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 III: Intermediate Coding Structures

Chapter 7: for Loops and Basic Data Manipulation

Conditional statements such as those starting with **if** are an excellent starting point for controlling the flow of a code. Such statements have been appropriately termed control flow tools (https://docs.python.org/3/tutorial/controlflow.html). This chapter will expand on such tools with a syntax known as **for loops**. **Loops** are a powerful coding structure that allow programmers to **iterate**. In other words, loops allow us to run a code over several items stored in an object without the need to implicitly write code for each run. If this isn't making sense just yet, think of loops as a faster/more advanced alternative to copy/pasting. We will solidify our understanding as we move forward through this chapter.

7.1 Types of Loops

Loops come in two forms: **for loops** and **while loops**. The former is used to iterate over an iterator (i.e. strings, lists, arrays, columns of data, etc.). Without more detailed instruction, **for loops** will run until they have completely iterated over every item in an object. For example, if we have five items in a list, a basic **for loop** will iterate over all five items one at a time. As with the conditional statements covered in **Chapter 6: Conditional**

Statements and Controlling Input, loops operate on a boolean pretense (True/False). For example, if you run a **for loop** on a list, the loop will run until it is *False* that there are still items in the list to iterate over. Just like with conditional statements, if a condition is *False*, Python will move on to the next line of code formatted at Column 0. An example of a **for loop** is displayed in *Code 7.1.1*.

A basic **while loop** can be thought of as an extension of a conditional statement. As long as the given condition is true, a **while loop** will continue to iterate. An example is displayed in *Code 7.1.2*, and these will be covered in more detail in **Chapter 8: While Loops and Basic Error Handling**.

In [ ]:

```
## Code 7.1.1 ##

lst = ['a', 'b', 'c', 'd', 'e']

# for loop
for element in lst:
    print(element)
```

In [ ]:

```
## Code 7.1.2 ##

x = 5

# while loop
while x > 0:
    print(x)
    x -= 1
```

7.2 Working with *for* Loops

As can be observed, the **for loop** in *Code 7.1.1* prints each item in *lst* one at a time. *Code 7.2.1* runs the same operation using list indexing, as explained in **Chapter 2: Printing, Dynamic Strings, and Escape Sequences**. To exemplify the value of **for loops**, imagine that *lst* had 5,000 items instead of just five. In such a case, 5,000 lines of code would need to be written where a single **for loop** would suffice.

Essentially, the **for loop** in *Code 7.7.1* starts by operating on the first item of *lst*, printing its value. When it has completed this task, it moves on to the second item and performs the same operation. This continues until the contents of the loop have run on all elements in *lst*. Afterwards, the **for loop** realizes that *lst* no longer has any items to iterate over and stops running (i.e. it runs until it is *False* that there are still items in the list to iterate over).

The anatomy of a **for loop** (*Figure 7.1*) is very similar to the anatomy of a conditional statement. The first line of a loop starts with a syntax identifying which type of loop is being coded (in this case a **for loop**), and ends with a semicolon (:). In between these two pieces of syntax is a phrase that tells Python what object to look into and what to iterate over. In other words:

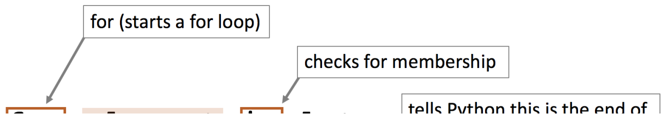
In [ ]:

```
## Code 7.2.1 ##

lst = ['a', 'b', 'c', 'd', 'e']

print(lst[0]) # a
print(lst[1]) # b
print(lst[2]) # c
print(lst[3]) # d
print(lst[4]) # e
```

```
    THINGS_TO_ITERATE in ITERATOR
```



Naming an Iteration Object

It makes no difference what you name the *THINGS_TO_ITERATE* component (also known as an iteration object). This is just an object name of your choosing. In *Code 7.2.2*, *random_name* has been used instead of *element*. More formal coders like to use the letter *i*. The key is to make sure that you do not use an object name that already exists in your code as this may cause conflict. Once an iteration object has been named, it will exist in your Python environment and will store the value of the last iteration of it's respective loop. In the case of the loop in *Code 7.2.2*, the last iteration from *lst* was the letter *'e'*. This can be observed from printing the contents of *random_name*, as in *Code 7.2.3*. Again, **it makes no difference what you name your iteration object**. This being stated, it is in your best interest to name it something that helps you remember what your loop is supposed to do. *Code 7.2.4* is an open coding block designed for you to be able experience the fact that it does not matter what the iteration object has been named.

```
In [ ]:
## Code 7.2.2 ##

# adapted from Code 7.1.1

lst = ['a', 'b', 'c', 'd', 'e']

# for loop
for random_name in lst:
    print(random_name)
```

```
In [ ]:
## Code 7.2.3 ##

# printing random_name
print(random_name)
```

Note: Try running *Code 7.2.4* without filling in the blank spaces (i.e. the underscores). According to Python, this also counts as a valid object name and the code runs without throwing an error.

```
In [ ]:
## Code 7.2.4 ##

# open coding block

fruits = ['apple', 'banana', 'grape',
          'orange', 'pear', 'tomato']

# for loop
for _____ in fruits:
    print(_____)
```

Loop Content

Notice how the body of a loop (i.e. the code run when the loop has items to iterate over) is indented. As mentioned in **Chapter 6: Conditional Statements and Controlling Input**, indenting is very important in Python. Also notice that the iterator variable is referenced in the loop's body. This is accomplishing two tasks, namely defining:

- how many iterations to run
- what to run in each iteration

These do not necessarily need to be the same. This is exemplified in *Code 7.2.5*, where instead of printing *random_name*, the loop is printing *number*. A lot is happening in this code, so let's break it down step-by-step.

```
In [ ]:
## Code 7.2.5 ##

# adapted from Code 7.2.2

lst = ['a', 'b', 'c', 'd', 'e']
number = 5

# for loop
for random_name in lst:
    print(number)
    number += number
```

Line	Interpretation
5-6	The objects *lst* and *number* are being defined.
9	for loop is being defined. Python is determining how many iterations to run by counting the length of *lst*.
10	*number* is being printed.
11	*number* is being incremented by itself (doubling with each iteration)

7.3 Iterating on Inner Objects

Oftentimes, **for loops** contain conditional statements. It is very likely that we will encounter situations where we would like to replace values in a list that meet a certain condition, while leaving all other list elements alone. For example, this may occur when working with datasets that contain missing values. Missing values are quite intriguing and will be covered in greater detail in a later chapter. However, for this chapter, it is important to realize that they hinder several analytical operations in programming languages such as Python.

The list in *Code 7.3.1* represents retail prices at a local clothing store. For this example, the company's database has become corrupted and some of the pricing data has been lost (labeled as *'missing'* in the dataset). Since *inventory* is a list of lists, the for loop in *Code 7.3.1* returns a list with each iteration.

In [ ]:

```
## Code 7.3.1 ##

# creating an inventory list
inventory = [['red blouse', 'missing'], ['pink blouse', 95],
                     ['orange blouse', 100], ['yellow blouse', 'missing'],
                     ['green blouse', 'missing'], ['blue blouse', 90],
                     ['navy blouse', 103], ['purple blouse', 129],
                     ['black blouse', 95], ['white blouse', 102],
                     ['red skirt', 75], ['pink skirt', 75],
                     ['orange skirt', 66], ['yellow skirt', 67],
                     ['green skirt', 73], ['blue skirt', 'missing'],
                     ['navy skirt', 79], ['purple skirt', 74],
                     ['black skirt', 'missing'], ['white skirt', 68]]

for info in inventory:
    print(info)
```

Looping Deeper
Given that lists are iterable, lists within lists are also iterable. We can iterate over each item in the inner

lists by passing two iteration objects instead of one (*Code 7.3.2*). If we only wanted to print prices, we could modify our loop as follows:

```
for i, price in inventory:
    print(price)
```

```
In [ ]:
## Code 7.3.2 ##

# Looping over each inner list
for item, price in inventory:
    print(item + ':', price)
```

Note: In proper terms, the loop in *Code 7.3.2* is **unpacking** two values from *inventory* (*item* and *price*). Since each of the inner lists have a length of two, Python knows it needs to iterate over each inner list instead of the outer list.

Looping Even Deeper
As you may recall from *Section 7.1*, loops can also iterate over strings (i.e. each clothing item in our dataset). In the case of the *inventory* object, this can be accomplished by nesting a loop inside of another loop. Below is a step-by-step explanation of *Code 7.3.3*:

```
In [ ]:
## Code 7.3.3 ##

# outer loop
for item, price in inventory:

    # adding an inner loop
    for letter in item:
        print(letter)
```

1. The outer loop tells Python to iterate over each inner list in the *inventory* object.

2. Although two components are being iterated over in the outer loop (*item* and *price*), only *item* is being used in the inner loop (*price* is never referenced, and thus its value is discarded with each iteration of the loop).

```
In [ ]:
## Code 7.3.4 ##

# creating a string object
my_string = 'string'

# Looping over a string
for letter in my_string:
    print(letter)
```

3. The inner loop inherits each element of *item*, one at a time, and iterates over each of its strings, letter by letter (remember, the inner loop only receives one string from *item* at a time). Once it has finished iterating over each letter in a string from *item*, the inner loop finishes running.

4. The outer loop iterates over the next item from *inventory*, passes *item* to the inner loop, and the inner loop prints out each individual letter in *item*. This process continues until the outer loop has nothing left to iterate over.

Note: For a further example of iterating over strings, see *Code 7.3.4*.

