (So make the most of this, your little day,) into the variable second_line and then prints it out.

Writing a File

Reading a file is all well and good, but what if we want to create a file of our own? With Python we can do just that. It turns out that our open() function that we're using to open a file to read needs another argument to open a file to write to.

```
with open('generated_file_name.txt', 'w') as gen_file:
  gen_file.write("Some content in the file")
```

```
## 24
```

Here we pass the argument 'w' to open() in order to indicate to open the file in write-mode. The default argument is 'r' and passing 'r' to open() opens the file in read-mode as we've been doing.

This code creates a new file in the same folder as script.py and gives it the text What an incredible file!. It's important to note that **if there is already a file called generated_file.txt it will completely overwrite that file**, erasing whatever its contents were before.

Appending to a File

So maybe completely deleting and overwriting existing files is something that bothers you. Isn't there a way to just add a line to a file without completely deleting it? Of course there is! Instead of opening the file using the argument 'w' for write-mode, we open it with 'a' for append-mode. If we have a generated file with the following contents:

```
"""
generated_file.txt
This was a popular file...
"""
```

```
## '\ngenerated_file.txt\nThis was a popular file...\n'
```

Then we can add another line to that file with the following code:

```
with open('generated_file.txt', 'a') as gen_file:
  gen_file.write("... and it still is")
```

```
## 19
```

In the code above we open a file object in the temporary variable gen_file. This variable points to the file generated_file.txt and, since it's open in append-mode, adds the line **"... and it still is"** as a new line to the file. If you were to open the file after running the script it would look like this:

```
"""
generated_file.txt
This was a popular file...
... and it still is
"""
```

```
## '\ngenerated_file.txt\nThis was a popular file...\n... and it still is\n'
```

Notice that opening the file in append-mode, with 'a' as an argument to open(), means that using the file object's .write() method appends whatever is passed to the end of the file in a new line. If we were to run script.py again, this would be what generated_file.txt looks like:

```
"""
generated_file.txt
This was a popular file...
... and it still is
... and it still is
"""
```

```
## '\ngenerated_file.txt\nThis was a popular file...\n... and it still is\n... and it still is\n'
```

Notice that we've appended "... and it still is" to the file a second time! This is because in script.py we opened generated_file.txt in append-mode.

What's With "with"?

We've been opening these files with this **with** block so far, but it seems a little weird that we can only use our file variable in the indented block. Why is that? The **with** keyword invokes something called a context manager for the file that we're calling open() on. This context manager takes care of opening the file when we call open() and then closing the file after we leave the indented block.

Recall:

```
with open('real_cool_document.txt') as cool_doc:
 cool_contents = cool_doc.read()
```

Why is closing the file so complicated? Well, most other aspects of our code deal with things that Python itself controls. All the variables you create: integers, lists, dictionaries — these are all Python objects, and Python knows how to clean them up when it's done with them. Since your files exist outside your Python script, we need to tell Python when we're done with them so that it can close the connection to that file. Leaving a file connection open unnecessarily can affect performance or impact other programs on your computer that might be trying to access that file.

The with syntax replaces older ways (Python2) to access files where you need to call .close() on the file object manually. We can still open up a file and append to it with the old syntax (Python2), as long as we remember to close the file connection afterwards.

Example (Python2):

```
fun_cities_file = open('fun_cities.txt', 'a')

# We can now append a line to "fun_cities".
fun_cities_file.write("\nMontréal")

# But we need to remember to close the file
```

```
## 9
```

```
fun_cities_file.close()
```

In the above script we added "Montréal" as a new line in our file fun_cities.txt. However, since we used the older-style syntax, we had to remember to close the file afterwards. Since this is necessarily more verbose (requires at least one more line of code) without being any more expressive, using with is preferred.

CSV Files

Text files aren't the only thing that Python can read, but they're the only thing that we don't need any additional parsing library to understand.

CSV files are an example of a text file that impose a structure to their data. CSV stands for Comma-Separated Values and CSV files are usually the way that data from spreadsheet software (like Microsoft Excel or Google Sheets) is exported into a portable format. A spreadsheet that looks like the following: In a CSV file that same exact data would be rendered like this:

```
"""
file.csv
Name,Username,Email
Roger Smith,rsmith,wigginsryan@yahoo.com
Michelle Beck,mlbeck,hcosta@hotmail.com
Ashley Barker,a_bark_x,a_bark_x@turner.com
Lynn Gonzales,goodmanjames,lynniegonz@hotmail.com
Jennifer Chase,chasej,jchase@ramirez.com
Charles Hoover,choover,choover89@yahoo.com
Adrian Evans,adevans,adevans98@yahoo.com
Susan Walter,susan82,swilliams@yahoo.com
Stephanie King,stephanieking,sking@morris-tyler.com
Erika Miller,jessica32,ejmiller79@yahoo.com
"""
```

```
## '\nfile.csv\nName,Username,Email\nRoger Smith,rsmith,wigginsryan@yahoo.com\nMichelle Beck,mlbeck,hco
```

Name	Username	Email
Roger Smith	rsmith	wigginsryan@yahoo.com
Michelle Beck	mlbeck	hcosta@hotmail.com
Ashley Barker	a_bark_x	a_bark_x@turner.com
Lynn Gonzales	goodmanjames	lynniegonz@hotmail.com
Jennifer Chase	chasej	jchase@ramirez.com
Charles Hoover	choover	choover89@yahoo.com
Adrian Evans	adevans	adevans98@yahoo.com
Susan Walter	susan82	swilliams@yahoo.com
Stephanie King	stephanieking	sking@morris-tyler.com
Erika Miller	jessica32	ejmiller79@yahoo.com

Notice that the first row of the CSV file doesn't actually represent any data, just the labels of the data that's present in the rest of the file. The rest of the rows of the file are the same as the rows in the spreadsheet software, just instead of being separated into different cells they're separated by… well I suppose it's fair to say they're separated by commas.

Reading a CSV File

Recall our CSV file from our last exercise. Even though we can read these lines as text without a problem, there are ways to access the data in a format better suited for programming purposes. In Python we can convert that data into a dictionary using the csv library's DictReader object. Here's how we'd create a list of the email addresses of all of the users in the above table:

```
import csv

list_of_email_addresses = []
with open('users.csv', newline='') as users_csv:
  user_reader = csv.DictReader(users_csv)
  for row in user_reader:
    list_of_email_addresses.append(row['Email'])
```

In the above code we first import our csv library, which gives us the tools to parse our CSV file. We then create the empty list list_of_email_addresses which we'll later populate with the email addresses from our CSV. Then we open the users.csv file with the temporary variable users_csv.

We pass the additional keyword argument newline=" to the file opening open() function so that we don't accidentally mistake a line break in one of our data fields as a new row in our CSV (read more about this in the Python documentation).

After opening our new CSV file we use csv.DictReader(users_csv) which converts the lines of our CSV file to Python dictionaries which we can use access methods for. The keys of the dictionary are, by default, the entries in the first line of our CSV file. Since our CSV's first line calls the third field in our CSV "Email", we can use that as the key in each row of our DictReader.

