Python for Business Analytics

A Nontechnical Approach for Nontechnical People

Custom Edition for Hult International Business School

Written by Chase Kusterer - Faculty of Analytics
Hult International Business School
https://github.com/chase-kusterer (https://github.com/chase-kusterer)

Part II: Fundamental Coding Structures

Chapter 4: Numbers, Comparisons, and Randomness

Up to this point, we have covered basic operations related to printing, formatting text, and prompting users for **input()**. These, along with other skills such as working with optional arguments and exploring documentation files, will prove invaluable as we move into the next chapters. In this chapter, we will dive into another important building block: **operands**. Operands come in two basic forms: numerical and comparison. Having a thorough understanding of both is vital to our success as we move into more advanced programming concepts.

This chapter also extends beyond these concepts with random number generation.

4.1 Numerical and Comparison Operands

As you may remember from **Chapter 1 - Setting Up for Success**, Python has been designed to be easy to learn, and numerical operands are no exception. To clarify, a **numerical operand** is a character that has a special meaning, allowing users to perform computational operations. For example, the **+** operand allows users to add two things together. These things can be numbers or something more exotic (for example, strings or more complicated objects). Below is a table of some of the more familiar numerical operands.

Operand	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder from division)
**	exponentiation

The order of operations in Python is the same as in standard mathematics, which can be denoted with the acronym PEMDAS (parentheses, expontentiation, multiplication/division; addition/subtraction). However, Python has a deeper set of operands (comparison operands). Many methods in Python take an input, make a calculation, and then do something based in its result. Therefore, our acronym needs to be extended to PEMDAS+. In other words, Python will first handle all PEMDAS computations before running any comparisons. Below is a table of comparison operands:

Operand	Description
==	checks to see if two operands are equal
!=	Checks to see if two operands are not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

4.2 Practice with Numerical and Comparison Operands

The syntax in the two tables above should be familiar, but if you need a refresher before starting this section, feel free to consult any of the great resources you may have on hand or on the Internet. In the following exercises, try to guess the result of each calculation before checking the solutions.

In [ ]:

```
## Code 4.2.1 ##

5 * 2
```

Show Solution

In [ ]:

```
## Code 4.2.2 ##

3 + (5 * 2)
```

Show Solution

In [ ]:

```
## Code 4.2.3 ##

(3 + 5) * 2
```

Show Solution

In [ ]:

```
## Code 4.2.4 ##

(3 + 5) * 2 == (5 + 3) * 2
```

Show Solution

In [ ]:

```
## Code 4.2.5 ##

int(True)
```

Show Solution

In [ ]:

```
## Code 4.2.6 ##

10 * ((3 + 5) * 2 == (5 + 3) * 2)
```

Show Solution

In [ ]:

```
## Code 4.2.7 ##

(3 + 5) * 2 == 50
```

Show Solution

In [ ]:

```
## Code 4.2.8 ##

int(False)
```

Show Solution

In [ ]:

```
## Code 4.2.9 ##

10 * ((3 + 5) * 2 == 50)
```

Show Solution

In [ ]:

```
## Code 4.2.10 ##

(3 + 5) * 2 != 50
```

Show Solution

4.3 Advanced Operand Practice

Below are some additional practice exercieses. Again, try to determine the each result before revealing the solutions.

In [ ]:

```
## Code 4.3.1 ##

x = 2
y = 3

((x <= y) + (y == y)) / (x - y)
```

Show Solution

In [ ]:

```
## Code 4.3.2 ##

obj_1 = 'apple'
obj_2 = """apple"""

obj_1 == obj_2
```

Show Solution

In [ ]:

```
## Code 4.3.3 ##

obj_1 = 'orange'
obj_2 = """oRaNgE"""

float(obj_1 == obj_2)
```

Show Solution

In [ ]:

```
## Code 4.3.4 ##

obj_1 = "'grape'"
obj_2 = "'grape'"

round(float(obj_1 == obj_2), 5000)
```

Show Solution

In [ ]:

```
## Code 4.3.5 ##

'apple' < 'grape'
```

Show Solution

4.4 Numerical Overwrites

Oftentimes, programmers have the need to overwrite a stored numerical value, as in *Code 4.4.1*. This is quite easy to accomplish, but such an operation happens so frequently that Python contains a built-in shortcut. Instead of overwriting an object as follows:

```
x = x + 5
```

We can do so by combining the plus (+) and equals (=) signs in our code:

```
x += 5
```

This may seem like a trivial enhancement, and you may feel it is unnecessary to learn. However, keep in mind that other people are likely to use this syntax in their code, and as an analyst using Python, you will be expected to know what it means. Think of it as you would a [contraction (https://dictionary.cambridge.org/grammar/british-grammar/writing/contractions)](https://dictionary.cambridge.org/grammar/british-grammar/writing/contractions) in the English language. Even if you do not use them, they're very common in everyday speech.

Essentially, any standard numerical operand can be contracted by simply adding an equals sign after its syntax. Below is a modified table of the numerical operand syntax covered thus far. Feel free to play with them in the open coding block below the table.

Operand	Description
+=	addition and overwrite
-=	subtraction and overwrite
*=	multiplication and overwrite
/=	division and overwrite
%=	modulus and overwrite
**=	exponentiation and overwrite

In [ ]:

```
## Code 4.4.1 ##

# creating an object
x = 5

# overwriting the object
x = x + 5

# printing the object
print(x)
```

In [ ]:

```
## Code 4.4.2 ##

# creating an object
x = 5

# overwriting the object
x += 5

# printing the object
print(x)
```

In [ ]:

```
## Code 4.4.3 ##

# Open coding block
```

4.5 Basic Randoms and Ranges

In many cases, programs require a way to generate a random number or element. There are many packages in Python to help solve such a challenge, and in this chapter we will focus on those existing in the **random** package. As its name implies, [the random package (https://docs.python.org/3.6/library/random)](https://docs.python.org/3.6/library/random) contains several methods to generate random variables.

4.5.1 The Need for Different Types of Generators

The field of random number generation is extremely interesting. This is partly because computers are not very good at doing things that they are not programmed to do. In other words, computers are not meant to be random. If this were *not* the case, tasks such as debugging would be incredibly difficult. This being said, computer scientists have developed methods to address this challenge. Essentially, random numbers come in two forms: pseudo-random and true random. According to [howtogeek.com (https://www.howtogeek.com/183051/htg-explains-how-computers-generate-random-numbers/)](https://www.howtogeek.com/183051/htg-explains-how-computers-generate-random-numbers/):

Computers can generate truly random numbers by observing some outside data, like mouse movements or fan noise, which is not predictable, and creating data from it. This is known as entropy. Other times, they generate "pseudorandom" numbers by using an algorithm so the results appear random, even though they aren't.

When we ask a program to pick a random number, we are asking it to follow a sequence (i.e. an algorithm), resulting in a number being generated that only *appears* to be random. In fact, the computer is not being random at all, but instead is following its programming instructions as expected. For many programming challenges (including those covered in this chapter), pseudo-random number generation is a good solution as it is easy to implement and control. However, in fields such as security encryption, pseudo-random can be dangerous for the very reason that it can be predictable.