7.4 Enhancing *for* loops with Conditional Statements

The weekend is coming and the shop owner would like to hold a store-wide sale, offering each item at 90% of its original price, rounded to the nearest whole number. The shop owner could calculate the sale price for each individual item line-by-line, but this has several disadvantages, namely:

1. The code would be hard to update if the discount level changed (lack of replicability).

2. If new items were added to the inventory, the shop owner would need to write more lines of code (lack of extensibility).

3. Compared to coding a loop, individually discounting each item would take longer and be more susceptible to bugs (poor coding hygiene).

Let's also imagine that instead of only 20 items, the shop owner had a much larger inventory, making it infeasible to code discounts item-by-item. Given these disadvantages, the shop owner writes the **for loop** in *Code 7.4.1*. However, due to the missing values in the list, the code throws an error. This is because Python does not know what to do when encountering a string being multiplied by a float. Thus, the list items labeled as *'missing'* are creating a problem.

In [ ]:

```
## Code 7.4.1 ##

# Note: Code will throw an error.

# empty list (to be filled in the loop)
weekend_sale_prices = []

# loop to create prices for the weekend sale
for price in inventory[:]:
    sale_price = int(price * 0.90)
    weekend_sale_prices.append(sale_price)

# printing the prices for the weekend sale
print(weekend_sale_prices)
```

The shop owner needs to come up with a strategy for imputing the missing prices. Given the owner has domain knowledge of the retail industry, as well as the prices of other items in the shop, she decides that each missing price should have a value of 100. The code snippet below exhibits an **if** statement to solve the challenge of imputing the *'missing'* prices. While this is a good approach, it suffers from one important drawback: the imputation value for price is being hard coded.

```
for price in inventory:

    if price == 'missing':
        price = 100

    sale_price = round(price * 0.90, 0)
    print(sale_price)
```

Hard Coding v. Soft Coding

Hard coding is the process of writing code that contains static syntax (i.e. using *hard* numbers instead of objects). It may not make a huge difference in the code snippet above, but it can create several frustrations in larger codes. For example, let's imagine we were using the imputed value of price (*price = 100*) in more than one place in our code. If it was later decided that missing prices should be imputed at a value of 200, we would need to be very careful to ensure that we update every *price = 100* throughout our script. Additionally, this change may take a long time to code, test, and debug.

Generally, a good practice is to **soft code** whenever possible. **Soft coding** is the process of writing code utilizing objects instead of static syntax. This allows programmers to declare an object in one place and reuse its value(s) throughout their code. If a requirement came in to change an object's value (i.e. the imputation value for price), this change would only need to be made in one place. Below is a modified code snippet that utilizes soft coding.

```
imputation_price = 100

for price in inventory:

    if price == 'missing':
        price = imputation_price

    sale_price = round(price * 0.90, 0)
    print(sale_price)
```

Code 7.4.2 utilizes a conditional statement to check whether a price is missing, and then utilizes a soft-coded value to impute such anomalies. Once missing prices have been imputed, sale prices are calculated by multiplying the original price by *0.90*. Sale prices are then stored in a new list object (*enhanced_inventory*). The final **for loop** prints *enhanced_inventory* item-by-item. As can be observed from the output of the final loop, sale prices have successfully been processed without error.

In [ ]:

```
## Code 7.4.2 ##

# adapted from Code 7.3.1

# creating a new inventory list to include sale price
enhanced_inventory = []

# soft coding imputation value for price
imputed_price = 100

# adding a conditional to the for loop
for item, price in inventory:

    # setting imputation for missing price
    if price == 'missing':
        price = imputed_price

    # setting sale price
    sale_price = round(price * 0.90, 0)

    # adding item, price, and sale_price to enhanced_inventory list
    enhanced_inventory.extend([[item, price, sale_price]])

# printing enhanced inventory
for info in enhanced_inventory:
    print(info)
```

7.5 Adding More Depth with Nested Conditionals

As can be imagined, imputing all missing prices with the same value may not be the best approach. In other words, this strategy may be too general, and it may be more appropriate to impute based on more detailed characteristics of the data. If we can find a trend in the given information, we may feel more confident that our imputed prices more accurately reflect the true prices of each item in our inventory. Given that we currently have no way of knowing what the true value of our missing prices should be, we are unable to measure our degree of imputation accuracy. This challenge will be covered in more detail in a later chapter, but for now, we can run some basic statistics in order to better understand our pricing scheme.

Take a look at the information (i.e. the data) in *Code 7.4.2*. Do you notice any trends on which we can use to modify our imputation strategy? An open coding block is available below in the event that you would like to make any calculations. After your analysis, click *Show Solution* to see which trend we will be utilizing to enhance our imputation strategy.

In [ ]:

```
## Code 7.5.1 ##

# open coding block
```

Show Solution

As the inventory list was small, no explicit calculations were necessary to arrive at the aforementioned trend. Every blouse is more expensive than every skirt with no exceptions. However, it will rarely be this straightforward in real-world data analysis. Thus, it is essential to confirm such a finding numerically.

Calculating (Arithmetic) Means from List Items
Python's numpy (https://docs.scipy.org/doc/numpy/user/index.html) package is packed with useful methods to make calculating descriptive statistics (https://en.wikipedia.org/wiki/Descriptive_statistics) quite simple. In the case of the (arithmetic) mean, *numpy* contains a method that is appropriately named **mean()**. In order to access it, we first need to import *numpy*. Conventionally, *numpy* is imported as *np*.

```
import numpy as np

help(np.mean)
```

Code 7.5.2 utilizes *numpy's* **mean()** method, as well as a conditional statement to identify whether or not an item is a skirt or a blouse:

In [ ]:

```
## Code 7.5.2 ##

import numpy as np

# creating empty lists for blouses and skirts
blouse_prices = []
skirt_prices  = []

for item, price, sale_price in enhanced_inventory:
    if 'blouse' in item:
        blouse_prices.append([price])

    elif 'skirt' in item:
        skirt_prices.append([price])

    else:
        print('Unidentified inventory.')

# calculating (arithmetic) means
blouse_avg = np.mean(blouse_prices)
skirt_avg  = np.mean(skirt_prices)

print(f"""
Arithmetic Averages:
Avg. Blouse Price: {blouse_avg}
Avg. Skirt Price:  {skirt_avg}
""")
```

The code block above can be broken down as follows:

Line	Interpretation
3	importing numpy as np
6-7	creating empty lists to separately store blouse and skirt prices
9-17	for loop to append the *blouse_prices* and *skirt_prices* lists
20-21	using *np.mean* to calculate (arithmetic) average prices
23-27	printing the results

Code 7.5.2 clearly shows a difference between the (arithmetic) average price of blouses and skirts. This indicates that the shop owner's imputation strategy may be improved with the use of such information. Such an enhancement has been performed in *Code 7.5.3*.

Note: The term *mean* (i.e. average) is very general and can refer to a number of different statistics, the goal of each being to accurately identify a central point in the data. The term arithmetic mean (https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html) specifically refers to the *'average'* that calculated by taking the sum of each item in an array (i.e. a list, series, or column of data) and dividing it by the number of items in the array. This is likely the *'average'* you learned in grade school, along with the *median* and *mode*. From this point forward, the term *mean* will refer to the arithmetic mean.

In [ ]:

```python
## Code 7.5.3 ##

import numpy as np

# creating empty lists for blouses and skirts
blouse_prices = []
skirt_prices  = []

# returning to the original inventory list
for item, price in inventory:

    if price != 'missing':
        if 'blouse' in item:
            blouse_prices.append([price])

        elif 'skirt' in item:
            skirt_prices.append([price])

        else:
            print('Unidentified inventory.')


# soft coding imputation prices
blouse_imputation = round(np.mean(blouse_prices), 2)
skirt_imputation  = round(np.mean(skirt_prices), 2)

# creating a new inventory list to include sale price
enhanced_inventory = []

# returning to the original inventory list
for item, price in inventory:

    # setting imputation for missing price
    if price == 'missing' and 'blouse' in item:
        price = blouse_imputation

    elif price == 'missing' and 'skirt' in item:
        price = skirt_imputation

    # setting sale price
    sale_price = round(price * 0.90, 0)

    # adding item, price, and sale_price to enhanced_inventory list
    enhanced_inventory.extend([[item, price, sale_price]])

# printing enhanced inventory
for info in enhanced_inventory:
    print(info)
```

Finally, the shop owner has arrived at appropriate sale prices for her inventory. As a final step, let's utilize our knowledge of f-strings to create a flier to help promote the weekend sale. *Code 7.5.4* is an open coding block designated for this purpose.

In [ ]:

```
## Code 7.5.4 ##

# open coding block
```

Show Solution

7.6 Summary

Although **for loops** may seem more confusing than our previous content, they are a very important concept to master. They are an incredibly valuable alternative to copy/pasting and trying to iterate line-by-line. The next chapter will continue the discussion of iteration by diving into **while loops**.

```
 __    __              _                        _
/ / /\ \ \__ _ _   _  | |_ ___     __ _  ___   / \
\ \/  \/ / _` | | | | | __/ _ \   / _` |/ _ \ /  /
 \  /\  / (_| | |_| | | || (_) | | (_| | (_) /\_/
  \/  \/ \__,_|\__, |  \__\___/   \__, |\___/\/
               |___/              |___/
```

Chapter 8: while Loops and Making Assumptions

As mentioned in **Chapter 7: for Loops and Basic Data Manipulation**, a **while loop** can be thought of as an extension of a conditional statement. More specifically, a **while loop** is like a hybrid between a conditional statement and a for loop. They are similar to for loops in the sense that they iterate. As with conditional statements, as long as the condition(s) specified is met, a **while loop** will continue running. This powerful coding structure is incredibly useful, as it enables programmers to accomplish a wide array of tasks. For example, they can be used to:

- take a list of all the students in a cohort and break them into teams of four
- allow a user to reenter their password if their first attempt was invalid
- calculate how many NBA seasons it took for Michael Jordan to score 25,000 points

In addition to the fundamentals of **while loops**, this chapter will also cover some syntax that is also useful in other coding structures, namely for loops and user-defined functions (covered in the next chapter). For example, in many situations a programmer needs to **break** out of the body of a loop before an iteration has finished. At other times, it may be necessary to skip over an item in an iterable and **continue** the loop by iterating on the next item. As you may have guessed, the syntaxes that enable such functionality are break (https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops) and continue (https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops). Both are essential in building a solid foundation in Python.