When we iterate through the rows of our user_reader object, we access all of the rows in our CSV as dictionaries (except for the first row, which we used to label the keys of our dictionary). By accessing the 'Email' key of each of these rows we can grab the email address in that row and append it to our list_of_email_addresses.

```
print(list_of_email_addresses)
```

```
## ['wigginsryan@yahoo.com', 'hcosta@hotmail.com', 'a_bark_x@turner.com', 'lynniegonz@hotmail.com', 'jc
```

Notice you can use for…in syntax to loop through a DictReader object. Each of these objects will be a dictionary, with the first row of the file indicating the keys.

```
#  for row in csv_file_dict:
#    print(row['Key'])
```

Reading Different Types of CSV Files

We've been acting like CSV files are Comma-Separated Values files. It's true that CSV stands for that, but it's also true that other ways of separating values are valid CSV files these days.

People used to call Tab-Separated Values files TSV files, but as other separators grew in popularity everyone realized that creating a new .[a-z]sv file format for every value-separating character used is not sustainable. So we call all files with a list of different values a CSV file and then use different **delimiters (like a comma or tab)** to indicate where the different values start and stop.

Let's say we had an address book. Since addresses usually use commas in them, we'll need to use a different delimiter for our information. Since none of our data has semicolons (;) in them, we can use those.

Example:

```
"""
addresses.csv
Name;Address;Telephone
Donna Smith;126 Orr Corner Suite 857\nEast Michael, LA 54411;906-918-6560
Aaron Osborn;6965 Miller Station Suite 485\nNorth Michelle, KS 64364;815.039.3661x42816
Jennifer Barnett;8749 Alicia Vista Apt. 288\nLake Victoriaberg, TN 51094;397-796-4842x451
Joshua Bryan;20116 Stephanie Stravenue\nWhitneytown, IA 87358;(380)074-6173
Andrea Jones;558 Melissa Keys Apt. 588\nNorth Teresahaven, WA 63411;+57(8)7795396386
Victor Williams;725 Gloria Views Suite 628\nEast Scott, IN 38095;768.708.3411x954
"""
```

```
## '\naddresses.csv\nName;Address;Telephone\nDonna Smith;126 Orr Corner Suite 857\nEast Michael, LA 544
```

Notice the \n character, this is the escape sequence for a new line. The possibility of a new line escaped by a \n character in our data is why we pass the newline=” keyword argument to the open() function.

Also notice that many of these addresses have commas in them! This is okay, we'll still be able to read it. If we wanted to, say, print out all the addresses in this CSV file we could do the following:

```
import csv

with open('addresses.csv', newline='') as addresses_csv:
  address_reader = csv.DictReader(addresses_csv, delimiter=';')
  for row in address_reader:
    print(row['Address'])
```

```
## 126 Orr Corner Suite 857\nEast Michael, LA 54411
## 6965 Miller Station Suite 485\nNorth Michelle, KS 64364
## 8749 Alicia Vista Apt. 288\nLake Victoriaberg, TN 51094
## 20116 Stephanie Stravenue\nWhitneytown, IA 87358
## 558 Melissa Keys Apt. 588\nNorth Teresahaven, WA 63411
## 725 Gloria Views Suite 628\nEast Scott, IN 38095
```

Notice that when we call csv.DictReader we pass in the delimiter parameter, which is the string that's used to delineate separate fields in the CSV. We then iterate through the CSV and print out each of the addresses.

Writing a CSV File

Naturally if we have the ability to read different CSV files we might want to be able to programmatically create CSV files that save output and data that someone could load into their spreadsheet software. Let's say we have a big list of data that we want to save into a CSV file. We could do the following:

```
big_list = [{'name': 'Fredrick Stein', 'userid': 6712359021, 'is_admin': False}, {'name': 'Wiltmore Deni
for item in big_list:
    print(item)
```

```
## {'name': 'Fredrick Stein', 'userid': 6712359021, 'is_admin': False}
## {'name': 'Wiltmore Denis', 'userid': 2525942, 'is_admin': False}
## {'name': 'Greely Plonk', 'userid': 15890235, 'is_admin': False}
## {'name': 'Dendris Stulo', 'userid': 572189563, 'is_admin': True}
```

```
import csv

with open('output.csv', 'w') as output_csv:
  fields = ['name', 'userid', 'is_admin']
  output_writer = csv.DictWriter(output_csv, fieldnames=fields)

  output_writer.writeheader()
  for item in big_list:
    output_writer.writerow(item)
```

```
## 22
## 33
## 30
## 29
## 30
```

In our code above we had a set of dictionaries with the same keys for each, a prime candidate for a CSV. We import the csv library, and then open a new CSV file in write-mode by passing the 'w' argument to the open() function.

We then define the fields we're going to be using into a variable called fields. We then instantiate our CSV writer object and pass two arguments. The first is output_csv, the file handler object. The second is our list of fields fields which we pass to the keyword parameter fieldnames.

Now that we've instantiated our CSV file writer, we can start adding lines to the file itself! First we want the headers, so we call .writeheader() on the writer object. This writes all the fields passed to fieldnames as the first row in our file. Then we iterate through our big_list of data. Each item in big_list is a dictionary with each field in fields as the keys. We call output_writer.writerow() with the item dictionaries which writes each line to the CSV file.

Another example:

We have a list in the workspace access_log which is a list of dictionaries we want to write out to a CSV file.

```
access_log = [{'time': '08:39:37', 'limit': 844404, 'address': '1.227.124.181'}, {'time': '13:13:35', 'l
fields = ['time', 'address', 'limit']
```

```
# We first import csv module
import csv
# Open up the file logger.csv in the temporary variable logger_csv in write mode
with open('logger.csv','w') as logger_csv:
  #Create a csv.DictWriter instance called log_writer
  log_writer=csv.DictWriter(logger_csv,fields)
  # Write the header to log_writer using the .writeheader() method.
  log_writer.writeheader()
  # Iterate through the access_log list and add each element to the CSV
  for item in access_log:
    log_writer.writerow(item)
```

```
## 20
## 31
## 32
## 29
## 34
## 30
## 32
## 32
## 33
## 27
## 29
```

Reading a JSON File

CSV isn't the only file format that Python has a built-in library for. We can also use Python's file tools to read and write JSON. JSON, an abbreviation of JavaScript Object Notation, is a file format inspired by the programming language JavaScript. The name, like CSV is a bit of a misnomer — some JSON is not valid JavaScript (and plenty of JavaScript is not valid JSON).