Random number generation also interesting because randomness makes it difficult to replicate results. If a researcher was attempting to confirm a study by a colleague, uncontrolled randomness would complicate this process. To hurdle this obstacle, most random number generators come with an optional **seed** argument. Essentially, a seed is a starting point in pseudo-random number generation. In other words, you can conceptualize a seed in random number generation as you would any other seed: they are the starting point for where something starts growing. For example, if we wanted to plant roses in our garden, we would bury rose seeds in the spots where we want our roses to grow. As another example, let's assume we wanted to select one number from the following sequence of ten numbers:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

If we do not set a seed (i.e. a starting point), each number would have a 10% chance of being selected. However, if we set a seed to start at, say, the fifth number, that number has a 100% chance of being selected. This becomes more useful in cases where we would like to select more than one number from a sequence. Also, as you might imagine, random number sequences are quite a bit more complex than the one above. Therefore, it is difficult to intuitively determine the results from using a given random seed.

4.5.2 Random Number Generation and the Philosophy of Analytics

Take a moment to ponder the following question:

What is random?

Try to think of a random number between 1 and 100. How did you make your selection? If asked the same question again, do you feel you would have chosen a different number? Do you feel that all numbers between 1 and 100 had an equal chance of selection, or do you feel that your choice may have been biased? If you felt your selection was biased, select another number. Did you compensate for this bias? If so, do you feel your selection is more random, or more predictable?

Thought experiments such as this help shed light on an interesting notion: it may be completely infeasible to generate a truly random number. We all have biases, and as analysts it is our responsibility to acknowledge and minimize their effects on our endeavors. Each of our ideas should be supported by data, and our results should be replicable. Also, it is unlikely that the results of our analysis will fully reflect the phenomenon we wish to study without error. In fact, a common philosophy in statistics that the true value of something as fundamental as a population mean will never truly be known, and can only be estimated within a certain degree of confidence. This implies that we also need to keep an open mind when attaining results that go against our intuition (i.e. one of our biases). It is unlikely that we will ever generate truly random numbers, just as it is unlikely that we will ever truly attain results that are completely error-free. This perspective will become incredibly valuable when we venture into the chapters dedicated to data analysis.

4.5.3 Ranges

In order to generate a random number in Python, we first need to specify a range of values in which to draw a random number from. In other words, we need to tell Python which values it is allowed to use. In Python, a **range** is a sequence of index values, and in simple terms, **index** means position. For example, each number in our previous sequence has its own position. This helps Python understand where each number is located, and allows us to do many powerful things (covered in **Chapter 5 - Lists and List Operations**). By default, indexes start at zero.

```
Index:    0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9  |
Value:    0  |  1  |  1  |  2  |  3  |  5  |  8  | 13  | 21  | 34  |
```

Note: You may be wondering why sequences in Python start at zero and not one. This is because Python was engineered using concepts from linear algebra. If you are unfamiliar with linear algebra, think of the x- y- coordinate plane, where the origin is at point (0, 0). As our sequence above is one dimensional, its origin is point zero. Thus, Python uses this number as the first index value.

4.5.4 Generating Pseudo-Random Numbers in Python

As stated earlier, we shall be utilizing the **random** package. This package is filled with a delightful set of methods that allow us to add randomness to our code. We shall start with importing **random** and using its **randint()** method. As always when using a new method for the first time, it is important to **read the Python documentation (https://docs.python.org/3/library/random.html#random.randint)**. Calling **help()** on this method gives the following result:

```
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Notice two things in the **help()** documentation: the square brackets *[ ]* and that the range includes both endpoints. These two aspects of the **help()** file are redundant. In mathematics, square brackets imply that an end point is inclusive. Their counterpart, round brackets *( )*, imply the opposite. Little details such as whether or not a range includes endpoints are extremely important. As mentioned in previous chapters, spending a little time to better understand how a method works can save you hours in the long run.

Let's start making some random numbers!

In [ ]:

```
## Code 4.5.1 ##

# importing random (don't forget to do this!)
import random
```

In [ ]:

```
## Code 4.5.2 ##

# spacing out my code so that it is more read
random.randint(a = 1,  # spacing out my code so
               b = 10) # that it is more readable
```

Woo hoo! We've just generated a random number! Notice how if you run this *Code 4.5.2* again, it will generate a different number. This is a perfect opportunity to create a guessing game. *Code 4.5.3* is open for you to create. Try to build something that:

1. asks a user to play your guessing game
2. prompts a user to make a guess
3. generates a random number between 1 and 10
4. prints whether or not the answer was correct

You have all the skills you need to accomplish this task, so have fun!

In [ ]:

```
## Code 4.5.3 ##

# Open coding block
```

Show Solution

In [ ]:

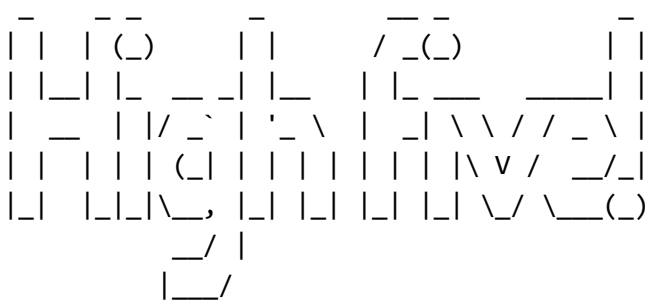
```
## Sample Solution 4.5.3 ##

# user input
print('Can you guess the number I am thinking of?')
guess = input('Pick a number between 1 and 10.\n')

# turning guess into an integer
guess = int(guess)

# generating a random number
number = random.randint(a = 1,
                        b = 10)

print(f"""
Did you guess correctly (True/False)?
{guess == number}

I was thinking of the number {number}!
""")
```

Remember, we can set a seed so that we always get the same result. This becomes very important in later chapters. For now, however, we can use this technique to *game* our guessing game. This is, potentially, a great way to impress your friends that do not know how to read code, and the method **random.seed()** will help us in doing so. If this is your first time using this method, take a minute to look at the help documentation (it briefly explains how the random package generates random numbers... interesting, right!).

In [ ]:

```
## Code 4.5.4 ##

random.seed(508)