Note: As mentioned, **while loops** will run until a specified condition is no longer met. If we are not careful, we may get stuck in a loop that does not stop running. Before moving forward, you may want to go back to **Chapter 1: Setting Up for Success** to refresh on what to do in such a situation. As an example, notice the difference between *Codes 8.0.1* and *8.0.2*. Without the final line in the body of the **while loop**, the loop's condition is always met and the code runs on into infinity (*Code 8.0.2*). **Make sure you understand how to interrupt your kernel before running this code!**

In [ ]:

```
## Code 8.0.1 ##

# adapted from Code 7.1.2

# declaring x
x = 5

# while loop
while x > 0:
    print(x)
    x -= 1
```

In [ ]:

```
## Code 8.0.2 ##

#!# WARNING! The while loop will run foreve

# declaring x
x = 5

# while loop with final line commented out
while x > 0:
    print(x)
#   x -= 1
```

8.1 The Fundamentals of *while loops*

As mentioned in the introduction to this chapter, **while loops** can be utilized to accomplish tasks such as taking a list of all the students in a cohort and breaking them into teams of four. To exemplify this, let's assume our cohort had a total of eight students, as in *Code 8.1.1*. Let's also assume that teams need to be random and that no student is allowed to be on more than one team. In order to accomplish our task, we need to develop a code that:

- randomly selects a student and puts them on a team
- removes the student from the selection pool so that they do not end up on more than one team (sampling without replacement)
- fills each team to exactly four members

- keeps iterating until every student has been placed on a team

We will start by importing the random package so that we can utilize the method [choice (https://docs.python.org/3/library/random.html#functions-for-sequences)](https://docs.python.org/3/library/random.html#functions-for-sequences) in our code. This method is designed to randomly choose one element from a population. Given our requirements, a drawback to using this method is that it samples with replacement, meaning there is a chance that a student will be selected more than once.

Unfortunately, there is no optional argument in **random.choice** that enables sampling without replacement. However, the problem of selecting a student more than once can be avoided through a clever design of our code. What if, after a student is selected in an iteration of our loop, they were removed from the *students* list? This way, there would be no chance of the same student being selected in the next iteration of the loop, as the student would no longer be available for selection. This can be achieved by applying the **.remove()** method on *students*.

Also note that at first, the condition of the **while loop** may seem unclear. Essentially, this condition is telling Python to keep looping as long as there is something to iterate over in *students*. In fact, the syntax:

```
while students:
```

Could also be written as:

```
while len(students)>0:
```

Both syntaxes will achieve the same result, although the first is much more common. Also, note that once the loop has iterated over every item in *students*, the condition that there is still something left to iterate over evaluates to *False* and the loop stops running.

Finally, each time *Code 8.1.1* is run, different teams are created. This does not violate any of the requirements for our task, but it makes it difficult to replicate our results. If our goal was to avoid this, we could employ the use of **random.seed()**, as explained in **Chapter 4: Numbers, Comparisons, and Randomness**.

In [ ]:

```
## Code 8.1.1 ##

# importing random
import random

# list of people
students = ['Neil', 'Alex', 'Andy', 'Ross',
            'Miles', 'Jon', 'Jed', 'Dustin

team_1 = []
team_2 = []

# while loop
while students:
    person = random.choice(students)

    if len(team_1) < 4:
        team_1.append(person)

    elif len(team_1) >= 4:
        team_2.append(person)

    else:
        print("Something went wrong.")

    # removing so students don't get repeat
    students.remove(person)

# printing teams
print(team_1)
print(team_2)
```

8.2 *while True* and *break*

Sometimes, it makes sense to allow a loop to run on into infinity. This technique becomes very useful in situations such as developing code to allow a user to reenter their password if their first attempt was invalid (i.e. user input does not match the stored password). Using a stand-alone conditional statement such as the one below will only permit a user to make a single input attempt, as after evaluating whether or not *pwd_attempt* and *password* match, Python will move on to the next line of code formatted at Column 0.

```
if pwd_attempt == password:
    [do something]

elif pwd_attempt != password:
    [do something else]

else:
    [catch bugs]
```

However, by wrapping a **while True** loop around this code, a user will have an infinite number of tries to correctly input their password. To reiterate, **a while loop will run until a condition is no longer met** (i.e. the condition evaluates to *False*). Since by default the condition of the loop is set to *True*, the loop will run indefinitely as there is no syntax to change its evaluation to *False*. Given that our goal is to stop looping when a user enters the correct password, we need to tell Python to **break** out of the loop when this happens. This can be coded as follows:

```
while True:
    if pwd_attempt == password:
        break

    elif pwd_attempt != password:
        [do something else]

    else:
        [catch bugs]
```

Code 8.2.1 exemplifies the use of these structures to create a basic check to see if a user's input matches a stored password.

In [ ]:

```python
## Code 8.2.1 ##

# creating a password
password = 'please open the door'


# while True
while True:
    pwd_attempt = input("Please enter your password.\n> ")

    if pwd_attempt == password:
        print('\nThat is correct. You may enter.\n')
        input('< Press enter to continue. >\n')
        break

    elif pwd_attempt != password:
        print('\nThat is not the correct password.\n')
        input('< Press enter to try again. >\n')

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

8.3 *while loops* with specific conditions

Although we have met the requirement of allowing a user to have multiple password entry attempts, allowing unlimited attempts is dangerous as it would be susceptible to malicious attacks on a user's account. Thus, it would be wise to modify our code so that it only allows a user a specific number of tries. In order to achieve this, the condition of the original **while loop**:

```
while True:
    [do something]
```

can be altered using a comparison operator (covered in **Chapter 4: Numbers, Comparisons, and Randomness**). In other words, the *'True'* in **while True** can be replaced with something such as *'password_attempts > 0'*. For example, if a user were to be allowed only three attempts, *Code 8.2.1* could be modified to include an object with a stored value of three. Then, the loop could diminish this value by one each time a user inputs an incorrect password. This has been done in *Code 8.3.1*.

In [ ]:

```python
## Code 8.3.1 ##

# adapted from Code 8.2.1

# creating a password
password = 'please open the door'

# setting password attempts to three
password_attempts = 3

# exiting the loop after three attempts
while password_attempts > 0:
    pwd_attempt = input("Please enter your password.\n> ")

    if pwd_attempt == password:
        print('\nThat is correct. You may enter.\n')
        input('< Press enter to continue. >\n')
        break

    # diminishing password_attempts
    elif pwd_attempt != password:
        password_attempts -= 1

        print(f"""
That is not the correct password.
You have {password_attempts} attempt(s) remaining.\n""")

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

Looping over Michael Jordan's NBA Career

To further exemplify **while loops** with specific conditions, let's turn our attention to basketball legend Michael Jordan (https://www.basketball-reference.com/players/j/jordami01.html). According to basketball-reference.com (https://www.basketball-reference.com/), Jordan played in over 1,000 NBA games across 15 seasons, and in 11 of those seasons, he led the league in total points scored. He also made 14 All-Star appearances and won the league MVP award 5 times. Additionally, he spent the *'93-'94* season playing Minor League Baseball for the Birmingham Barons (labeled as *'DNP'*), and went into retirement for three seasons between 1998 and 2001 (labeled as *'Retired'*). Each sublist in *Code 8.3.2* represents total points scored for each season in Jordan's NBA career with the following format:

```
[ SEASON, TOTAL POINTS SCORED, LEAGUE SCORING LEADER Y/N ]
```

First, let's load this information into our working environment and utilize it to determine how many seasons it took for Jordan to surpass 25,000 points.

In [ ]:

```
## Code 8.3.2 ##

jordan_stats = [["'84-'85", 2313, 'Y'],
                ["'85-'86", 408,  'N'],
                ["'86-'87", 3041, 'Y'],
                ["'87-'88", 2868, 'Y'],
                ["'88-'89", 2633, 'Y'],
                ["'89-'90", 2753, 'Y'],
                ["'90-'91", 2580, 'Y'],
                ["'91-'92", 2404, 'Y'],
                ["'92-'93", 2541, 'Y'],
                ["'93-'94", 'DNP', 'DNP'],
                ["'94-'95", 457,  'N'],
                ["'95-'96", 2491, 'Y'],
                ["'96-'97", 2431, 'Y'],
                ["'97-'98", 2357, 'Y'],
                ["'98-'99", 'Retired', 'Retired'],
                ["'99-'00", 'Retired', 'Retired'],
                ["'00-'01", 'Retired', 'Retired'],
                ["'01-'02", 1375, 'N'],
                ["'02-'03", 1640, 'N']]
```

The long solution to this task would be to sum points in each season one-by-one until we reached the desired amount:

```
jordan_stats[0][1] + jordan_stats[1][1] + jordan_stats[2][1] ...
```

However, this is improper for a number of reasons:

1. This is a tedious copy/paste approach. Copy/pasting is inefficient and prone to bugs (a programmer would need to remember to update the code after a new line has been pasted). Forgetting to update just once would lead to a bug that may go unnoticed (the code will likely run without throwing an error).

2. What if we wanted to change our code so that it calculated something else, such as how many points Jordan scored in seasons where he also led the league in scoring? This slight alteration to our requirements would require us to significantly rewrite our code.