JSON's format is endearingly similar to Python dictionary syntax, and so JSON files might be easy to read from a Python developer standpoint. Nonetheless, Python comes with a json package that will help us parse JSON files into actual Python dictionaries. Suppose we have a JSON file like the following:

```
"""
purchase_14781239.json
{
  'user': 'ellen_greg',
  'action': 'purchase',
  'item_id': '14781239',
}
"""
```

```
## "\npurchase_14781239.json\n{\n  'user': 'ellen_greg',\n  'action': 'purchase',\n  'item_id': '1478123
```

We would be able to read that in as a Python dictionary with the following code:

```
import json

with open("purchase_14781239.json") as purchase_json:
  purchase_data = json.load(purchase_json)

print(purchase_data['user'])
  # Prints 'ellen_greg'
```

```
## ellen_greg
```

First we import the json package. We opened the file using our trusty open() command. Since we're opening it in read-mode we just need to pass the file name. We save the file in the temporary variable purchase_json.

We continue by parsing purchase_json using json.load(), creating a Python dictionary out of the file. Saving the results into purchase_data means we can interact with it. We print out one of the values of the JSON file by keying into the purchase_data object.

Writing a JSON File

Naturally we can use the json library to translate Python objects to JSON as well. This is especially useful in instances where you're using a Python library to serve web pages, you would also be able to serve JSON. Let's say we had a Python dictionary we wanted to save as a JSON file:

```
turn_to_json = {
  'eventId': 674189,
  'dateTime': '2015-02-12T09:23:17.511Z',
  'chocolate': 'Semi-sweet Dark',
  'isTomatoAFruit': True
}
```

We'd be able to create a JSON file with that information by doing the following:

```
import json

with open('output.json', 'w') as json_file:
  json.dump(turn_to_json, json_file)
```

We import the json module, open up a write-mode file under the variable json_file, and then use the json.dump() method to write to the file. json.dump() takes two arguments: first the data object, then the file object you want to save.

Review

Now you know all about files!

- Open up file objects using open() and with.
- Read a file's full contents using Python's .read() method.
- Read a file line-by-line using .readline() and .readlines()
- Create new files by opening them in write-mode.
- Append to a file non-destructively by opening a file in append-mode.
- Apply all of the above to different types of data-carrying files including CSV and JSON!

Clases

If you have arrived so far, I must give you a disclaimer for the rest of this course. This section and the following cover more complex areas of programming and require a basic understanging of computer science. Myself, as an economist, found really hard to go through these subjects and I'll do my best to give a basic explanaition. Although I honestly recommend to look for supporting material if this is not clear enough.

Types

Python equips us with many different ways to store data. A float is a different kind of number from an int, and we store different data in a list than we do in a dict. These are known as different types. We can check the type of a Python variable using the type() function.

```
a_string = "Cool String"
an_int = 12

print(type(a_string))
# prints "<class 'str'>"
```

```
## <class 'str'>
```

```
print(type(an_int))
# prints "<class 'int'>"
```

```
## <class 'int'>
```

Above, we defined two variables, and checked the type of these two variables. A variable's type determines what you can do with it and how you can use it. You can't .get() something from an integer, just as you can't add two dictionaries together using +. This is because those **operations are defined at the type level.**

```
print(type(5))
```

```
## <class 'int'>
```

```
my_dict={}
print(type(my_dict))
```

```
## <class 'dict'>
```

```
my_list=[]
print(type(my_list))
```

```
## <class 'list'>
```

Class

A class is a template for a data type. It describes the kinds of information that class will hold and how a programmer will interact with that data. Define a class using the class keyword. PEP 8 Style Guide for Python Code recommends capitalizing the names of classes to make them easier to identify.

```
class CoolClass:
  pass
```

In the above example we created a class and named it CoolClass. We used the **pass** keyword in Python to indicate that the body of the class was intentionally left blank so we don't cause an IndentationError. We'll learn about all the things we can put in the body of a class in the next few exercises.

Instantiation

A class doesn't accomplish anything simply by being defined. A class must be instantiated. In other words, we must create an instance of the class, in order to breathe life into the schematic.

Instantiating a class looks a lot like calling a function. We would be able to create an instance of our defined CoolClass as follows:

```
cool_instance = CoolClass()
```

Above, we created an object by adding parentheses to the name of the class. We then assigned that new instance to the variable cool_instance for safe-keeping so we can access our instance of CoolClass at a later time.

Object-Oriented Programming

A class instance is also called an object. The pattern of defining classes and creating objects to represent the responsibilities of a program is known as Object Oriented Programming or OOP.

Instantiation takes a class and turns it into an object, the **type()** function does the opposite of that. When called with an object, it returns the class that the object is an instance of.

```
print(type(cool_instance))
# prints "<class '__main__.CoolClass'>"
```

```
## <class '__main__.CoolClass'>
```

We then print out the type() of cool_instance and it shows us that this object is of type __main__.CoolClass.

In Python __main__ means "this current file that we're running" and so one could read the output from type() to mean "the class CoolClass that was defined here, in the script you're currently running."

Class Variables

When we want the same data to be available to every instance of a class we use a class variable. A class variable is a variable that's the same for every instance of the class.

You can define a class variable by including it in the indented part of your class definition, and you can access all of an object's class variables with **object.variable** syntax.

```
class Musician:
  title = "Rockstar"

drummer = Musician()
print(drummer.title)
# prints "Rockstar"
```

```
## Rockstar
```

Above we defined the class Musician, then instantiated drummer to be an object of type Musician. We then printed out the drummer's .title attribute, which is a class variable that we defined as the string "Rockstar".

If we defined another musician, like guitarist = Musician() they would have the same .title attribute.

```
guitarist = Musician()
print(guitarist.title)
```

```
## Rockstar
```

Example:

You are digitizing grades for Jan van Eyck High School and Conservatory. At Jan van High, as the students call it, 65 is the minimum passing grade. We create a Grade class with a class attribute minimum_passing equal to 65.

```
class Grade:
  minimum_passing=65
```

Methods

Methods are functions that are defined as part of a class. The first argument in a method is always the object that is calling the method. Convention recommends that we name this first argument self. Methods always have at least this one argument.

We define methods similarly to functions, except that they are indented to be part of the class.

```
class Dog:
  dog_time_dilation = 7

  def time_explanation(self):
    print("Dogs experience {} years for every 1 human year.".format(self.dog_time_dilation))

pitbull = Dog()
pitbull.time_explanation()
# Prints "Pitbull experience 7 years for every 1 human year."
```

```
## Dogs experience 7 years for every 1 human year.
```

Above we created a Dog class with a time_explanation method that takes one argument, self, which refers to the object calling the function. We created a Dog named pitbull and called the .time_explanation() method on our new object for Pitbull

Notice we didn't pass any arguments when we called .time_explanation(), but were able to refer to self in the function body. When you call a method it automatically passes the object calling the method as the first argument.

Methods with Arguments

Methods can also take more arguments than just self:

```
class DistanceConverter:
  kms_in_a_mile = 1.609
  def how_many_kms(self, miles):
    return miles * self.kms_in_a_mile

converter = DistanceConverter()
kms_in_5_miles = converter.how_many_kms(5)
print(kms_in_5_miles)
# prints "8.045"
```

```
## 8.045
```

Above we defined a DistanceConverter class, instantiated it, and used it to convert 5 miles into kilometers. Notice again that even though how__many__kms takes two arguments in its definition, we only pass miles, because self is implicitly passed (and refers to the object converter).