# user input
print('Can you guess the number I am thinking of?')
guess = input('Pick a number between 1 and 10.\n')

# turning guess into an integer
guess = int(guess)

# generating a random number
number = random.randint(a = 1,
                        b = 10)

print(f"""
Did you guess correctly (True/False)?
{guess == number}

I was thinking of the number {number}!
""")
```

4.6 Summary



There is much, much more to learn about random number generation in Python. However, we will save these conversations for a later chapter. For now, feel good about yourself for moving a little further in your Python journey.

Chapter 5: Lists and List Operations

5.1 Why Lists are Important

Having learned the basics of indexes in the previous chapter, we are ready to understand the value of declaring lists. For our purposes:

Lists are a way to better organize our information.

Our goal with lists is to store more than one item in the same place. You can think of this as declaring one object that holds many things. Conceptually, lists are the simplest of Python's four built-in data structures (lists, tuples, dictionaries, and sets), and are something that you must be comfortable with before conducting data analysis in Python. They are used very frequently, and your time spent understanding them will pay dividends on your ability to grasp more advanced concepts in Python.

A special note for Excel users:
If you're used to working with Excel sheets, also known as flat files, please note that these exist in Python as well. We call these DataFrames, and they are housed in the **pandas** package. You may be wondering:

"Why do I need to learn about lists if I can use a data structure that I'm more familiar with?"

This mentality held me back when I was first learning to code. I was used to my data being structured in a spreadsheet format and expected it to always be this way. It also didn't help that in most cases, data science competitions provided data in some sort of spreadsheet format, or something that could easily be converted into one. I did not fully understand the value of other data structures until I started to work on real projects. In the real world, data comes in *many* formats, and in today's world of analytics, you will be expected to do much more than work with flat files.

The key lesson to keep in mind is that **coding involves a different type of thinking than working with software such as Excel**. You will not regret the time invested in learning how to work with different data structures.

5.2 Declaring a List

Declaring a list is very similar to declaring any of the objects we have covered in previous chapters. In fact, declaring almost anything in Python is very similar to what we have already covered.

Codes 5.2.1 and *5.2.2* are virtually the same. The only difference is that in the latter, we added **square brackets** *[ ]* around the item being stored. The square brackets tell Python that we are creating a list. We can even declare a list with nothing inside of it. This idea may seem bizarre, but it is actually done quite often. We will explain the rationale for this when we discuss **for loops** in a later chapter.

Notice that the output of the **print()** statement in *Code 5.2.2* also has square brackets. This is to inform you that *my_list* is a list and not a scalar (single) variable. This is also apparent when using the **type()** wrapper around each object, as in *Code 5.2.3*. As can be observed, the objects *my_variable* and *my_list* are of different classes.

In [ ]:

```
## Code 5.2.1 ##

# Declaring a variable
my_variable = 5

print(my_variable)
```

In [ ]:

```
## Code 5.2.2 ##

# Declaring a list
my_list = [5]

print(my_list)
```

A Brief Explanation of Classes
A **class** is a template to create objects. When we created *my_variable*, we created an object that stores an integer. Integers are very common in Python, thus they have a template (i.e. a class). Classes help to ensure that objects behave the way that they are supposed to so that programmers don't get unexpected results. More information on classes can be found in [the Python Documentation (https://docs.python.org/3.7/tutorial/classes.html)](https://docs.python.org/3.7/tutorial/classes.html).

Returning to Lists

Since *my_variable* and *my_list* are of different classes, they are not the same type of object. Therefore, we cannot expect them to behave in the same way. To exemplify this, see *Code 5.2.4*. Even though both objects contain the value 5, we do not get the same result when applying simple numerical operands such as multiplication. Again,

Coding involves a different type of thinking.

Knowing that these objects are of different classes informs you that they are not intended to work in the same way. In fact, one of the first places to check when your code is not working properly is your object types.

In [ ]:

```
## Code 5.2.3 ##

# Printing object types
print(f"""
my_variable is of type:  {type(my_variable)}
my_list is of type: \t {type(my_list)}
""")
```

In [ ]:

```
## Code 2.2.4 ##

# Applying basic operations
print(f"""
my_variable * 2 = {my_variable * 2}
my_list     * 2 = {my_list * 2}
""")
```

5.3 The Advantages of Lists Over Several Independent Objects

Lists have several advantages over creating a large amount of individual objects. In the context of coding for analytics, lists have three of the primary advantages. They are:

- Mutatable (i.e. easy to change)
- Extensible (i.e. easy to add new items/quantities)
- Convertable (i.e. from a list to a DataFrame)

In this chapter, we will work with some of the code from **Project Chapter 5 - Preparing for a Party**. To get started, we need to load some required base objects. This has been done in *Code 5.3.1*, where three lists are being declared to store the names and amounts of various food items. These lists are then being stored into a master list named *food_n_beverage*.

In [ ]:

```
## Code 5.3.1 ##

# Required from Project Chapter 5

chips           = ['bags of chips', 50]
salsa           = ['jars of salsa', 25]
still_water     = ['liters of still water', 100]

food_n_beverage = [chips, salsa, still_water]
```

Before moving forward, notice a few characteristics of the lists created in *Code 5.3.1*. First, the lists *chips*, *salsa*, and *still_water* each contain more than one object type (strings and integers). This is no problem for Python. Also notice that *food_n_beverage* is a list of lists. Since lists are similar to any of our other object types, Python can handle this as well.

Lists are mutable.

If we needed more chips, we could simply change the amount of chips in the chips object. Notice that after doing so, we need to make sure to rerun our *food_n_beverage* object so that it contains this new information.

In [ ]:

```
## Code 5.3.2 ##

chips           = ['bags of chips', 75] # changed to 75
salsa           = ['jars of salsa', 25]
still_water     = ['liters of still water', 100]

food_n_beverage = [chips, salsa, still_water]

# Printing the current food and beverage list
print(food_n_beverage)
```

Lists are extensible.

We can add new items to lists in several different ways. For example, we can use the **.append()** method to add something to the end of a list. We will cover more methods for adding items to lists later in this chapter.

In [ ]:

```
## Code 5.3.3 ##

sparkling_water = ['liters sparkling water', 100]

food_n_beverage.append(sparkling_water) # appending the list

print(food_n_beverage)
```

Lists are convertable.

Lists are easy to convert into other object types. Although this is beyond our current scope, *Code 5.3.4* converts our *food_n_beverage* list into a pandas DataFrame (i.e. Python's equivalent of a spreadsheet). This simple process will be explained in more detail in a later chapter.

' ' '

In [ ]:

```
## Code 5.3.4 ##

import pandas as pd

food_n_beverage_df = pd.DataFrame(food_n_beverage) # df is short for DataFrame

print(food_n_beverage_df)
```

The conversion in *Code 5.3.4* becomes even more powerful when using common DataFrame features. Such features are exemplified in *Code 5.3.5*, where each column is being named and the *Item* column is being used as an index. Performing the latter allows us to search for each item by name instead of its index number.

In [ ]:

```
## Code 5.3.5 ##

food_n_beverage_df = pd.DataFrame(food_n_beverage,
                                  columns = ['Item', 'Quantity'])

food_n_beverage_df.set_index('Item',
                             inplace = True)

print(food_n_beverage_df)
```

5.4 Accessing List Elements

Mastering how to access list elements will pay off greatly in the future. This technique is applicable in a wide array of situations, such as accessing elements in more complex data structures, filtering and subsetting, and creating new columns in a dataset. As you may have noticed, the foods (chips and salsa) are at the beginning of the *food_n_beverage* list. These can be accessed by referencing each element's **index** value.

Indexing in Python

Consider the following list:

```
['first', 'second', 'third']
```

Reiterating from the previous chapter, **indexing in Python starts at 0.** Therefore, the first element of a list is at index 0. You can think of this like the x- y-coordinate plane in geometry, where (0, 0) represents the origin, or where the x-axis meets the y-axis. There are two common methods for indexing lists: forward and backward.

To index forward, place a *non-negative* number in brackets immediately after a list (i.e. example_list[0]). This will access the list's first element.

To index backward, use a *negative* number instead of a non-negative number (i.e. example_list[-1]). The *-1* represents the last element of a list, implying that *-2* represents the second to last element, *-3* the third to last, and so on.

Slicing an index refers to pulling more than one sequential value from a list (i.e. example_list[0:2]). Lists can also be sliced by specifying multiple index values, separated by a semi-colon (i.e. example_list[0:2; 5]).

Note: The last value in an index slice is exclusive, meaning it is not included as part of the slice.

In [ ]:

```
## Code 5.4.1 ##

example_list = ['first', 'second', 'third']

# Printing the first element of the list (forward indexing)
print(example_list[0])
```

In [ ]:

```
## Code 5.4.2 ##

# Printing the last element of the list (backward indexing)
print(example_list[-1])
```

In [ ]:

```
## Code 5.4.3 ##

# Slicing the first and second elements of the list
print(example_list[0:2])
```

In [ ]:

```
## Code 2.4.4 ##

# Equating forward and backwards slicing methods
example_list[0:2] == example_list[-3:-1]
```

Note: More information related to data structures and their methods can be found in the Python documentation (https://docs.python.org/3/tutorial/datastructures.html).

5.5 Adding Items to a List

In this section, we will cover three methods to add items to a list: **.append()**, **.insert()**, and **.extend()**.

list.append()

The method **.append()** adds a single object to the end of a list. This object can be of any data type (string, integer, float, etc.). It can even be a more complex structure such as another list or a tuple. *Code 5.5.2* is an example as to how to use **.append()**. Try running this code block several times to see what happens to *example_list*.

In [ ]:

```
## Code 5.5.1 ##

# Creating a list
example_list = [7]
```

In [ ]:

```
## Code 5.5.2 ##

# Appending the list
example_list.append(9)

# Printing the results
print(example_list)
```

list.insert()

The method **.insert()** takes two arguments: an index and an object. You can consider this as a more complex version of the **.append()** method, giving you the flexibility to choose in which position you would like to insert an item. This method works as follows: list.insert(*index = where you want to insert your object*, *object = what you want to insert*). As with other methods, it is not required to specify each argument by name. As long as the arguments are kept in order, this method should work as expected.

Be careful: **.insert()** places an object in the position you specify and shifts the positions of the following list objects one place to the right. In other words, the index value of list items after the position of your insert are one value higher than their original index. *Code 5.5.3* exemplifies the use of this method.

In [ ]:

```
## Code 5.5.3 ##

# Resetting example_list
example_list = [7, 9]

# Inserting at the beginning of the list
example_list.insert(0, 6)

# Inserting in the middle of the list
example_list.insert(2, 8)

# Inserting at the end of the list
example_list.insert(4, 10)

# Printing the results
print(example_list)
```

Notice what happens when we try to use backward indexing to insert an item at the end of the list (*Code 5.5.4*). As can be observed, our result is not as we expected. However, this makes perfect sense.

.insert() added a new item to the end of the existing list. Since there was already an object at the end of the list, it was shifted one place to the right. Obviously, this was not what we intended. Your first thought may be to try indexing at *-0* instead of *-1*, but Python will treat *-0* as it treats *0* and insert at the beginning of the list. We could also simply count the number of items that are currently in the list, but this is not always feasible given that our list is changing size with each **.insert()**.

In [ ]:

```
## Code 5.5.4 ##

example_list.insert(-1, 11)

print(example_list)
```

Instead, we can employ the [built-in function len()](https://docs.python.org/3/library/functions.html) to solve this challenge. The purpose of **len()** is to count the number of items in a container (an object that can hold more than one item). Therefore, this function will adapt to the changing size of our list, making it an ideal solution for what we are trying to achieve. This has been done in *Code 5.5.5*.

In [ ]:

```
## Code 5.5.5 ##

# Resetting example_list
example_list = [6, 7, 8, 9, 10]

example_list.insert(len(example_list),
                    11)

print(example_list)
```

list.extend()

The method **.extend()** is useful when you need to append multiple objects to the end of a list. It takes an **iterable** as its only argument. The word **iterable** is programming jargon for something that can be operated over sequentially. In other words, an **iterable** is a series of objects that can be processed one-by-one. Each of our lists are examples of iterables.

If we used **.extend()** to add the values of *example_list* to another list (*my_list* in *Code 5.5.6*), it would add each element in *example_list* sequentially. If we instead used the **.append()** method, *example_list* would be added as a single item (a list). This can be observed in *Code 5.5.7*.

In [ ]:

```
## Code 5.5.6 ##

# Creating a new list
my_list = [1, 2, 3, 4, 5]

# Extending the list
my_list.extend(example_list)

# Printing the list
print(my_list)
```

In [ ]:

```
## Code 5.5.7 ##

# Resetting the list
my_list = [1, 2, 3, 4, 5]

# Appending the list
my_list.append(example_list)

# Printing the list
print(my_list)
```

5.6 The Elegance of Simplicity

.extend() was designed to add items to the end of a list. This being stated, what if we wanted to add multiple items to the beginning of a list? For example, what if we wanted to add foods to the beginning of the *food_n_beverage* list mentioned in the beginning of this chapter? We could take the long and tedious approach of coding several **.insert()** statements, but there is a more elegant solution. To get there, let's analyze our current situation:

Current Situation

```
Goal:             Add multiple items to the beginning of a list.
Optimal Solution: One line of code.
Rationale:        One chunk of code will be easy to update/maintain/reuse.

Drawback(s) of Current Techniques
.insert() - more than one line of code is needed
.extend() - adds to the end of a list, not the beginning
```

We could attempt to feed more than one object into **.insert()** as in *Code 5.6.1*, but this will generate a **TypeError**. Type errors occur when we provide a method with an object type that it was not designed to handle. In the case of *Code 5.6.1*, we have provided the **.insert()** method with list for the index argument. Unfortunately for us, the index argument for **.insert()** must be an integer. This makes perfect sense, as indexes are integers.

Also, as **.insert()** is one of Python's built-in base functions, it is very likely that its designers wanted to keep it as simple and stable as possible. This also makes a lot of sense, as other programmers can use **.insert()** to build more advanced functions without having to worry about unexpected results.

In [ ]:

```
# Code 5.6.1 ##

food_n_beverage.insert([0,1], [['avocados', 10],
                               ['bananas', 10]])

"""
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-e8dc5f2de52c> in <module>
      2
      3 food_n_beverage.insert([0,1], [['avocados', 10],
----> 4                                ['bananas', 10]])