3. Let's say our requirements changed again and we wanted to calculate Michael Jordan's total career points. However, let's also say Jordan wasn't retired. In other words, what if new lists were continually being added to *jordan_stats*? This would require continual updates to our copy/paste solution.

Given the above, it is highly beneficial to use loops. First, this would significantly alleviate concerns of accidental copy/paste bugs. Second, if our requirements changed, we may be able to take advantage of other coding structures, such as a conditional statement in the body of the loop. Finally, if the *jordan_stats* list were continually growing, our loop could adjust to its new length. In other words, it makes no difference if Jordan keeps playing as the loop could be programmed to continue iterating until it has run out of things to iterate over. Before developing our code, however, let's take a moment to discuss the assumptions involved in our task.

8.4 Making Assumptions about Data

Before getting started, we should clean up our data as this would simplify the coding for our analysis. In seasons where Jordan did not play in the NBA, total points scored is coded as a string. Given our goal, it seems unfair to consider these seasons in our calculation, and thus they should be removed. In this, we are making an assumption, and **all assumptions should be documented.** Another analyst could approach the same problem and feel it is more appropriate to include seasons where Jordan did not play in the NBA. The rationale for such a decision could be argued from a number of viewpoints, such as:

- This calculation should be made as the total number of seasons between Jordan's first and the one in which he surpassed 25,000 points.
- Jordan choose not to play basketball in the '93-'94 season even though he was capable of doing so.
- Even though Jordan retired, he came back from retirement. Therefore, those seasons should be counted.

The value of an analysis stretches far beyond the sciences, and in many cases, it is far more of an art. Given this, each analysis requires substantial thought regarding the problem at hand, and such considerations lead to various assumptions that make each analytical perspective unique. Without documentation, it is incredibly difficult to understand an analyst's thought process. Moreover, it becomes challenging to trust the findings and actionable insights that resulted from such work. Assumptions need to be documented, whether embedded within scripts, in a document or project management tool, or even in a notebook that you keep next to you while you are working. Although this takes time up front, it will save you countless hours in the long run, especially when working with other analysts or reviewing an analysis that you conducted several months ago.

Code 8.4.1 is designed to create a new list of lists that excludes seasons where Jordan did not play in the NBA. The aforementioned assumption has been embedded at the top of the code block as a triple-quoted string.

In [ ]:

```
## Code 8.4.1 ##

"""
Assumptions:
    Calculations should only include seasons where Jordan played.
"""

# creating an empty list
jordan_stats_2 = []


# writing a loop on original stats lists
for season, points, lead_scorer in jordan_stats:

    # appending new list if points is an integer
    if type(points) == int:
        jordan_stats_2.append([season, points, lead_scorer])


# writing a loop to print new stats lists one-by-one
for stats in jordan_stats_2:
    print(stats)
```

Developing our Solution
In order to calculate how many seasons it took for Jordan to score more than 25,000 points, we first need to develop a method to iterate over total points in each list of *jordan_stats_2*. This can be accomplished with the use of a for loop, as covered in **Chapter 7 - for Loops and Basic Data Manipulation**. Our goal is to stop iteration after accumulating a certain number of total points, and for this we have multiple options. More specifically, we may choose to fit out the body of the for loop with either a conditional statement or a **while loop**, as both could solve this task. Since conditional statements were already covered in **Chapter 6 - Conditional Statements and Controlling Input**, we will build a solution using this method first. *Code 8.4.2* has been left open for this task. Make sure to:

- iterate over *jordan_stats*
- stop iteration after accumulating at least 25,000 points
- use a conditional statement inside the body of a for loop (don't use a **while loop**)
- avoid soft coding wherever feasible

Below is a skeleton (i.e. an outline) to help get you started:

```
for _______ in jordan_stats_2:

    if ________:
```

After developing a conditional statement solution, compare your results to the sample solution. When you are ready, develop a solution using a **while loop** in *Code 8.4.3*, which has also been left open. Below is another skeleton to help get you started:

```
for _______ in jordan_stats_2:

    while ________:
```

In [ ]:

```
## Code 8.4.2 ##

# open coding block (for loop + conditional statement)
```

Show Solution

In [ ]:

```
## Code 8.4.3 ##

# open coding block (for loop + while loop)
```

Show Solution

Benefits of using a *while* loop

The bodies of *Codes 8.4.2* and *8.4.3* look very similar. As mentioned earlier in this chapter: **a while loop can be thought of as an extension of a conditional statement.** Also, as expected, both codes lead to the same result. The solution utilizing the **while loop**, however, uses less lines of code. Additionally, this solution appears to be easier to extend if there was a change in our requirements. For example, let's assume that it was decided that points should only be accumulated in seasons where Jordan led the league in scoring. This change is very easy to implement given the **while loop** solution, as only a single line of code needs to be modified:

```
while total_points < point_limit and lead_scorer == 'Y':
```

This has been implemented in *Code 8.4.4*. Notice how the assumptions and print statement have also been updated.

In [ ]:

```
## Code 8.4.4 ##

# adapted from Code 8.4.3

"""
Assumptions:
    Calculations should only include seasons where Jordan:
      - played
      - led the league points
"""

# declaring objects
total_points  = 0
total_seasons = 0
point_limit   = 25000 # avoiding soft coding


# writing the outer loop
for season, points, lead_scorer in jordan_stats_2:

    # MODIFIED inner loop
    while total_points < point_limit and lead_scorer == 'Y':
        total_points  += points
        total_seasons += 1
        break

# MODIFIED the print statement
print(f"""
{'*' * 40}

In seasons where he led the league in scoring,
Jordan surpassed {point_limit} points after {total_seasons}
seasons (scoring {total_points}).

{'*' * 40}
""")
```

Adjusting the Conditional Statement Solution
Adjusting the conditional statement solution is a bit more daunting, as the same modification would result in the *else* clause running when either of the two conditions is not met. This can be observed in *Code 8.4.5*.

In [ ]:

```python
## Code 8.4.5 ##

# adapted from Code 8.4.2

# declaring objects
total_points  = 0
total_seasons = 0
point_limit   = 25000 # avoiding soft coding

# writing the loop
for season, points, lead_scorer in jordan_stats_2:

    # MODIFIED conditional
    if total_points < point_limit and lead_scorer == 'Y':
        total_points  += points
        total_seasons += 1
        print('All is well!') # added print statement for clarity

    elif total_points >= point_limit:
        break

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

By rewriting our code to include a nested conditional statement, we can attain the functionality we desire. If you need a refresher on nested conditionals, please see **Chapter 6: Conditional Statements and Controlling Input**. *Code 8.4.6* includes this change and also introduces a new syntax: **continue**. The role of this syntax is to terminate the current iteration of a loop and move on to the next iteration. In other words, it stops what's currently happening and *continues* by starting the next iteration. Since we expect *lead_scorer* to be equal to *'N'* in some iterations, it is a good practice to write an *elif* statement to **continue** when this is the case. If we do not do this, the *else* clause will run, falsely indicating that something went wrong.

In [ ]:

```
## Code 8.4.6 ##

# adapted from Code 8.4.5

"""
Assumptions:
    Calculations should only include seasons where Jordan:
      - played
      - led the league points
"""

# declaring objects
total_points  = 0
total_seasons = 0
point_limit   = 25000 # avoiding soft coding

# writing the loop
for season, points, lead_scorer in jordan_stats_2:

    # writing the conditional
    if total_points < point_limit:

        if lead_scorer == 'Y':
            total_points  += points
            total_seasons += 1

        # applying continue
        elif lead_scorer == 'N':
            continue

        else:
            print('Something went wrong.')

    elif total_points >= point_limit:
        break

    else:
        print('Something went wrong.')


# printing the results
print(f"""
{'*' * 40}

In seasons where he led the league in scoring,
Jordan surpassed {point_limit} points after {total_seasons}
seasons (scoring {total_points}).

{'*' * 40}
""")
```

As expected, the results when using a conditional statement are the same as when using a **while** loop. As with before, the **while** loop uses less lines of code. One drawback, however, is that it does not contain an *else* clause to help catch bugs. This is because *else* can only be applied to conditional statements. Thus, we need to be diligent in testing our code to ensure it does what it is intended to do. Additionally, if there are potential errors in our code due to things that should have been controlled for, they may go undetected.

To illustrate, let's assume we wanted to run our code on every player in the history of the NBA. One challenge we may run into is that what we call the NBA today is actually the result of a merger that took place in 1976. From 1967-1976, two nationally-scoped basketball leagues existed in the United States (the National Basketball Association and the American Basketball Association (https://en.wikipedia.org/wiki/American_Basketball_Association)). Thus, each season played within these years has two scoring leaders. If our code was designed to allow only one scoring leader per season, it may throw an error. We may also run into a situation where a player was traded and thus has two records for a given season. This may prove to be a challenge when calculating how many seasons the player took to attain a certain number of points. The good news is that Python contains more advanced coding syntaxes that are incredibly useful in such situations (covered in the next chapter).

8.4 Summary

A **while loop** is a useful coding structure that allows programmers to accomplish several tasks. They can be thought of as a hybrid between a conditional statement and a for loop in the sense that they: 1) iterate, and 2) run until a condition or set of conditions is no longer met. The syntax **break** stops a loop's from iterating further, and the syntax **continue** ends a loop's current iteration and moves on to the next one.