Another example:

It's March 14th (known in some places as Pi day), and you're feeling awfully festive. You decide to create a program that calculates the area of a circle.

```
class Circle:
  pi=3.14
  def area(self,radius):
    #Since pi is a class variable, you can access it as an attribute of the class.
    return Circle.pi * radius ** 2
#Create an instance of Circle in var circle
circle=Circle()

# You go to measure several circles you happen to find around. A medium pizza that is 12 inches across.
pizza_area = circle.area(12/2)
teaching_table_area = circle.area(36/2)
round_room_area = circle.area(11460/2)

print(pizza_area)
```

```
## 113.04
```

```
print(teaching_table_area)
```

```
## 1017.36
```

```
print(round_room_area)
```

```
## 103095306.0
```

Constructors

There are several methods that we can define in a Python class that have special behavior. These methods are sometimes called **"magic"**, because they behave differently from regular methods. Another popular term is **dunder methods**, so-named because they have two underscores (double-underscore abbreviated to "dunder") on either side of them.

The first dunder method we're going to use is the **__init__** method (note the two underscores before and after the word "init"). This method is used to initialize a newly created object. It is called every time the class is instantiated.

Methods that are used to prepare an object being instantiated are called **constructors**. The word "constructor" is used to describe similar features in other object-oriented programming languages but programmers who refer to a constructor in Python are usually talking about the __init__ method.

```
class Shouter:
  def __init__(self):
    print("HELLO?!")

shout1 = Shouter()
# prints "HELLO?!"
```

```
## HELLO?!
```

```
shout2 = Shouter()
# prints "HELLO?!"
```

```
## HELLO?!
```

Above we created a class called Shouter and every time we create an instance of Shouter the program prints out a shout. Don't worry, this doesn't hurt the computer at all.

Pay careful attention to the instantiation syntax we use. Shouter() looks a lot like a function call, doesn't it? If it's a function, can we pass parameters to it? We absolutely can, and those parameters will be received by the __init__ method.

```
class Shouter:
  def __init__(self, phrase):
    # make sure phrase is a string
    if type(phrase) == str:
      # then shout it out
      print(phrase.upper())

shout1 = Shouter(2)
# prints "SHOUT"
shout2 = Shouter("shout")
# prints "SHOUT"
```

```
## SHOUT
```

```
shout3 = Shouter("let it all out")
# prints "LET IT ALL OUT"
```

```
## LET IT ALL OUT
```

Above we've updated our Shouter class to take the additional parameter phrase. When we created each of our objects we passed an argument to the constructor. The constructor takes the argument phrase and, if it's a string, prints out the all-caps version of phrase.

Back to our circle area example: Since we seem more frequently to know the diameter of a circle, it should take the argument diameter. We add a constructor:

```
class Circle:
  pi = 3.14
  # Add constructor here:
  def __init__(self, diameter):
    print("New circle with diameter: {diameter}".format(diameter=diameter))

teaching_table=Circle(36)
```

```
## New circle with diameter: 36
```

```
print(teaching_table)
```

```
## <__main__.Circle object at 0x7ff521543460>
```

```
teach_table_area = circle.area(36/2)
#Notice we still can use our .area method
print("The teaching table area is "+ str(teach_table_area))
```

```
## The teaching table area is 1017.36
```

Instance Variables

We've learned so far that a class is a schematic for a data type and an object is an instance of a class, but why is there such a strong need to differentiate the two if each object can only have the methods and class variables the class has? This is because each instance of a class can hold different kinds of data.

The data held by an object is referred to as an instance variable. Instance variables aren't shared by all instances of a class — they are variables that are specific to the object they are attached to.

Let's say that we have the following class definition:

```
class FakeDict:
  pass
```

We can instantiate two different objects from this class, fake_dict1 and fake_dict2, and assign instance variables to these objects using the same attribute notation that was used for accessing class variables.

```
fake_dict1 = FakeDict()
fake_dict2 = FakeDict()

fake_dict1.fake_key = "This works!"
fake_dict2.fake_key = "This too!"

# Let's join the two strings together!
working_string = "{} {}".format(fake_dict1.fake_key, fake_dict2.fake_key)
print(working_string)
# prints "This works! This too!"
```

```
## This works! This too!
```

Example:

```
#Let’s say that we have the following class definition:
class Store:
  pass

# We create two objects from this store class:
alternative_rocks = Store()
isabelles_ices = Store()
```

```
# We give them both instance attributes called store_name
alternative_rocks.store_name = "Alternative Rocks"
isabelles_ices.store_name = "Isabelle's Ices"
```

Attribute Functions

Instance variables and class variables are both accessed similarly in Python. This is no mistake, they are both considered attributes of an object. If we attempt to access an attribute that is neither a class variable nor an instance variable of the object Python will throw an *AttributeError.*

Example:

```
class NoCustomAttributes:
  pass

attributeless = NoCustomAttributes()
try:
  attributeless.fake_attribute
except AttributeError:
  print("You prodiced an Attribute Error!")

# prints "This text gets printed!"
```

```
## You prodiced an Attribute Error!
```

What if we aren't sure if an object has an attribute or not? **getattr()** is a Python function that works a lot like the usual dot-syntax (i.e., object_name.attribute_name) but we can supply a third argument that will be the default if the object does not have the given attribute.

What if we only really care whether the attribute exists? **hasattr()** will return True if an object has a given attribute and False otherwise.

```
hasattr(attributeless, "fake_attribute")
# returns False
```

```
## False
```

```
getattr(attributeless, "other_fake_attribute", 800)
# returns 800, the default value
```

```
## 800
```

Above we checked if the *attributeless* object has the attribute *fake_attribute.* Since it does not, hasattr() returned False. After that, we used getattr to attempt to retrieve other_fake_attribute. Since other_fake_attribute isn't a real attribute on attributeless, our call to getattr() returned the supplied default value 800, instead of throwing an AttributeError.

Example:

We have a list of different data types, some strings, some lists, and some dictionaries, all saved in the variable how_many_s.

For every element in the list, check if the element has the attribute *.count.* If so, count the number of times the string "s" appears in the element. Print this number.

```
how_many_s = [{'s': False}, "sassafrass", 18, ["a", "c", "s", "d", "s"]]

for element in how_many_s:
  if hasattr(element, "count"):
    print(element.count("s"))
```

```
## 5
## 2
```

Self

Since we can already use dictionaries to store key-value pairs, using objects for that purpose is not really useful. Instance variables are more powerful when you can guarantee a rigidity to the data the object is holding.