TypeError: 'list' object cannot be interpreted as an integer
"""
```

Let's consider **.extend()** from *Code 5.5.6*. This method takes one and only one argument: the iterable to sequentially append to the end of a list. Although this method does not natively solve our problem, what if we restructured our problem instead of our solution? In other words, what if we:

1. Reversed our list so that foods are at the end
2. Used the **.extend()** method to add new foods
3. Reversed the list again so that foods are back at the beginning

This would solve our problem, and we can use a wonderful method named **.reverse()** to help accomplish this. How did I know about this method? I read the Python documentation (https://docs.python.org/3/tutorial/datastructures.html) related to lists. If you skipped over this link in Section 5.4 above, now is a good time to do your research. Let's apply this method and see if it solves our problem.

Reversing a list with .reverse()

For lists, the **.reverse()** method does exactly as its name implies: it reverses the order of a list. We can see this from *Codes 5.6.2* and *5.6.3*.

We need to be careful to remember whether or not our list is reversed. As this is our first time working with **.reverse()**, it is best to take a conservative approach and create a new object for the reversed-version of our list (*Code 5.6.4*). In doing so, we also need to be careful not to copy our original list in the wrong way.

In [ ]:

```
## Code 5.6.2 ##

my_list = [1, 2, 3, 4, 5]

print(my_list)
```

The *Wrong* Way to Copy Lists

Before we learn how to do this the right way, let's see what happens when we copy an object in the *wrong* way. In *Code 5.6.4*, we have created a new object (*rev_my_list*) with the intention of using it to make our reversal while preserving the integrity of our original list (*my_list*).

So far, everything appears to be working properly. However, notice what happens to *my_list* in *Code 5.6.5*. Our change to *rev_my_list* also changed *my_list*. This may seem strange, but this is the way objects work in Python. When we ran:

```
rev_my_list = my_list
```

in *Code 5.6.5*, we created two objects that share the same address. In other words, the lists are two different names for the same object. Changing either list will affect both of them. This is better observed by the use of the built-in function **id()** in *Code 5.6.6*. We can resolve this problem by using the method **.copy()**.

Copying a list with .copy()

In short, the method **.copy()** makes a copy of an object, storing it in a new address. In the context of our lists, this will prevent both lists from changing when we use **.reverse()** and **.extend()**. This has been done in *Code 5.6.7*. As can be observed, using **.copy()** prevents the undue consequence of two objects being effected by each other's changes. We have arrived at our intended result, and covered some of the key components of lists and list operations along the way.

Wrapping Up

At the time of this writing, Python 3 has 11 built-in methods for lists, which can be found in [the Python documentation (https://docs.python.org/3/tutorial/datastructures.html)](https://docs.python.org/3/tutorial/datastructures.html). Masting the ones presented in this chapter, as well as indexing and slicing, shall provide a strong foundation to move forward into more advanced concepts. **Project Chapter 5 - Preparing for a Party** is designed to help you achieve such mastery.

In [ ]:

```
## Code 5.6.3 ##

my_list.reverse()

print(my_list)
```

In [ ]:

```
## Code 5.6.4 ##

my_list = [1, 2, 3, 4, 5]

# The WRONG way to copy lists
rev_my_list = my_list

print(f"""
my_list:     {my_list}
rev_my_list: {rev_my_list}
""")
```

In [ ]:

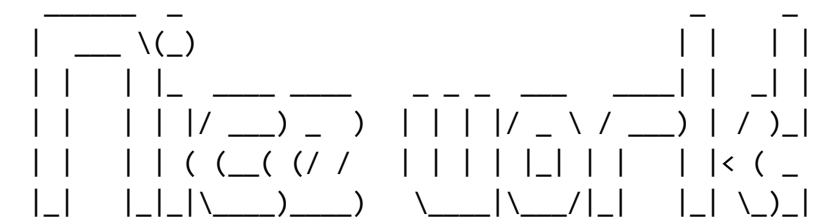
```
## Code 5.6.5 ##

# Reversing my_list
rev_my_list.reverse()

# Extending rev_my_list with new items
rev_my_list.extend([0, -1])

# Reversing the list back to original form
rev_my_list.reverse()

# Printing the result
print(f"""
my_list:     {my_list}
rev_my_list: {rev_my_list}
""")
```

In [ ]:

```
## Code 5.6.5 ##

# Reversing my_list
rev_my_list.reverse()

# Extending rev_my_list with new items
rev_my_list.extend([0, -1])

# Reversing the list back to original form
rev_my_list.reverse()

# Printing the result
print(f"""
my_list:     {my_list}
rev_my_list: {rev_my_list}
""")
```

In [ ]:

```
## Code 5.6.7 ##

# Resetting my_list
my_list = [5, 6, 7, 8]

# Copying my_list
rev_my_list = my_list.copy()

# Reversing rev_my_list
rev_my_list.reverse()

# Extending rev_my_list with new items
rev_my_list.extend([4, 3, 2, 1])

# Reversing rev_my_list
rev_my_list.reverse()

# Printing the result
print(f"""
my_list:     {my_list}
rev_my_list: {rev_my_list}
""")
```



Chapter 6: Conditional Statements and Controlling Input

Up to this point, we have covered several fundamental coding principles. Now, we are ready to dive into more advanced structures, starting with techniques to conditionally control a code's output. In short, we are ready to discuss conditional statements. A **conditional statement** (also known as a conditional) is a coding syntax that runs if a given condition is met. Conditional statements are very common in the real world. Take, for example, the following statement:

```
If it's raining, the baseball game will be canceled.
Otherwise, the game will take place as scheduled.
```

As generally in the real world, Python conditional statements start with the word *if*. The syntax for the alternative statement (*'otherwise'* in the example above), is *else*. In the case of the statement above, if it is not raining, the condition to cancel the game is not met and the thus is disregarded. This is better exemplified in the following expansion of our baseball example:

```
If it's raining, the baseball game will be canceled and we will email all of the p
eople
that bought tickets. We also have to issue refunds, and will do so by directly
crediting people's credit cards. Since some people paid in cash, we will provide t
hem
with a cash voucher in the email that can be redeemed within the next 90 days. Als
o,
since some people may have traveled a long distance to attend the game, we need to
make
sure to remind them that our Terms and Conditions state that we are not responsibl
e for reimbursing travel expenses. Finally, we should inform our call center that
 they may
receive a high call volume given the situation.