```
               _            __  __     _                          _
  /\  /\__ _| |_ ___     ___  / _|/ _|  | |_ ___    _   _  ___  _   _  / \
 / /_/ / _` | __/ __|   / _ \| |_| |_   | __/ _ \  | | | |/ _ \| | | |/  /
/ __  / (_| | |_\__ \  | (_) |  _|  _|  | || (_) | | |_| | (_) | |_| /\_/
\/ /_/ \__,_|\__|___/   \___/|_| |_|     \__\___/   \__, |\___/ \__,_\/
                                                    |___/
```

Chapter 9: User-Defined Functions and Exception Handling

Python is an extraordinarily popular language for many reasons, one of them being that it has a strong open-source community. In general, open source (https://opensource.com/resources/what-open-source) projects are community-driven, publicly available, and free of charge. More often than not, our analytical endeavors will rely on open source packages such as numpy (https://docs.scipy.org/doc/numpy/user/index.html), pandas (https://pandas.pydata.org/pandas-docs/stable/), scikit-learn (https://scikit-learn.org/stable/documentation.html), statsmodels (https://www.statsmodels.org/stable/index.html), matplotlib (https://matplotlib.org/3.1.1/contents.html), seaborn (https://seaborn.pydata.org/), and many others depending on our task at hand. Each of these packages contains functions that were designed for a specific purpose, and most contain optional or variable arguments to allow a certain level of flexibility and customization in their use.

Before diving into the aforementioned packages, it is important to have a good understanding of the inner workings of functions. Comprehending how to write, document, test, and exception handle user-defined functions will greatly increase your ability to benefit from functions available in the open source community. This will not only help in utilizing publicly-available resources to their fullest extent, but will also help in identifying when a developer may have cut corners. Every so often, a package is released that contains functions that are unstable or that only work in a limited context. Generally, such packages can be identified by the quality of their documentation. The content of this chapter emcompasses these aspects, as well as best practices when naming functions, variable scope, and common error types.

9.1 The Function Framework

Functions follow a similar framework to conditional statements and loops in that their first line starts with a designated syntax and ends with a semicolon (:). Additionally, anything inside a function's body must be indented. Unlike other coding structures covered thus far, functions need to be defined before they can be run. In other words, you need to give Python the blueprints of your function before you will be able to use it. You can think of this like working with packages in that they must be imported before their methods become available.

Defining a Function
The syntax **def** lets Python know that you are defining a function. After this, the function must be given a unique name, followed by a set of parenthesis *( )* and a semicolon.

```
def FUNCTION_NAME(ARGUMENTS):
    FUNCTION BODY
```

User-defined functions can have any nonnegative number of arguments (zero or more). However, mandatory arguments must come before optional arguments, and optional arguments must come before variable arguments. If you are a bit rusty on your argument types, return to **Chapter 1: Setting Up for Success** for a refresher.

Let's begin by writing a very simple function that has no arguments. This function, which has been made available in *Code 9.1.1*, is designed to wish a user a happy birthday. Notice that running the definition of the function does not generate any output. Instead, *Code 9.1.1* simply loads the function so that Python knows what to do every time it is called. When *Code 9.1.2* is run, Python reads the *bday* function and outputs the birthday message.

In [ ]:

```
## Code 9.1.1 ##

# defining a function
def bday():
    print('Happy Birthday!')
```

It is very important that user-defined functions are given a UNIQUE name.

If not, Python will realize that another function exists with the same name and overwrite it.

In [ ]:

```
## Code 9.1.2 ##

# running the function
bday()
```

This should be avoided unless you have a very good reason for doing so. Other than overwriting a function in your working environment, another massive danger to this practice is that it will also affect anyone that uses your function. This causes a big headache when a user needs both your function and the one you overwrote.

To illustrate, let's take Python's built-in function abs() (https://docs.python.org/3/library/functions.html#abs), which takes a number and returns its absolute value. *Code 9.1.3* uses this built-in to take the absolute value of negative seven. As expected, the function returns positive seven. However, in *Code 9.1.4*, *abs( )* is being overwritten as a user-defined function with the same name. After running *Code 9.1.4*, try running *Code 9.1.3* again. As expected, this code block now throws an error.

In [ ]:

```
## Code 9.1.3 ##

# built-in abs
abs(-7)
```

In [ ]:

```
## Code 9.1.4 ##

# overwriting abs
def abs():
    print('Happy Birthday')

abs()
```

Restarting the Python Kernel
As mentioned earlier, overwriting existing functions should be avoided as much as possible. Sometimes, this may happen by accident. In such situations, you can reset your Python kernel, restoring all built-in functions to their default forms. In Jupyter notebook, this can be accomplished by navigating to **Kernel > Restart**.

In [ ]:

```
## Code 9.1.5 ##

# Code to kill a kernel
import os
os._exit(00)
```

Note that upon restarting, all of your declared objects and user-defined functions will be wiped from your working environment (you can add them back by rerunning their codes). Additionally, you can kill your kernel by running *Code 9.1.5*. How did I discover this code? At one point in my Python journey, I needed to kill my kernel and stumbled upon this technique in a StackOverflow thread (https://stackoverflow.com/questions/37751120/restart-ipython-kernel-with-a-command-from-a-cell). As mentioned in Chapter 1:

As long as you are not one of the world's most advanced Python coders, someone has already experienced and solved your problem.

There is no shame in needing to look things up. In fact, many coders virtually live on sites like StackOverflow (https://stackoverflow.com/) in the early days of their coding journeys, and I have yet to meet a programmer that does not at least occasionally need to make queries. The coding community is there to help you, so make sure to take advantage and give back by contributing your knowledge when you get to that point.

9.2 Arguments and Environments

When we want more out of our function, arguments can be added to user-defined functions by placing variables within the parenthesis right after the function's name. As mentioned, the order in which arguments must be specified is as follows:

1. mandatory arguments (no default value)
2. optional arguments (contains default value)
3. variable arguments (vary in length)

Let's move forward by defining a function that takes one argument. *Code 9.2.1* is a user-defined function that takes one mandatory argument and squares its value. To keep things simple, it has been named *square*. Judging by the body of this function, one may expect *Code 9.2.1* to output the result of three raised to the power of two. However, running this code block produces no output for a very important reason: we haven't told the function to **return** anything.

Programs do what they are programmed to do.

Computers need to be given instructions in order to perform tasks. In the case of *Code 9.2.1*, the function did exactly what we told it to do, and that is why it didn't produce any output. This can be addressed by modifying the function to include a **return** statement.

The Limited World of a Function
It is important to understand why **return** statements are required to get something back from a function. To get there, think of your working environment as the entire planet on which all of your code lives. We will call this the **global environment**. Also, think of declaring an object as creating an living being on your planet, and this being is free to interact with other objects in the global environment. If another object is declared with the same name, it will cause conflict and the former object will be overwritten.

Now, think of the body of a function as its own island where its inhabitants have been secluded for so many generations that they are completely oblivious to anything that exists elsewhere. This is good because these objects have no way to adversely affect the

In [ ]:

```
## Code 9.2.1 ##

# defining a function
def square(x):
    x**2

# running the function
square(x = 3)
```

global environment (i.e. overwriting objects). We will call this island the function's environment. Oftentimes, we need something from the function's environment to interact with the global environment, and as the master of both environments, we have the ability to make this happen. In other words, at times we need something from the island to **return** to the global environment, and the syntax to accomplish this is, not surprisingly, named **return**.

In [ ]:

```
## Code 9.2.2 ##

# adapted from Code 9.2.1

# adding return statement
def square(x):
    return x**2

# running the function
square(x = 3)
```

A **return** statement has been added to *Code 9.2.2*, where the *square()* function is returning the results of a calculation. More commonly, such results are first stored within an object in the body of a function and the object is returned. This is occurring in *Code 9.2.3*, where the calculation results have been stored in the object *val*, which is referenced in the **return** statement. Both codes give the same result.

In [ ]:

```
## Code 9.2.3 ##

# adapted from Code 9.2.2

# adding return statement
def square(x):
    val = x**2
    return val

# running the function
square(x = 3)
```

Note that even though *val* was returned in *Code 9.2.3*, it is not actually available as an object in the global environment. This is Python's way of protecting the function's users from accidentally overwriting object names. For example, if the global environment already contained an object named *val*, Python protects this object from being overwritten. On the same token, any object in the global environment that shares the same name as an object in a function's environment will not be overwritten when the function is called, even if it contains a **return** statement. There is one exception to this rule, which will be covered later in this chapter. The results of *Code 9.2.4* confirm that *val* has not been defined globally.

In [ ]:

```
## Code 9.2.4 ##

print(val)
```

Storing Function Output

A common method for storing the returned output of a function is to declare it as an object in the global environment. The type of the declared object depends on what was returned from the function. In *Code 9.2.5*, since the *square()* function returned an integer, the object *' y '* is of this type.

In [ ]:

```
## Code 9.2.5 ##

# storing output as an object
y = square(x = 3)

# printing the results
print(y)
print(type(y))
```

Also note that once Python executes a **return** statement, it leaves the body of the function, even if another **return** statement is on the next line. However, a **return** statement can reference more than one thing (separated by commas). By default, a function will return a [tuple (https://docs.python.org/2/tutorial/datastructures.html#tu and-sequences)](https://docs.python.org/2/tutorial/datastructures.html#tu) if more than one object referenced in a **return** statement (*Code 9.2.6*). **Tuples** are very similar to lists, but are represented with curved brackets and are immutable (they cannot be changed). **Tuples** are very important in functional programming, but are beyond the scope of our current coding needs. For now, it is important to understand that indexing a **tuple** is no different than indexing a list.

Note: If you would like to return a list instead of a **tuple**, try wrapping square brackets around the objects in your **return** statement:

```
return [a, b]
```

In [ ]:

```
## Code 9.2.6 ##

# adapted from Code 9.2.3

# modifying return statement
def square(x):
    val = x**2
    return val, x

# running the function
y = square(x = 3)

# printing the results
print(y)
print(type(y))
```

9.3 Functions with Multiple Arguments

Notice in that in the codes in *Section 9.2*, the argument *' x '* was explicitly declared when the function was run. This is not required, but it is a good practice. This is especially valuable when a function has more than one argument. Without explicit declaration (i.e. *' x = 5 '* instead of just *' 5 '*), things can get very confusing. For example, functions such as the following would be far more difficult without explicitly declaring the values for each argument.

