Python 3 Fundamentals

Diego López Tamayo [*]

Contents

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

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:

```python
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 /.

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

# Prints "50"
```

```
## 500
```

```python
print(25 * 2)

# Prints "2.0"
```

```
## 50
```

```python
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:

```python
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:

```python
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.

```python
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

```python
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 0x7fa1e7d169c8>
```

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

Pandas Library

Introduction

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)
```

Connecting R and Python

Reticulate package

Thanks to the R reticulate package, you can run Python code right within an R script—and pass data back and forth between Python and R. In addition to reticulate, you need Python installed on your system. You also need any Python modules, packages, and files your Python code depends on.

If you'd like to follow along, install and load reticulate with install.packages("reticulate") and library(reticulate).

To keep things simple, let's start with just two lines of Python code to import the NumPy package for basic scientific computing and create an array of four numbers. The Python code looks like this:

```
import numpy as np
my_python_array = np.array([2,4,6,8])
for item in my_python_array:
    print(item)
```

```
## 2
## 4
## 6
## 8
```

Here's the cool part: You can use that array in R.In this next code chunk, I store that Python array in an R variable called my__r__array. And then I check the class of that array.

```
my_r_array <- py$my_python_array
class(my_r_array)
```

```
## [1] "array"
```

It's a class "array," which isn't exactly what you'd expect for an R object like this. But I can turn it into a regular vector with as.vector(my__r__array) and run whatever R operations I'd like on it, such as multiplying each item by 2.

```
my_r_vector <- as.vector(py$my_python_array)
class(my_r_vector)
```

```
## [1] "numeric"
```

```
my_r_vector_2 <- my_r_vector * 2
```

Next cool part: I can use that R variable back in Python, as r.my__r__array (more generally, r.variablename), such as

```
my_python_array2 = r.my_r_vector_2
print(my_python_array2)
```

```
## [4.0, 8.0, 12.0, 16.0]
```