This convenience is most apparent when the constructor creates the instance variables, using the arguments passed in to it. If we were creating a search engine, and we wanted to create classes for each separate entry we could return. We'd do that like this:

```python
class SearchEngineEntry:
  def __init__(self, url):
    self.url = url

codecademy = SearchEngineEntry("www.codecademy.com")
wikipedia = SearchEngineEntry("www.wikipedia.org")

print(codecademy.url)
# prints "www.codecademy.com"
```

```
## www.codecademy.com
```

```python
print(wikipedia.url)
# prints "www.wikipedia.org"
```

```
## www.wikipedia.org
```

Since the self keyword refers to the object and not the class being called, we can define a secure method on the SearchEngineEntry class that returns the secure link to an entry.

```python
class SearchEngineEntry:
  secure_prefix = "https://"
  def __init__(self, url):
    self.url = url

  def secure(self):
    return "{prefix}{site}".format(prefix=self.secure_prefix, site=self.url)

codecademy = SearchEngineEntry("www.codecademy.com")
wikipedia = SearchEngineEntry("www.wikipedia.org")

print(codecademy.url)
# prints "https://www.codecademy.com"
```

```
## www.codecademy.com
```

```python
print(wikipedia.secure())
# prints "https://www.wikipedia.org"
```

```
## https://www.wikipedia.org
```

Above we define our secure() method to take just the one required argument, self. We access both the class variable self.secure_prefix and the instance variable self.url to return a secure URL. This is the strength of writing object-oriented programs. We can write our classes to structure the data that we need and write methods that will interact with that data in a meaningful way.

Example:

Recal the Circle class.Even though we usually know the diameter beforehand, what we need for most calculations is the radius.

```
class Circle:
  pi = 3.14
  def __init__(self, diameter):
    print("Creating circle with diameter {d}".format(d=diameter))
    #In Circle‘s constructor we set the instance variable
    #self.radius to equal half the diameter that gets passed in.
    self.radius = diameter / 2
  # We efine a new method circumference for your circle object that takes only one argument, self, and
  def circumference(self):
    return 2 * self.pi * self.radius

#We define three Circles with three different diameters.
medium_pizza = Circle(12)
```

```
## Creating circle with diameter 12
```

```
teaching_table = Circle(36)
```

```
## Creating circle with diameter 36
```

```
round_room = Circle(11460)
```

```
## Creating circle with diameter 11460
```

```
print('The circle circumference is '+str(medium_pizza.circumference()))
```

```
## The circle circumference is 37.68
```

```
print('The circle circumference is '+str(teaching_table.circumference()))
```

```
## The circle circumference is 113.04
```

```
print('The circle circumference is '+str(round_room.circumference()))
```

```
## The circle circumference is 35984.4
```

Everything is an Object

Attributes can be added to user-defined objects after instantiation, so it’s possible for an object to have some attributes that are not explicitly defined in an object’s constructor. We can use the dir() function to investigate an object’s attributes at runtime. dir() is short for directory and offers an organized presentation of object attributes.

```
class FakeDict:
  pass

fake_dict = FakeDict()
fake_dict.attribute = "Cool"

dir(fake_dict)
```

```
## ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__
```

That’s certainly a lot more attributes than we defined! Python automatically adds a number of attributes to all objects that get created. These internal attributes are usually indicated by double-underscores. But sure enough, attribute is in that list (at the end).

Do you remember being able to use type() on Python's native data types? This is because they are also objects in Python. Their classes are int, float, str, list, and dict. These Python classes have special syntax for their instantiation, 1, 1.0, "hello", [], and {} specifically. But these instances are still full-blown objects to Python.

```
fun_list = [10, "string", {'abc': True}]

type(fun_list)
# Prints <class 'list'>
```

```
## <class 'list'>
```

```
dir(fun_list)
# Prints ['__add__', '__class__', [...], 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert
```

```
## ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__
```

Above we define a new list. We check it's type and see that's an instantiation of class list. We use dir() to explore its attributes, and it gives us a large number of internal Python dunder attributes, but, afterward, we get the usual list methods.

Even Functions are objects too!

```
def this_function_is_an_object(anything):
  print ("Anything")

dir(this_function_is_an_object)
```

```
## ['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__
```

String Representation

One of the first things we learn as programmers is how to print out information that we need for debugging. Unfortunately, when we print out an object we get a default representation that seems fairly useless.

```
class Employee():
  def __init__(self, name):
    self.name = name

argus = Employee("Argus Filch")
print(argus)
# prints "<__main__.Employee object at 0x104e88390>"
```

```
## <__main__.Employee object at 0x7ff521543880>
```

This default string representation gives us some information, like where the class is defined and our computer's memory address where this object is stored, but is usually not useful information to have when we are trying to debug our code.

We learned about the dunder method ___init___. Now, we will learn another dunder method called **___repr___**. This is a method we can use to tell Python what we want the string representation of the class to be. ___repr___ can only have one parameter, self, and must return a string.

In our Employee class above, we have an instance variable called name that should be unique enough to be useful when we're printing out an instance of the Employee class.

```
class Employee():
  def __init__(self, name):
    self.name = name
  def __repr__(self):
    return self.name

argus = Employee("Argus Filch")
print(argus)
# prints "Argus Filch"
```

```
## Argus Filch
```

We implemented the ___repr___ method and had it return the .name attribute of the object. When we printed the object out it simply printed the .name of the object! Cool!

Example: Adding the ___repr___ method to the Circle class

```
class Circle:
  pi = 3.14

  def __init__(self, diameter):
    self.radius = diameter / 2

  def __repr__(self):
    return "Circle with radius {}".format(self.radius)

  def area(self):
    return self.pi * self.radius ** 2

  def circumference(self):
    return self.pi * 2 * self.radius


medium_pizza = Circle(12)
teaching_table = Circle(36)
round_room = Circle(11460)

print(medium_pizza)
```

```
## Circle with radius 6.0
```

```
print(teaching_table)
```

```
## Circle with radius 18.0
```

```
print(round_room)
```

```
## Circle with radius 5730.0
```

Review

So far we've covered what a data type actually is in Python. We explored what the functionality of Python's built-in types (also referred to as primitives) are. We learned how to create our own data types using the class keyword.

We explored the relationship between a class and an object — we create objects when we instantiate a class, we find the class when we check the type() of an object. We learned the difference between class variables (the same for all objects of a class) and instance variables (unique for each object).

We learned about how to define an object's functionality with methods. We created multiple objects from the same class, all with similar functionality, but with different internal data. They all had the same methods, but produced different output because they were different instances.

Take a moment to congratulate yourself, object-oriented programming is a complicated concept.