```
def personal_favorites(food, beverage, book, movie, place, color):
        print(f"""
        Your favorite food is {food}.
        Your favorite beverage is {beverage}.
        Your favorite book is {book}.
        Your favorite movie is {movie}.
        Your favorite place is {place}.
        Your favorite color is {color}.
        """)
```

From this point forward, this text will explicitly declare function arguments.

Setting Default Values with Optional Arguments
A good approach to writing functions is to use as few mandatory arguments as is reasonably feasible. This approach has a number of benefits, such as making a function easier to use and reducing the chance of errors due to an invalid argument input. Oftentimes, however, you will have the need to develop additional features so that your function can be used in a wider array of applications or to allow users with a certain level of customization. Such situations call for the use of **optional arguments**, or arguments that have a set default value. To illustrate, let's modify *Code 9.2.3* so that it takes any number to any power.

This modification can be accomplished by defining a second argument in the first line of the function, which has been done in *Code 9.3.1*. The second argument, *power*, has been made optional with a default value of two. When selecting the default value for an optional argument, it is a good practice to consider what users will commonly input if the argument was mandatory. Considering our original function was designed to square a number, defaulting power with a value of two seems appropriate.

In [ ]:

```
## Code 9.3.1 ##

# adapted from Code 9.2.3

# adding return statement
def power_up(x, power=2):
    val = x**power
    return val

# running the function
power_up(x = 3)
```

Overriding Defaults in Optional Arguments
Default values in optional arguments can be overridden by declaring a new **parameter** when the function is called (i.e. when it is run). In Python, **parameter** is another word for the value of an argument. In fact, most of Python's help files have a section labeled *Parameters* that explains what each argument does and indicates what types of inputs are acceptable.

In the final line of *Code 9.3.2*, the default value for *power* is being overridden. As expected, this outputs a different value than that of *Code 9.3.1*. Also note that overriding an optional argument does not change its default value. If we called *power_up* again without specifying a value for *power*, the default for this argument would be used.

In [ ]:

```
## Code 9.3.2 ##

# adapted from Code 9.2.3

# adding return statement
def power_up(x, power=2):
    val = x**power
    return val

# running the function
power_up(x = 3, power = 3)
```

9.4 Docstrings

Up to this point, we have created a total of three user-defined functions:

- *bday( )* - prints a birthday message
- *square(x)* - raises a number to the power of two
- *power_up(x, power=2)* - raises any number to any power

Each of these functions has been given a name that provides suggestion as to what the function does. This is a good start, but a critical element of explanation is missing: the docstring (https://www.python.org/dev/peps/pep-0257/#what-is-a-docstring). **Docstrings**, also known as documentation strings, do as their name implies: they provide documentation. In other words, their purpose is to help users understand what something does and how it should be used. They are a key component of many coding structures in Python, including functions, classes, modules, and packages, although we will only be discussing them in terms of functions in this chapter. At this point, you should have ample experience working with docstrings, as they are the text that appears when you call *help( )* on something.

Writing a Docstring
Coding a **docstring** for a function is incredibly simple: place a triple-quoted string on the first line of the function's body.

```
def FUNCTION_NAME(ARGUMENTS):
    """ DOCSTRING """

    FUNCTION BODY
```

As long as the **docstring** is on the first line, it will be referenced every time a user calls *help( )* on the function. Also, **docstrings** for functions are generally written concisely, consisting of only a single line explaining what the function does. However, there is no reason to restrict yourself from being thorough if you feel it will benefit users. At the time of this writing, the **docstring** for the method *pandas.DataFrame* is 72 lines long. Not only does the documentation provide details as to what the method does, but also offers a solid explanation of its parameters, outlines similar methods that have been designed to address slightly different needs, and provides usage examples. This **docstring** is shown below.

```
Two-dimensional size-mutable, potentially heterogeneous tabular data
structure with labeled axes (rows and columns). Arithmetic operations
align on both row and column labels. Can be thought of as a dict-like
container for Series objects. The primary pandas data structure.

Parameters
----------
data : ndarray (structured or homogeneous), Iterable, dict, or DataFrame
    Dict can contain Series, arrays, constants, or list-like objects

    .. versionchanged :: 0.23.0
       If data is a dict, column order follows insertion-order for
       Python 3.6 and later.

    .. versionchanged :: 0.25.0
       If data is a list of dicts, column order follows insertion-order
       Python 3.6 and later.

index : Index or array-like
    Index to use for resulting frame. Will default to RangeIndex if
    no indexing information part of input data and no index provided
columns : Index or array-like
    Column labels to use for resulting frame. Will default to
    RangeIndex (0, 1, 2, ..., n) if no column labels are provided
dtype : dtype, default None
    Data type to force. Only a single dtype is allowed. If None, infer
copy : boolean, default False
    Copy data from inputs. Only affects DataFrame / 2d ndarray input

See Also
--------
DataFrame.from_records : Constructor from tuples, also record arrays.
DataFrame.from_dict : From dicts of Series, arrays, or dicts.
DataFrame.from_items : From sequence of (key, value) pairs
    read_csv, pandas.read_table, pandas.read_clipboard.

Examples
--------
Constructing DataFrame from a dictionary.

>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4

Notice that the inferred dtype is int64.

>>> df.dtypes
col1    int64
col2    int64
```

```
dtype: object

To enforce a single dtype:

>>> df = pd.DataFrame(data=d, dtype=np.int8)
>>> df.dtypes
col1    int8
col2    int8
dtype: object

Constructing DataFrame from numpy ndarray:

>>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
...                    columns=['a', 'b', 'c'])
>>> df2
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

The *bday( )* function has been modified to include a **docstring** (*Code 9.4.1*). It may seem trivial to write a **docstring** for such a simple function that has no arguments and offers no customization. However, all coding structures that support **docstrings** should have them, regardless of how simple they may be. They are easy to set up and can immensely speed up a user's learning curve. Additionally, poor quality documentation may indicate that corners have been cut. When reading *help( )* files, you should always look for:

- an explanation as to what the function is supposed to do
- a list of parameters and how to apply them
- usage examples

In [ ]:

```
## Code 9.4.1 ##

# adapted from Code 9.1.1

# defining a function with a docstring
def bday():
    """Function prints 'Happy Birthday!'""'
    print('Happy Birthday!')

# calling help() on the function
help(bday)
```

If these are not available, search the **docstring** for a web link that leads to these things. If no such link exists, it may be best to find an alternative tool. *Codes 9.4.2* and *9.4.3* have been left open so that you may create **docstrings** for the *square( )* and *power_up( )* functions, respectively. As a good practice, make sure to include an explanation of each function's parameters and a usage example.

In [ ]:

```python
## Code 9.4.2 ##

# open coding block

# adapted from Code 9.2.3

def square(x):
    val = x**2
    return val

help(square)
```

Show Solution

In [ ]:

```python
## Code 9.4.3 ##

# open coding block

# adapted from Code 9.3.1

def power_up(x, power=2):
    val = x**power
    return val

help(power_up)
```

Show Solution

9.5 Functions with Variable Arguments

Recall from **Chapter 1: Learn This Before Learning Else** that **args* and ***kwargs* allow for arguments of varying length. In other words, these syntaxes are very useful when there is need to allow flexibility in the amount of inputs a user can specify. A good example of such a need is a shopping list. To keep things simple, let's assume that the requirements for our *shopping_list* function are to take items as arguments and then print them one-by-one. Also, common knowledge implies that different users will have shopping lists of different lengths. Therefore, it would be very challenging if users were forced to make their list of a certain length. For example, if our function was designed as follows:

```
def shopping_list(item_1, item_2, item_3):
    print(f"""
Shopping List:
    * {item_1}
    * {item_2}
    * {item_3}
    """)
```

users would be required to have shopping lists that consist of exactly three items. Otherwise, the function will throw an error. A remedy to allow users more flexibility would be to make all of the arguments optional, setting their default values to *None*. Using this approach, users would be able to input as many items as they like, up to the amount of optional arguments that exist in the function. Additionally, the number of optional arguments could be increased to allow for longer lists. Although this approach solves the need for more flexibility, the function's body becomes unnecessarily long and inefficient:

```
def shopping_list(item_1=None, item_2=None, item_3=None):
    print("Shopping List:")

    item_list = []

    if item_1 != None:
        item_list.append(item_1)

    if item_2 != None:
        item_list.append(item_2)

    if item_3 != None:
        item_list.append(item_3)

    for item in item_list:
        print('\t*',item)
```

A more pragmatic approach is to use **args*, which will allow for flexibility while keeping the function short and simple. Such an approach creates an opportunity to redesign the body of the function so that it contains a single *for* loop, as exemplified in *Code 9.5.1*.

In [ ]:

```
## Code 9.5.1 ##

# replicated from Code 1.1.5

# defining a function with *args
def shopping_list(*args):
    print("Shopping List:")

    for item in args:
        print(item)

# testing the function
shopping_list('bananas', 'oranges',
              'grapes', 'pears',
              'apples')
```

The concept of ***kwargs*** is very similar to that of **args*, but instead of operating on individual argument inputs, *kwargs* operates on key/value pairs. Conceptually, keys can be thought of as groups or categories, and values the members of each group. In an Excel spreadsheet, keys are the names of each column and values the data in each row. In Python, key/value pairs are the essence of a special built-in object known as dictionaries (https://docs.python.org/3/tutorial/datastructures.html#dictionaries), which will be covered in a later chapter.