Otherwise, the game will take place as scheduled.
```

Compared to the first example, the second takes longer to read and process. It would be incredibly inefficient if we were required to read and process all instructions for every conceivable situation every time there was a baseball game, especially if the conditions for each instruction were not met. Python feels the same way, and takes a similar, pragmatic approach.

A Note to Excel Users
The syntax covered in the chapter is very similar to the IF function in Excel (https://support.office.com/en-us/article/if-function-69aed7c9-4e8a-4755-a9bc-aa8bbff73be2):

```
IF(logical test, [value if true], [value if false])
```

This function is excellent and has tremendous value. However, you have also likely experienced the frustration of debugging an IF function with multiple criteria:

```
IF(logical test, [value if true], IF(logical test, [value if true], IF(logical tes
t,
[value if true], IF(logical test, [value if true], IF(logical test, [value if tru
e], IF(logical test, [value if true], IF(logical test, [value if true], [value if
false])))))))
```

As you will discover, this concept in Python is quite powerful and much easier to organize. Another advantage of using Python for such syntax is that the code is much easier to replicate when new data is available (i.e. new spreadsheets).

6.1 Python's Pragmatic Approach to Conditional Statements

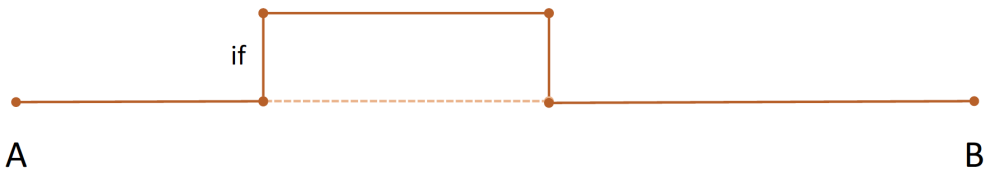


Figure 6.1: Conceptual Drawing of a Conditional Statement

Figure 6.1 portrays a conceptual drawing of a conditional statement. As can be observed, **if** a condition is true, Python will change the path it follows to the code of this condition. Thus, Python pretends that the other code path does not exist. Once Python is finished running the conditional code, it returns to the main path. Think of this as you would the phrase:

Out of sight, out of mind.

When Python reads a conditional statement, it makes a decision as to which path to take based on whether or not its condition is true. **This is not AI**. The computer is not thinking; it is simply following pre-written instructions. You'd be incredibly surprised at what some charlatans of the trade will try to market as AI to the ignorant buyer. It is my hope that by the end of this book you will be better educated in making such distinctions.

Code 6.6.1 contains a basic conditional and *Figure 6.2* explains its anatomy. The first line of a conditional statement starts with the word **if** and ends with a **semicolon** (**:**). This informs Python to check to see

if the condition in between these two syntaxes is true or false. In other words, the conditional statement in *Code 6.1.1* is checking to see if *people* is greater than *chairs*. After this, one of three things will occur:

- If the condition is **true**, Python will run the indented code under the statement.
- If the condition is **false**, Python will ignore all of the indented code and resume processing when it sees a line of code that is not indented.
- An error will occur, as will be discussed later in this chapter.

In Python, **indentation is very important.** It is a critical component of conditional statements, loops, and functions. You can think of it as a special character that tells Python where it can skip code and where it must resume, allowing it to process your code more efficiently.

In [ ]:

```
## Code 6.1.1 ##

# declaring objects
people = 100
chairs = 50

# creating a condition
if people > chairs:
    print('We need more chairs!')
```

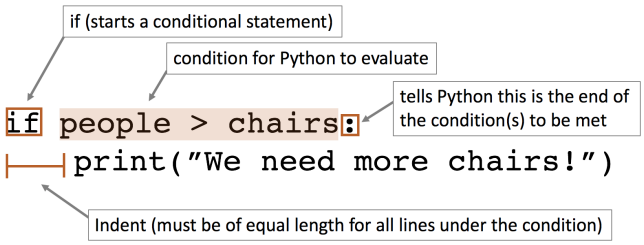


Figure 6.2: Anatomy of a Conditional Statement

In *Code 6.1.1*, the condition is met and the **print()** statement runs. However, what if we wanted to display a message when the condition is *not* met? This is easily done with the use of an **else** clause. Essentially, an **else** clause runs in the event that the criteria given in an **if** statement is not met. This syntax is not mandatory, but can add a lot of value to your code. This can be exemplified from a modified version of the baseball example at the beginning of this chapter:

```
IF it's raining...

ELSE, the game will take place as scheduled.
```

Without the **else** clause, there would be no instructions to start playing the game! Common sense dictates that the game will start without the need for explicit instructions, but programs do not share the same common sense as humans. This lack of common intuition is similar to a story from when my uncle was in rodeo:

> One time, my uncle was carpooling to a rodeo competition with two other people. In order to be at their best, he and one of his carpool mates decided they would sleep until they arrived at the competition. The person they left responsible for driving was not the brightest of bulbs, but the only thing he needed to do was stay on the same road the entire time. My uncle and the other passenger instructed him to:
>
> ***Stay on this road and don't get off for any reason!***
>
> Later, they awoke to find that the driver did as requested, but had driven two hours past the exit

for the competition. On a side note, while the driver was competing in the rodeo, he got bucked off his steer and landed on his head. His head was didn't injured, but somehow he broke his foot.

Code 6.1.2 is a modification of *Code 6.1.1*. In this code, since *chairs* is greater than *people*, the **else** statement runs instead of the **if** statement. Notice how the **else** clause does not contain a condition. This is because it only runs if all previous conditions are not met. Also notice how the **else** clause is on the margin, also known as Column 0. When Python realized the condition in the **if** statement was not met, it ignored all indented code and jumped to the next line at Column 0. This reemphasizes the importance of indents.

In [ ]:

```
## Code 6.1.2 ##

# declaring objects
people = 100
chairs = 150 # changed to 150

# creating a condition
if people > chairs:
    print('We need more chairs!')

# writing an else clause (not indented)
else:
    print('We have enough chairs.')
```

6.2 Different Actions for Different Conditions

Conditionals in Python can be extended to include different actions for different conditions. This can be conceptualized as in *Figure 6.3*, and can be extended as many times as needed. The syntax to extend a conditional is **elif**, and it has the same anatomy as an **if** statement. As can be observed from *Code 6.2.1*, **elif** statements are placed at Column 0. This tells Python if the condition in the first statement is not met, read the next conditional statement. If that condition is also not met, continue until you find an **else** clause, or another line of code on Column 0.

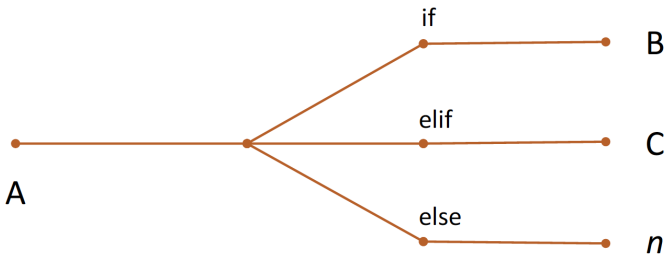


Figure 6.3: Conceptual Drawing of an Extended Conditional Statement

What happens if more than one condition is met?

In many situations, the conditions from more than one statement may be met (i.e. both the **if** and **elif** statements evaluate to true). In cases such as these, only the first condition will run. Remember, once Python finds a conditional statement that evaluates to

true, it ignores all of the other conditional statements, even if their conditions are more suitable for a given situation. In other words, if the conditions of the **if** statement are true, Python will skip over all of its following conditional statements (**elif** and **else**). Order matters, so make sure you organize your code according to the way you would like it to run. This concept is exemplified in *Codes 6.2.2* and *6.2.3*.

In [ ]:

```
## Code 6.2.1 ##

# declaring objects
people = 100
chairs = 150

# writing a conditional statement
if people > chairs:
    print('We need more chairs!')

elif people == chairs:
    print('Good job everyone!')