Example (follow the following code by steps):

```
# 1 We define a class Student this will be our data model
# 2 Add a constructor for Student. Have the constructor take in two parameters: a name and a year. Save

class Student:
  def __init__(self, name, year):
    self.name = name
    self.year = year
    #Step 6
    self.grades = []
  #Step 7
  def add_grade(self,grade):
    if type(grade) is Grade:
      self.grades.append(grade)
  #Extra
  def __repr__(self):
    return self.name

# 4 Create a Grade class, with minimum_passing as an attribute set to 65.
# 5 Give Grade a constructor. Take in a parameter score and assign it to self.score.

class Grade:
  minimum_passing=65
  def __init__(self, score):
    self.score = score

# 3 Create three instances of the Student class. Remember: Create an instance by passing arguments to t
roger = Student('Roger van der Weyden',10)
sandro = Student('Sandro Botticelli',12)
pieter = Student('Pieter Bruegel the Elder', 8)
#Step 8
pieter.add_grade(Grade(100))

# 6 In the body of the constructor for Student, declare self.grades as an empty list.
# 7 Add an .add_grade() method to Student that takes a parameter, grade.
# .add_grade() should verify that grade is of type Grade and if so, add it to the Student‘s .grades.
# If grade isn’t an instance of Grade then .add_grade() should do nothing.
# 8 Create a new Grade with a score of 100 and add it to pieter‘s .grades attribute using .add_grade().

print(pieter)
```

```
## Pieter Bruegel the Elder
```

You’ve created two classes and defined their interactions. This is object-oriented programming! From here you could:

- Write a Grade method .is_passing() that returns whether a Grade has a passing .score.
- Write a Student method get_average() that returns the student’s average score.
- Add an instance variable to Student that is a dictionary called .attendance, with dates as keys and booleans as values that indicate whether the student attended school that day.
- Write your own classes to do whatever logic you want!

Inheritance and polymorphism.

Inheritance

Classes are designed to allow for more code reuse, but what if we need a class that looks a lot like a class we already have? If the bulk of a class's definition is useful, but we have a new use case that is distinct from how the original class was used, we can inherit from the original class. Think of inheritance as a remix — it sounds a lot like the original, but there's something… different about it.

```
class User:
  is_admin = False
  def __init__(self, username):
    self.username = username

class Admin(User):
  is_admin = True
```

Above we defined User as our base class. We want to create a new class that inherits from it, so we created the subclass Admin. In the above example, Admin has the same constructor as User. Only the class variable is_admin is set differently between the two.

Sometimes a base class is called a parent class. In these terms, the class inheriting from it, the subclass, is also referred to as a child class.

In general

```
"""
class Subclass(Superclass):
  pass
"""
```

```
## '\nclass Subclass(Superclass):\n  pass\n'
```

Exceptions

There's one very important family of class definitions built in to the Python language. An Exception is a class that inherits from **Python's Exception class**.

We can validate this ourselves using the **issubclass() function**. issubclass() is a Python built-in function that takes two parameters. issubclass() returns True if the first argument is a subclass of the second argument. It returns False if the first class is not a subclass of the second. issubclass() raises a TypeError if either argument passed in is not a class.

```
issubclass(ZeroDivisionError, Exception)
# Returns True
```

```
## True
```

Above, we checked whether ZeroDivisionError, the exception raised when attempting division by zero, is a subclass of Exception. It is, so issubclass returns True.

Why is it beneficial for exceptions to inherit from one another? Let's consider an example where we create our own exceptions. What if we were creating software that tracks our kitchen appliances? We would be able to design a suite of exceptions for that need:

```
class KitchenException(Exception):
  """
  Exception that gets thrown when a kitchen appliance isn't working
  """
class MicrowaveException(KitchenException):
  """
  Exception for when the microwave stops working
  """
class RefrigeratorException(KitchenException):
  """
  Exception for when the refrigerator stops working
  """
```

In this code, we define three exceptions. First, we define a KitchenException that acts as the parent to our other, specific kitchen appliance exceptions. KitchenException subclasses Exception, so it behaves in the same way that regular Exceptions do. Afterward we define MicrowaveException and RefrigeratorException as subclasses.

Since our exceptions are subclassed in this way, we can catch any of KitchenException‘s subclasses by catching KitchenException. For example:

```
#  def get_food_from_fridge():
#    if refrigerator.cooling == False:
#      raise RefrigeratorException
#    else:
#      return food
#
#  def heat_food(food):
#    if microwave.working == False:
#      raise MicrowaveException
#    else:
#      microwave.cook(food)
#      return food
#
#  try:
#    food = get_food_from_fridge()
#    food = heat_food(food)
#  except KitchenException:
#    food = order_takeout()
```

In the above example, we attempt to retrieve food from the fridge and heat it in the microwave. If either RefrigeratorException or MicrowaveException is raised, we opt to order takeout instead. We catch both RefrigeratorException and MicrowaveException in our try/except block because both are subclasses of KitchenException.

Explore Python’s exception hierarchy in the Python documentation!

Another example:

We’ve defined a CandleShop class for our new candle shop that we’ve named Here’s a Hot Tip: Buy Drip Candles. We want to define our own exceptions for when we run out of candles to sell. We define our own exception called OutOfStock that inherits from the Exception class and have CandleShop raise your OutOfStock exception when CandleShop.buy() tries to buy a candle that’s out of stock.

```
#class OutOfStock(Exception):
#  pass

#class CandleShop:
#  name = "Here's a Hot Tip: Buy Drip Candles"
#  def __init__(self, stock):
#    self.stock = stock
#  def buy(self, color):
#    if self.stock[color] < 1:
#      raise OutOfStock
#    self.stock[color] = self.stock[color] - 1

#candle_shop = CandleShop({'blue': 6, 'red': 2, 'green': 0})
#candle_shop.buy('blue')

# This should raise OutOfStock:
#candle_shop.buy('green')
```

Overriding Methods

Inheritance is a useful way of creating objects with different class variables, but is that all it's good for? What if one of the methods needs to be implemented differently? In Python, all we have to do to override a method definition is to offer a new definition for the method in our subclass.

An overridden method is one that has a different definition from its parent class. What if User class didn't have an is_admin flag but just checked if it had permission for something based on a permissions dictionary? It could look like this:

```python
class User:
  def __init__(self, username, permissions):
    self.username = username
    self.permissions = permissions

  def has_permission_for(self, key):
    if self.permissions.get(key):
      return True
    else:
      return False
```

Above we defined a class User which takes a permissions parameter in its constructor. Let's assume permissions is a dict. User has a method .has_permission_for() implemented, where it checks to see if a given key is in its permissions dictionary. We could then define our Admin user like this:

```python
class Admin(User):
  def has_permission_for(self, key):
    return True
```

Here we define an Admin class that subclasses User. It has all methods, attributes, and functionality that User has. However, if you call has_permission_for on an instance of Admin, it won't check its permissions dictionary. Since this User is also an Admin, we just say they have permission to see everything!