To exemplify, let's use ***kwargs* to create a function for daily exercise that consists of three parts: warm up, workout, and cool down (the *keys*). Each activity done in these parts can be considered a value. For example, if our warm up consisted of stretching, running, push ups, and sit ups, we could use this information to create the following key/value pair:

```
warm_up  = ['stretch', 'run', 'push ups', 'sit ups']
```

Indexing key/value pairs is a bit more complicated than indexing lists or tuples, and this will be discussed in a later chapter. For now, let's turn our attention to *Code 9.5.2* to better understand how ***kwargs* can be applied.

In [ ]:

```
## Code 9.5.2 ##

# defining a function with **kwargs
def daily_workout(**kwargs):

    for key, value in kwargs.items():
        print(f'\n{key.upper()}:')

        for v in value:
            print('\t__',v)


# testing the function
daily_workout(warm_up  = ['stretch',
                          'run for 10 minutes',
                          '10 push ups',
                          '10 sit ups'],

              workout  = ['3 sets shadow boxing',
                           '3 sets weight lifting circuit (arms & back)'
                           'cycle for 30 minutes'],

              cooldown = ['run for 10 minutes',
                           'stretch'])
```

Having an understanding of keyword arguments will pay dividends when working with open-source packages. For example, the method heatmap (https://seaborn.pydata.org/generated/seaborn.heatmap.html) from the data visualization package seaborn (https://seaborn.pydata.org/) contains a number of optional arguments, allowing for a high degree of customization. Even so, the final argument in its function definition is ***kwargs*, as can be seen below:

```
heatmap(data, vmin=None, vmax=None, cmap=None, center=None, robust=False, annot=No
ne, fmt='.2g', annot_kws=None, linewidths=0, linecolor='white', cbar=True, cbar_kw
s=None, cbar_ax=None, square=False, xticklabels='auto', yticklabels='auto', mask=N
one, ax=None, **kwargs)
```

Through an analysis of the *help( )* file for *seaborn.heatmap( )*, the ***kwargs* parameter is meant to house other keyword arguments that are passed to ax.pcolormesh (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.pcolormesh.html):

```
kwargs : other keyword arguments
        All other keyword arguments are passed to ``ax.pcolormesh``.
```

This means that you can use arguments from *ax.pcolormesh( )* to further customize the output of *seaborn.heatmap( )*. The reason this is possible is because *seaborn* was built on top of the package matplotlib (https://matplotlib.org/3.1.1/index.html). Instead of redundantly loading *seaborn.heatmap( )* with a plethora of arguments that already exist in *matplotlib.axes.Axes.pcolormesh( )*, the developers made these arguments available as ***kwargs*. Therefore, they can be accessed indirectly.

9.6 Variable Scope and Nested Functions

As with conditional statements, functions can be written within functions. Before discussing further, however, it is important to reemphasize that variables in a function's body are in a different environment than variables (i.e. objects) outside of a function's body. This is a concept known as **variable scope**. The default environment outside of a function's body is known as the **global environment**. This is where you will be if you open a blank Jupyter Notebook or a new Python script. All variables created in the **global environment** are globally scoped. This means that these variables are available in every part of your script (including the body of a function).

For example, if we defined the object *' y '* in the global environment, we could then reference it in the body of a function. This is occurring in *Code 9.6.1*, which is an adapted version of *Code 9.2.3*. As can be observed, *' y '* is being declared in the global environment and then is referenced within the body of a function. Even though this variable has not been explicitly declared within *power_function*, *Code 9.6.1* runs without error. Behind the scenes:

1. Python realizes that an object has been referenced in the body of a function.
2. It starts searching upwards, line by line, within the function's body to see if this object has been declared.
3. If it finds the object's declaration within the body, it uses this value.
4. If it doesn't find a declaration, continues its search in the global environment.

In [ ]:

```
## Code 9.6.1 ##

# adapted from Code 9.2.3

# declaring an object
y = 2

# writing a function using the declared obj
def power_function(x):
    val = x**y
    return val

power_function(x=2)
```

In *Code 9.6.2*, the object *' y '* has been declared in both the global environment and the function's environment (also known as the **local environment**). As *' y '* has been declared locally, Python uses the local declaration of *' y '* each time *power_function* is run. However, this does not effect the global declaration of this variable, which retains its originally declared value of two.

In [ ]:

```
## Code 9.6.2 ##

# adapted from Code 9.6.1

# declaring an object
y = 2

# writing a function using the declared obj
def power_function(x):
    y = 3
    val = x**y
    return val

print(power_function(x=2))
print(y)
```

Things get very interesting when working with nested functions, as in such situations there are more than two levels of scope:

```
GLOBAL

    def outer_function():
        LOCAL

        def inner_function:
            EVEN MORE LOCAL
```

This creates a challenge when trying to manage the scope of objects. To help sort out the confusion, let's give a unique name to each level:

Level of Scope	Interpretation
global	outer-most environment
non-local	environment of the **outer** function
local	environment of the **inner** function

As stated earlier in this chapter, by default, anything that takes place inside of a function has no effect on objects in the global environment. The same is true for functions nested inside of other functions: by default, the environment of the inner function has no effect on the environment of the outer function. It was also mentioned earlier that you can conceptualize the **global environment** as an entire planet and the **non-local environment** as its own secluded island. As such, the **local environment** can be thought of as a cave on the secluded island, where its inhabitants are isolated from the rest of the island's population. If a **return** statement is placed in the **non-local environment**, something returns to the **global environment**. Likewise, if a **return** statement is placed in the **local environment**, something returns to the **non-local environment**. This is exemplified in *Code 9.6.3*. For the convenience of the reader, each environment has been labeled with comments and outlined with hashtags (`'#'`).

In [ ]:

```python
## Code 9.6.3 ##

# global environment

##############################################################################
def outer_function():
    # non-local environment
    given_name  = 'Long'
    family_name = 'Silver'

    ##################################################
    def inner_function():
        # local environment
        middle_name = 'John'
        return middle_name
    ##################################################

    # non-local environment
    middle = inner_function()
    full_name = given_name + ' ' + middle + ' ' + family_name

    return full_name

##############################################################################

# global environment
outer_function()
```

This code starts by defining an outer function where two objects are declared: *given_name* and *family_name*. In the definition of the inner function, one new object is declared (*middle_initial*) and returned to the **non-local environment** (i.e. the environment of the outer function). Notice how after the inner function is defined, it is called in the outer function. Remember, in order to use a function, it must be called after it has been defined. The results of *inner_function* are being stored as a new object in *outer_function*, and this object is being used as an input for the object *full_name*. Finally, *full_name* is being returned to the **global environment**. When the function is called on the last line of *Code 9.6.3*, Python outputs what is stored in *full_name*, which is a concatenation of *given_name*, *middle_initial*, and *family_name*.

9.7 *global* and *nonlocal* Variable Assignment

As mentioned in *Section 9.2*, Python has been designed such that objects in inner environments have no effect on objects in outer environments **with one exception:** explicitly stating that an inner object should override variables in outer environments. This can be accomplished with the use of the syntaxes **global** and **nonlocal**.

Essentially, when these syntaxes are used, they override assigned objects in outer environments. As their names suggest, **global** will override objects in the global environment, and **nonlocal** will override objects in an outer function. If there are multiple levels of function nesting, **nonlocal** will act similar to *break* in nested loops: it will override objects one level up in the environment hierarchy. *Code 9.7.1* exemplifies how to declare a variable as **global**.

Note: Declaring variables as **global** or **nonlocal** should be done with extreme caution as it can cause undue side effects. For more information, check out this thread on StackOverflow (https://stackoverflow.com/questions/19158339/why-are-global-variables-evil).

In [6]:

```python
## Code 9.7.1 ##

# adapted from Code 9.6.3

# global environment

################################################################################
def character():
    # non-local environment

    global character_name
    character_name    = 'Long John Silver'

################################################################################

# global environment

# calling character() and printing name
character()
print(f'Character Name:  {character_name}')
```

```
Character Name:  Long John Silver
```

Notice in *Code 9.7.1* that *character_name* was not originally defined in the global environment, and the *character( )* function does not have a **return** statement. Even so, the function returns *character_name*. Behind the scenes, when *character_name* is declared **global**, its return statement is implied and it is automatically returned to the **global environment**. Also note that *character_name* must be declared **global** before it is assigned any value.

global can also be used to override an existing object in the **global environment**. This is exemplified in *Code 9.7.2*, where *character_name* is being overridden when the *character( )* function is being called.

In [ ]:

```python
## Code 9.7.2 ##

# adapted from Code 9.6.3

# global environment
character_name = 'Long Silver'

##############################################################################
def character():
    # non-local environment

    global character_name
    character_name    = 'Long John Silver'

##############################################################################

# global environment

# printing name before function is called
print(f'Character Name BEFORE function: {character_name}')

# calling character() and printing name again
character()
print(f'Character Name AFTER function:  {character_name}')
```

Declaring an object as **nonlocal** is very similar to declaring it as **global**. As with before, it is unnecessary to write a **return** statement at the end of the inner function. However, in order for a variable to be declared **nonlocal**, it must first be defined in the outer function's environment. Otherwise, the code will throw an error. Also note that declaring an object as **nonlocal** does not affect the **global environment**. *Code 9.7.2* provides an example as to how to apply a **nonlocal** declaration.