else:
    print('We have enough chairs.')
```

In [ ]:

```
## Code 6.2.2 ##

# declaring objects
people = 100
chairs = 100 # changed to 100

# changed if condition to >=
if people >= chairs:
    print('We need more chairs!')

# elif will never run because
# the if condition is met
elif people == chairs:
    print('Good job everyone!')

else:
    print('We have enough chairs.')
```

In [ ]:

```
## Code 6.2.3 ##

# declaring objects
people = 100
chairs = 100 # changed to 100

# reordering the conditions
if people == chairs:
    print('Good job everyone!')

# has a chance to run
elif people >= chairs:
    print('We need more chairs!')

else:
    print('We have enough chairs.')
```

As can be observed, the **elif** statement in *Code 6.2.2* has no chance of ever running because the condition *people == chairs* is met in the **if** statement. By reordering the sequence of the conditionals (*Code 6.2.3*), each conditional statement has a chance of running given the values of *people* and *chairs*.

6.3 Using *else* for something else

Let's shift our attention to *Code 6.3.1*, which is a reorganized version of *Code 4.5.3* from **Chapter 4: Numbers, Comparisons, and Randomness** with a few modifications:

- The user input prompt offers a hint, telling users the number to guess
- The line *guess = int(guess)* has been commented out.
- The final **print()** statement has been turned into a conditional.

Try inputting the correct number and see which condition runs.

In [ ]:

```
## Code 6.3.1 ##

# adapted from Code 4.5.3

# importing random
import random

# generating a random number
number = random.randint(a = 1,
                        b = 10)

# user input
print('Can you guess the number I am thinking of?')
guess = input(f"""
Pick a number between 1 and 10.
(Hint: The number is {number}.)
""")

# turning guess into an integer
#guess = int(guess)

# creating a conditional
if guess == number:
    print('Nice job!')

else:
    print(f"Good try, but I was thinking of the number {number}.")
```

No matter what number a user inputs, the **else** clause will always run. Without the line *guess = int(guess)*, the objects *number* and *guess* are of different types, thus Python will not consider them to be equal. Obviously, this is a bug in our code, which can be recognized from the **else** clause running. Even though **else** is not mandatory, it is a good practice to include it in conditional statements for this very reason. As our programs become more complex, techniques to recognize bugs become all the more important. Let's consider a more complex example of using **else** as a basic approach to for this task. Take, for example, *Code 6.3.2* below.

Note: For a review of type conversions, return to **Chapter 3: User Input and Variable Types**.

In [ ]:

```
## Code 6.3.2 ##

guess = input(f"""
Which of these sports is played with a basket and a ball?
(Please input 1, 2, or 3.)
    1. Basketball
    2. Fencing
    3. Hockey

""")

if guess == '1':
    print("\nYou know your sports!")

elif guess == '2':
    print("\nI'm sorry, that's incorrect.")

elif guess == '3':
    print("\nI'm sorry, that's incorrect.")

else:
    print('\nSomething went wrong')
```

Code 6.3.2 has some merit in controlling for bugs. It clearly explains that a user should input a number between one and three. Also, the first three conditional statements consider the fact that Python will treat user input as a string, controlling each condition accordingly. Finally, an **else** clause has been coded to catch invalid user input or any additional bugs. If a user were to input *'Basketball'* instead of *1*, the **else** clause would run.

Although *Code 6.3.2* makes a good effort to control for bugs, it does not provide an optimal user experience. Most notably, the code is too restrictive. Even though the input prompt clearly explains that a user should input *1*, *2*, or *3*, *Basketball* should be a valid answer. This would allow more flexibility to users and create a better user experience. Thinking along these terms, should the inputs *'basketball'*, *'BASKETBALL'*, and *'BaSkEtBaLl'* also be valid entries? Lucky for us, with a few simple modifications, our code can handle such situations with ease.

6.4 Adding *in* and Controlling for Case Sensitivity

One of the most powerful syntaxes to address our challenge of making *Code 6.3.2* more user friendly is the two letter word **in**. This syntax checks to see if a certain element exists inside a larger set of elements. In the case of strings, **in** can be used to determine if a certain sequence (i.e. a substring) exists in a larger sequence. In other words, it allows us to check if a

character or set of characters exists in a string. In *Code 6.4.1*, since *'P'* exists in the string stored in the object *name*, the line of code:

```
'P' in name
```

evaluates to true. However, since there is no lowercase *'p'* stored in *name*, *Code 6.4.2* evaluates to false. Remember, in Python each letter has its own unique address, therefore capital and lowercase letters are different. This aspect of Python becomes a challenge when prompting users for input as there is no guarantee that a user will follow our directions.

In [ ]:

```
## Code 6.4.1 ##

name = 'Python'

# big P
'P' in name
```

In [ ]:

```
## Code 6.4.2 ##

name = 'Python'

# small p
'p' in name
```

For this reason, we must plan accordingly. Instead of trying to consider every possible case for user input, we can utilize one of Python's wonderful [string methods (https://docs.python.org/3/library/stdtypes.html#string-methods)](https://docs.python.org/3/library/stdtypes.html#string-methods). Several methods are available to help us accomplish our task, including those listed in the following table.

Method	Description
.capitalize()	capitalizes the first character of a string, leaving the rest lowercase
.casefold()	lowercases all characters in a string (*includes* special characters)
.lower()	lowercases all characters in a string (*excludes* special characters)
.upper()	capitalizes all characters in a string

Any of these methods would work for our task of standardizing the word *'basketball'* in *Code 6.3.2*. However, what if instead of inputting text from the English alphabet, a user input 'ß' from the German alphabet? Such a case may be difficult to control given it is in another language. Since it is already lowercase, methods such as **lower()** would do nothing to this character. However, according to the example from the string methods link above, **casefold()** will convert 'ß' to 'ss'. *Code 6.4.3* uses this character to test the results of each of the aforementioned string formatting methods.

In [ ]:

```
## Code 6.4.3 ##

german_example = 'ß'

# Testing different methods
print(f"""
FORMAT     | RESULT
original   | {german_example}
capitalize | {german_example.capitalize()}
casefold   | {german_example.casefold()}
lower()    | {german_example.lower()}
upper()    | {german_example.upper()}
""")
```

After applying **.casefold()** to our basketball example, our program will become more robust, granting users more input flexibility and potentially creating a better user experience. However, what if a user were to input something more elaborate? For example: *"I'm not sure, but I think it's probably basketball."* Obviously, our code is not designed to handle such situations. Coding for every conceivable input using syntax such as == or *!=* would be incredibly inefficient. Luckily, this functionality can be developed with the use of **in**.

Instead of coding a conditional statement as follows:

```
elif guess == 'basketball':
    print("\nYou know your sports!")
```

we can utilize **in** to search a user's input for the key word or phrase we are looking for:

```
elif 'basketball' in guess:
    print("\nYou know your sports!")
```

To further exemplify, *Code 6.4.4* is checking whether or not the word *'stella'* is part of the word *'constellation'*. As this exact character sequence exists in the word *'constellation'*, this code evaluates to true.

The syntax **in** can also be applied to other data structures, such as the list in *Code 6.4.5*. This code is checking if *'z'* is a member of the object *lst*. As *'z'* is not an element of this object, *Code 6.4.5* evaluates to false.

The counterpart of **in** is **not in**. This syntax works in a similar fashion to *!=* from **Chapter 4 - Numbers,**

In [ ]:

```
## Code 6.4.4 ##

# checking for membership
'stella' in 'constellation'
```

Comparisons, and Randomness. *Code 6.4.6* utilizes this syntax in order to reverse the logic of **in**. Since *'key'* is not in inventory, the **elif** clause runs.

In [ ]:

```
## Code 6.4.5 ##

# creating a list
lst = ['a', 'b', 'c']

# checking for membership
print('z' in lst)
```

Code 6.4.7 returns to our basketball example, taking **casefold()** and **in** into consideration. Although more user friendly, this has greatly increased the amount of conditional statements we are using. The next section will discuss ways in which we can consolidate conditions, thus allowing us to shorten our code.

In [ ]:

```
## Code 6.4.7 ##

# adapted from Code 6.3.2

# adapting the directions
guess = input(f"""
Which of these sports is played with a basket and a ball?
(Please input the number or name of your choice.)
    1. Basketball
    2. Fencing
    3. Hockey

""")

# converting guess using .casefold()
guess = guess.casefold()

# extending the conditional
if guess == '1':
    print("\nYou know your sports!")

elif 'basketball' in guess:
    print("\nYou know your sports!")

elif guess == '2':
    print("\nI'm sorry, that's incorrect.")

elif 'fencing' in guess:
    print("\nI'm sorry, that's incorrect.")

elif guess == '3':
    print("\nI'm sorry, that's incorrect.")

elif 'hockey' in guess:
    print("\nI'm sorry, that's incorrect.")

else:
    print('\nSomething went wrong')
```

6.5 Consolidating Conditionals with *and* / *or*

Instead of writing several conditionals that lead to the same result, we can utilize the syntaxes **and** and **or** to help us consolidate our code. As an example, let's apply the **or** syntax to *Code 6.4.7*.

In [ ]:

```
## Code 6.5.1 ##

# adapted from Code 6.4.7

# adapting the directions
guess = input(f"""
Which of these sports is played with a basket and a ball?
(Please input the number or name of your choice.)
    1. Basketball
    2. Fencing
    3. Hockey

""")

# converting guess using .casefold()
guess = guess.casefold()

# extending the conditional
if guess == '1' or 'basketball' in guess:
    print("\nYou know your sports!")

elif guess in ['2', '3'] or 'fencing' in guess or 'hockey' in guess:
    print("\nI'm sorry, that's incorrect.")

else:
    print('\nSomething went wrong')
```

With the use of **or**, our code has been significantly shortened. As expressed in the code above, **in** and **not in** can be utilized in tandem with **and** and **or**. The following codes are designed to give you practice in using these syntaxes. Fill in the blanks so that each code block evaluates to true.

In [ ]:

```
### Code 6.5.6 ##

name = 'xiong'

'xing' in name ___ 'ong' in name
```

Show Solution

In [ ]:

```
### Code 6.5.7 ##

places = ['CAN', 'MEX', 'USA']


'CAN' ___ places and 'UK' ___ places
```

Show Solution

In [ ]:

```
### Code 6.5.8 ##

colors = ['red', 'orange', 'yellow',
          'green', 'blue', 'indigo',
          'violet']

len(colors) > 50 ___ \
'r' ___ colors[0] ___ \
'r' ___ colors[2]
```

Show Solution

In [ ]:

```
### Code 6.5.9 ##

primes = [2, 3, 5, 7, 11]

2 ___ primes ___ \
1 ___ _primes ___ \
53 ___ primes
```

Show Solution

6.6 Nested Conditionals

This final section will discuss **nested conditionals** or conditionals that are inside of other conditionals. **Nested conditionals** allow us to develop programs for more complex situations. Conceptually, nested conditionals can be thought of as in *Figure 6.4* below.

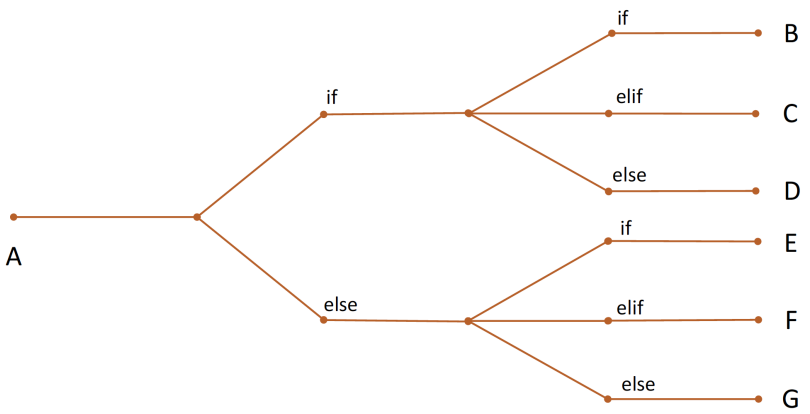


Figure 6.4: Conceptual Drawing of a Nested Conditional Statement

The logic of *Figure 6.4* can be extended to as many levels of inner nesting as needed. The key to managing nested conditionals is to **make sure that each statement is properly indented.** To exemplify, let's return to our people and chairs example from *Section 6.2*.

In [ ]:

```
## Code 6.6.1 ##

# adapted from 6.2.3

# declaring objects
event_manager = 'Mona'
people = 100
chairs = 100

# outer conditional (indented one level)
if 'M' in event_manager:

    # inner conditional (indented two levels)
    if people == chairs:
        print("We'd better get some extra chairs just in case!")

    elif people > chairs:
        print("Quick, find more chairs!")

    else:
        print("We have enough chairs.")

# back to outer conditional (indented one level)
elif 'M' not in event_manager:
    print("I think we can get by with less chairs.")

else:
    print("Something went wrong.")
```

As can be observed, utilizing nested conditionals allows us to develop programs that can handle specific situations. This alleviates the pressure of trying to ensure that our conditionals are in an optimal order, making it possible for the results of each condition to run. Remember, indenting is important and should be done with care. This will save time as indentation errors can be hard to spot and debug.

6.7 Summary

A conditional statement (i.e. a conditional) is a coding syntax that runs if a given condition is met. They are very common in many programming languages, and in Python are structured as follows:

```
if [CONDITION]:
    [CODE TO RUN IF CONDITION IS MET]
```

Conditionals can be extended with the use of **elif** and **else**, and several conditionals can be combined with the use of **and** / **or**. Remember, indenting is very important, as it tells Python what code it is required to run and what code it can skip if a given condition is not met.

Finally, case methods such as **.capitalize()**, **.casefold()**, **.lower()**, and **.upper()** can help limit the amount of conditionals needed to control user input as well as in other situations where strings are being utilized. Finally, **in** and **not in** are powerful syntaxes that check to see if a certain element exists inside a larger set of elements.