Example:

In script.py, we've defined two classes, Message and User. We create an Admin class that subclasses the User class.

```python
class Message:
  def __init__(self, sender, recipient, text):
    self.sender = sender
    self.recipient = recipient
    self.text = text

class User:
  def __init__(self, username):
    self.username = username

  def edit_message(self, message, new_text):
    if message.sender == self.username:
      message.text = new_text

# class Admin(User):
#  pass
```

We override User's .edit_message() method in Admin so that an Admin can edit any messages.

```
class Message:
  def __init__(self, sender, recipient, text):
    self.sender = sender
    self.recipient = recipient
    self.text = text

class User:
  def __init__(self, username):
    self.username = username

  def edit_message(self, message, new_text):
    if message.sender == self.username:
      message.text = new_text

class Admin(User):
  def edit_message(self, message, new_text):
    message.text = new_text
```

Super()

Overriding methods is really useful in some cases but sometimes we want to add some extra logic to the existing method. In order to do that we need a way to call the method from the parent class. Python gives us a way to do that using super().

super() gives us a **proxy object**. With this proxy object, we can invoke the method of an object's parent class (also called its superclass). We call the required function as a method on super():

```
class Sink:
  def __init__(self, basin, nozzle):
    self.basin = basin
    self.nozzle = nozzle

class KitchenSink(Sink):
  def __init__(self, basin, nozzle, trash_compactor=None):
    super().__init__(basin, nozzle)
    if trash_compactor:
      self.trash_compactor = trash_compactor
```

Above we defined two classes. First, we defined a Sink class with a constructor that assigns a rinse basin and a sink nozzle to a Sink instance. Afterwards, we defined a KitchenSink class that inherits from Sink. KitchenSink's constructor takes an additional parameter, a trash_compactor. KitchenSink then calls the constructor for Sink with the basin and nozzle it received using the super() function, with this line of code: **super().___init___(basin, nozzle)**.

This line says: "call the constructor (the function called ___init___) of the class that is this class's parent class." In the example given, KitchenSink's constructor calls the constructor for Sink. In this way, we can override a parent class's method to add some new functionality (like adding a trash_compactor to a sink), while still retaining the behavior of the original constructor (like setting the basin and nozzle as instance variables).

Example:

You're invited to a potluck this week and decide to make your own special version of Potato Salad! You'll find a class called PotatoSalad, we make a subclass of PotatoSalad called SpecialPotatoSalad.

```python
class PotatoSalad:
  def __init__(self, potatoes, celery, onions):
    self.potatoes = potatoes
    self.celery = celery
    self.onions = onions

# Your special potato salad recipe is pretty similar to a regular potato salad, so let's start with mak

# We create a new constructor for SpecialPotatoSalad that just calls the parent constructor for PotatoS

# The difference with your special potato salad is... you add raisins to it. You always use the full pa

class SpecialPotatoSalad(PotatoSalad):
  def __init__(self, potatoes, celery, onions):
    super().__init__(potatoes, celery, onions)
    self.raisins = 40
```

Interfaces

You may be wondering at this point why we would even want to have two different classes with two differently implemented methods to use the same method name. This style is especially useful when we have an object for which it might not matter which class the object is an instance of. Instead, we're interested in whether that object can perform a given task.

If we have the following code:

```python
#  class Chess:
#    def __init__(self):
#      self.board = setup_board()
#      self.pieces = add_chess_pieces()
#
#    def play(self):
#      print("Playing chess!")
#
#  class Checkers:
#    def __init__(self):
#      self.board = setup_board()
#      self.pieces = add_checkers_pieces()
#
#    def play(self):
#      print("Playing checkers!")
```

In the code above we define two classes, Chess and Checkers. In Chess we define a constructor that sets up the board and pieces, and a .play() method. Checkers also defines a .play() method. If we have a play_game() function that takes an instance of Chess or Checkers, it could call the .play() method without having to check which class the object is an instance of.

```
#  def play_game(chess_or_checkers):
#    chess_or_checkers.play()
#
#  chess_game = Chess()
#  checkers_game = Checkers()
#  chess_game_2 = Chess()
#
#  for game in [chess_game, checkers_game, chess_game_2]:
#    play_game(game)

#Prints out the following:
#Playing chess!
#Playing checkers!
#Playing chess!
```

In this code, we defined a play_game function that could take either a Chess object or a Checkers object. We instantiate a few objects and then call play_game on each.

When two classes have the same method names and attributes, we say they implement the same interface. An **interface** in Python usually refers to the names of the methods and the arguments they take. Other programming languages have more rigid definitions of what an interface is, but it usually hinges on the fact that different objects from different classes can perform the same operation (even if it is implemented differently for each class).

Example:

We've defined an InsurancePolicy class.

```
class InsurancePolicy:
  def __init__(self, price_of_item):
    self.price_of_insured_item = price_of_item
#Subclass of InsurancePolicy called VehicleInsurance.
#.get_rate() method that takes self as a parameter. Return .001 multiplied by the price of the vehicle.

class VehicleInsurance(InsurancePolicy):
  def get_rate(self):
    return self.price_of_insured_item * .001
# Subclass of InsurancePolicy called HomeInsurance.
#get_rate() method that takes self as a parameter. Return .00005 multiplied by the price of the home.
class HomeInsurance(InsurancePolicy):
  def get_rate(self):
    return self.price_of_insured_item * .00005
```

Polymorphism

All this talk of interfaces demonstrates flexibility in programming. Flexibility in programming is a broad philosophy, but what's worth remembering is that we want to implement forms that are familiar in our programs so that usage is expected. For example, let's think of the + operator. It's easy to think of it as a single function that "adds" whatever is on the left with whatever is on the right, but it does many different things in different contexts:

```
# For an int and an int, + returns an int
2 + 4 == 6

# For a float and a float, + returns a float
```

```
## True
```

```
3.1 + 2.1 == 5.2

# For a string and a string, + returns a string
```

```
## True
```

```
"Is this " + "addition?" == "Is this addition?"

# For a list and a list, + returns a list
```

```
## True
```

```
[1, 2] + [3, 4] == [1, 2, 3, 4]
```

```
## True
```

Look at all the different things that + does! The hope is that all of these things are, for the arguments given to them, the intuitive result of adding them together. **Polymorphism** is the term used to describe the same syntax (like the + operator here, but it could be a method name) doing different actions depending on the type of data.

Polymorphism is an abstract concept that covers a lot of ground, but defining class hierarchies that all implement the same interface is a way of introducing polymorphism to our code.

Another example: Is the same operation happening for each? How is it different? How is it similar? Does using len() to refer to these different operations make sense?

```
a_list = [1, 18, 32, 12]
a_dict = {'value': True}
a_string = "Polymorphism is cool!"
print(len(a_list))
```

```
## 4
```

```
print(len(a_dict))
```

```
## 1
```

```
print(len(a_string))
```

```
## 21
```

Dunder Methods I

One way that we can introduce polymorphism to our class definitions is by using Python's special dunder methods. We've explored a few already, the constructor ___init___ and the string representation method ___repr___, but that's only scratching the tip of the iceberg.

Python gives us the power to define dunder methods that define a custom-made class to look and behave like a Python builtin. What does that mean? Say we had a class that has an addition method:

```
class Color:
  def __init__(self, red, blue, green):
    self.red = red
    self.blue = blue
    self.green = green

  def __repr__(self):
    return "Color with RGB = ({red}, {blue}, {green})".format(red=self.red, blue=self.blue, green=self.
  def add(self, other):
    """
    Adds two RGB colors together
    Maximum value is 255
    """
    new_red = min(self.red + other.red, 255)
    new_blue = min(self.blue + other.blue, 255)
    new_green = min(self.green + other.green, 255)

    return Color(new_red, new_blue, new_green)

red = Color(255, 0, 0)
blue = Color(0, 255, 0)
green = Color(0, 0, 255)

magenta = red.add(blue)
print(magenta)
# Prints "Color with RGB = (255, 255, 0)"
```

```
## Color with RGB = (255, 255, 0)
```

In this code we defined a Color class that implements an addition function. Unfortunately, red.add(blue) is a little verbose for something that we have an intuitive symbol for (i.e., the + symbol). Well, Python offers the dunder method ___add___ for this very reason! If we rename the add() method above we can add them together using the + operator!

```
class Color:
  def __init__(self, red, blue, green):
    self.red = red
    self.blue = blue
    self.green = green

  def __repr__(self):
    return "Color with RGB = ({red}, {blue}, {green})".format(red=self.red, blue=self.blue, green=self.g
# ***This is the difference with the last code *** add changed to __add__
  def __add__(self, other):
    """
    Adds two RGB colors together
    Maximum value is 255
    """
    new_red = min(self.red + other.red, 255)
    new_blue = min(self.blue + other.blue, 255)
    new_green = min(self.green + other.green, 255)
    return Color(new_red, new_blue, new_green)

red = Color(255, 0, 0)
blue = Color(0, 255, 0)
green = Color(0, 0, 255)

### Now we can use + instead of red.add(blue)
# Color with RGB: (255, 255, 0)
magenta = red + blue
print(magenta)
# Color with RGB: (0, 255, 255)
```

```
## Color with RGB = (255, 255, 0)
```

```
cyan = blue + green
print(cyan)
# Color with RGB: (255, 0, 255)
```

```
## Color with RGB = (0, 255, 255)
```

```
yellow = red + green
print(yellow)
# Color with RGB: (255, 255, 255)
```

```
## Color with RGB = (255, 0, 255)
```

```
white = red + blue + green
print(white)
```

```
## Color with RGB = (255, 255, 255)
```

Another example:

There are two classes defined, Atom and Molecule. We give Atom a .__add__ (self, other) method that returns a Molecule with the two Atoms together in a list.

```
class Atom:
  def __init__(self, label):
    self.label = label

  def __add__(self, other):
    """
    A method that returns a Molecule with the two Atoms together in a list.
    """
    return Molecule([self, other])

class Molecule:
  def __init__(self, atoms):
    if type(atoms) is list:
        self.atoms = atoms

sodium = Atom("Na")
chlorine = Atom("Cl")
salt = Molecule([sodium, chlorine])
# salt = sodium + chlorine
```

Dunder Methods II

Python offers a whole suite of magic methods a class can implement that will allow us to use the same syntax as Python's built-in data types. You can write functionality that allows custom defined types to behave like lists:

```
class UserGroup:
  def __init__(self, users, permissions):
    self.user_list = users
    self.permissions = permissions

  def __iter__(self):
    return iter(self.user_list)

  def __len__(self):
    return len(self.user_list)

  def __contains__(self, user):
    return user in self.user_list
```

In our UserGroup class above we defined one constructor and three methods:

- ___init___, our constructor, which sets a list of users to the instance variable self.user_list and sets the group's permissions when we create a new UserGroup.
- ___iter___, the iterator, we use the iter() function to turn the list self.user_list into an iterator so we can use for user in user_group syntax. For more information on iterators, review Python's documentation of Iterator Types.
- ___len___, the length method, so when we call len(user_group) it will return the length of the underlying self.user_list list.
- ___contains___, the check for containment, allows us to use user in user_group syntax to check if a User exists in the user_list we have.

These methods allow UserGroup to act like a list using syntax Python programmers will already be familiar with. If all you need is something to act like a list you could absolutely have used a list, but if you want to bundle some other information (like a group's permissions, for instance) having syntax that allows for list-like operations can be very powerful.

We would be able to use the following code to do this, for example:

```
class User:
  def __init__(self, username):
    self.username = username

diana = User('diana')
frank = User('frank')
jenn = User('jenn')

can_edit = UserGroup([diana, frank], {'can_edit_page': True})
can_delete = UserGroup([diana, jenn], {'can_delete_posts': True})

print(len(can_edit))
# Prints 2
```

```
## 2
```

```
for user in can_edit:
  print(user.username)
# Prints "diana" and "frank"
```

```
## diana
## frank
```

```
if frank in can_delete:
  print("Since when do we allow Frank to delete things? Does no one remember when he accidentally delet
else:
  print("Frank better not have permition to delete things")
```

```
## Frank better not have permition to delete things
```

Above we created a set of users and then added them to UserGroups with specific permissions. Then we used Python built-in functions and syntax to calculate the length of a UserGroup, to iterate through a UserGroup and to check for a User's membership in a UserGroup.

Another example:

You'll find the class LawFirm. We give LawFirm a .___len___() method that will return the number of lawyers in the law firm. We also give a .___contains___() method that takes two parameters: self and lawyer and checks to see if lawyer is in self.lawyers.

```
class LawFirm:
  def __init__(self, practice, lawyers):
    self.practice = practice
    self.lawyers = lawyers

  def __len__(self):
    return len(self.lawyers)

  def __contains__(self, lawyer):
    return lawyer in self.lawyers
```

Review

In this section, we learned more complicated relationships between classes. We learned:

- How to create a subclass of an existing class.
- How to redefine existing methods of a parent class in a subclass by overriding them.
- How to leverage a parent class's methods in the body of a subclass method using the super() function.
- How to define a Python exception that inherits from Exception.
- How to write programs that are flexible using interfaces and polymorphism.
- How to write data types that look and feel like native data types with dunder methods.

These are really complicated concepts! It's a long journey to get to the state of comfortably being able to build class hierarchies that embody the concerns that your software will need to.

Example (follow the following code by steps):

```
# 1 Create a class SortedList that inherits from the built-in type list.
# 2 Recall that lists have a .append() method that takes a two arguments, self and value. We're going t
# Overwrite the append method
# 3 We want our new .append() to actually add the item to the list. After you've appended the new value
# Recall You can call a parent class method using the super() function like this:
# def class_method(self, argument1):
#  super().class_method(argument1)
# 4 After you've appended the new value, sort the list.

# 1
class SortedList(list):
  # 2
  def append(self, value):
    # 3
    super().append(value)
    # 4
    self.sort()
```

We subclassed a Python primitive and introduced new behavior to it.

Some things to consider:

- When a SortedList gets initialized with unsorted values (say if you call SortedList([4, 1, 5])) those values don't get sorted! How would you change SortedList so that the list is sorted right after the object gets created?
- What other Python builtins have functionality "missing"? Could you write a new dictionary that uses a fallback value when it tries to retrieve an item and can't?