In [ ]:

```python
## Code 9.7.2 ##

# adapted from Code 9.6.3

# global environment

##############################################################################
def character():
    # non-local environment

    # setting defaults as Homer Simpson
    given_name  = 'Long John'
    family_name = 'Silver'
    character   = given_name + ' ' + family_name

    ##################################################
    def change_character():
        # local environment

        # using nonlocal to override local given_name
        nonlocal character
        character = 'Captain Flint'
    ##################################################

    # non-local environment
    print(f"""
You are currently playing as {given_name} {family_name}.\n
    """)

    # allowing for character change
    change = input("Would you like to play as Captain Flint instead? [y]/n\n")
    change = change.casefold()


    # conditionally changing character
    if 'y' in change:
        change_character()

    # passing if a user does not want to change characters
    elif 'y' not in change:
        pass

    else:
        print('Something went wrong.')

    # outputting final character
    return character

##############################################################################

# global environment
player = character()
print(f'\nWelcome to the game {player}!')
```

This code block also introduces the syntax **pass**, which Python's way of saying *'do nothing and continue'*. Generally, **pass** is used as a placeholder when building complex code. For example, when building a code with several nested functions, loops, or conditional statements, a programmer may decide to first build a skeleton (i.e. an outline). With the use of **pass**, the programmer can build various parts of the code and immediately test them. Without **pass**, Python will throw an error each time it sees a conditional statement with no body. The use of **pass** is illustrated in the code below.

```
buttons = 50

if buttons < 5:
    pass

elif buttons <= 25:
    pass

elif buttons > 25:
    print("That's a lot of buttons!")

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

9.8 Exception Handling

According to the Python documentation (https://docs.python.org/3/tutorial/errors.html), errors come in two forms: syntax errors and exceptions. Syntax errors occur when a code violates the grammar of Python (a missing semicolon, improper indentation, etc.), and generally result in the following error message:

```
SyntaxError: invalid syntax
```

Exceptions are events that disrupt the flow of a program's execution. Things such as attempting to divide my zero, calling an object that has not been defined, and trying to divide a string by another string lead to exceptions. When an exception occurs, Python throws an exception error (https://docs.python.org/3/library/exceptions.html#concrete-exceptions), which is a message designed to help programmers understand why their code did not run as intended. The table below exhibits some common exception messages you may experience while applying Python to business analytics.

Exception Message	Interpretation
ImportError	raised during *import* when Python is having trouble loading a module
ModuleNotFoundError	raised during *import* when a module cannot be found
IndexError	raised when an index value is out of range
KeyboardInterrupt	raised when you interrupt/restart your Python kernel

NameError	raised when an object is referenced that cannot be found
TypeError	raised when an operation fails due to an inappropriate object type
ValueError	raised when an object is of an appropriate type, but is of an inappropriate value
ZeroDivisionError	raised when an operation attempts to divide by zero
FileNotFoundError	raised when Python cannot find a file that has been referenced

In [ ]:

```
## Code 9.8.1 ##

# replicated from Code 3.2.5

number = input("""
What is your favorite number between 1 and 10?
Please input numbers (no text).\t""")

print(f"\nYou've input {number}.")

# Converting number to type int
number = int(number)

double = number * 2

print(f"""
If you double that number, it becomes {double}.
""")
```

As an example, let's turn our attention to *Code 9.8.1*, which is a replication of *Code 3.2.5*. As you may recall, if a user inputs anything other than an integer, this code will throw an exception. For example, if a user were to input a string or float, Python would throw the following:

```
    ValueError: invalid literal for int() with base 10: 'VALUE'
```

This is due to the type conversion in *Line 12*, as Python does not know what to do when encountering a non-integer value. However, **try**/**except** can be utilized to prevent this code from terminating in such a situation. In its base form, the **except** clause will run regardless which type of exception occurs. If applied to the body of a *while True* loop, **try**/**except** can be used with *break* and *continue* to keep looping until a user gives an appropriate input. This has been developed in *Code 9.8.2*.

In [ ]:

```python
## Code 9.8.2 ##

# adapted from Code 3.2.5

while True:
    number = input("""
What is your favorite number between 1 and 10?
Please input numbers (no text).\t""")

    print(f"\nYou've input {number}.")

    # Converting number to type int
    try:
        number = int(number)
        break

    except:
        print("That wasn't a proper input.")
        continue

double = number * 2

print(f"""
If you double that number, it becomes {double}.
""")
```

Customizing *try*/*except* based on Exception Types

Generally, it is a good practice to write an exception clause for each error that a user may reasonably encounter. Such a practice allows for different exceptions to lead to different actions. For example, the user-defined function in *Code 9.8.3* is likely to encounter either a *TypeError* (user did not overwrite the default value for the argument *apples*), or a *ValueError* (user did not input an integer value for *apples*). As such, two **except** clauses have been coded, each leading to a different action in order to remedy its respective error. Once the function receives an appropriate value for *apples*, the **try** clause runs successfully and breaks out of the *while True* loop.

In [ ]:

```python
## Code 9.8.3 ##

# declaring an object
apple_inventory = 0

# writing a function
def apple_orchard(apples=None, inventory=apple_inventory):
    """This function adds apples to your inventory."""

    # handling exceptions
    while True:
        try:
            apples = int(apples)
            break

        except TypeError:
            apples = input("How many apples have you picked?\n")

        except ValueError:
            print("That's not a valid number of apples. Please try again")
            apples = input('>')

    print(f"Adding {int(apples)} apples to your inventory!")
    inventory += apples
    return inventory

# running the function
apple_inventory = apple_orchard()
```

Let's apply **try**/**except** to *Code 8.4.3*, which has been replicated in *Code 9.8.4*. As you may recall, this code was designed to calculate how many seasons it took for Michael Jordan to score 25,000 points. Our original approach threw exceptions when encountering the strings in the points per season data (index 1 in each sublist). Thus, one of our first steps was to clean the data so that points per season only contained integers. This was a good approach, but it is limited. To illustrate, let's assume an overnight change was made to our data source and all data within had been converted to strings. In such a case, the data cleaning process conducted in **Chapter 8: while Loops and Making Assumptions** would result in all data being removed due to its type. A more sound approach would be to **try** to convert each string into an integer and tally successful attempts accordingly. This would also allow for the possibility of calculating how many seasons were skipped due to Jordan not playing. *Code 9.8.4* has been left open for you to:

- convert the values for points per season into an integer
- handle exceptions when a value is inappropriate for the aforementioned type conversion
- add new functionality such that the code outputs the number of seasons where Jordan did not play leading up to surpassing the point limit

In [ ]:

```
## Code 9.8.4 ##

# open coding block

# replicated from Code 8.3.2
jordan_stats = [["'84-'85", '2313', 'Y'],
                ["'85-'86", '408',  'N'],
                ["'86-'87", '3041', 'Y'],
                ["'87-'88", '2868', 'Y'],
                ["'88-'89", '2633', 'Y'],
                ["'89-'90", '2753', 'Y'],
                ["'90-'91", '2580', 'Y'],
                ["'91-'92", '2404', 'Y'],
                ["'92-'93", '2541', 'Y'],
                ["'93-'94", 'DNP', 'DNP'],
                ["'94-'95", '457',  'N'],
                ["'95-'96", '2491', 'Y'],
                ["'96-'97", '2431', 'Y'],
                ["'97-'98", '2357', 'Y'],
                ["'98-'99", 'Retired', 'Retired'],
                ["'99-'00", 'Retired', 'Retired'],
                ["'00-'01", 'Retired', 'Retired'],
                ["'01-'02", '1375', 'N'],
                ["'02-'03", '1640', 'N']]


# adapted from Code 8.4.3
"""
Assumptions:
    Calculations should only include seasons where Jordan played.
"""

# declaring objects
total_points  = 0
total_seasons = 0
point_limit   = 25000


# writing the outer loop
for season, points, lead_scorer in jordan_stats_2:

    # writing the inner loop
    while total_points < point_limit:
            total_points  += points
            total_seasons += 1
            break


# printing the results
print(f"""
{'*' * 40}

It took {total_seasons} seasons for Jordan to score
more than {point_limit} points (scoring {total_points}).

{'*' * 40}
""")
```

Show Solution

9.9 Summary

User-defined functions are a powerful tool that allow programmers to blueprint code so that it can be used over and over again. Defining a function is similar to writing a conditional statement or a loop in that the first line starts with a designated syntax and ends with a semicolon. Also, the body of a function is indented. Arguments can be specified when defining a function, and can be mandatory (no default value), optional (contains default value), or variable (varies in length). All functions should contain docstrings, regardless of how simple they may be. Docstrings should explain what a function is intended to do, list and explain parameters, and give usage examples.

A variable's scope can have unexpected consequences on other objects. Nested functions require one to pay particular attention to the scope of each variable, which can be managed with the use of **return** statements and scoping syntax such as **global** and **nonlocal**. Scoping syntax should be used with extreme caution as it may adversely affect a user's code. Also, function names should be unique as to not unintentionally overwrite an existing function.

Finally, comprehending how to write, document, test, and exception handle user-defined functions will greatly increase your ability to benefit from functions available in the open source community. When researching packages available in the open-source community, the quality of a package's documentation can indicate whether or not the package is stable and/or will work as intended. When encountering a package that is not documented well, it may be wise to search for an alternative tool.

```
   ,ggggggg,
 ,dP""""""Y8b                                ,dPYb, ,dPYb,
I8
 d8'    a  Y8                                IP'`Yb IP'`Yb
I8
 88     "Y8P'                                I8  8I I8  8I                      88
888888
 `8baaaa                                     I8  8' I8  8'
I8
,d8P""""         ,gg,   ,gg  ,gggg,    ,ggg,   I8 dP  I8 dP   ,ggg,    ,ggg,,ggg,
I8
d8"            d8""8b,dP"  dP"  "Yb i8" "8i  I8dP   I8dP   i8" "8i  ,8" "8P" "8,
I8
Y8,           dP   ,88"   i8'       I8, ,8I  I8P    I8P    I8, ,8I  I8   8I   8I
 ,I8,
`Yba,,_____,,dP  ,dP"Y8, ,d8,_    _ `YbadP' ,d8b,_ ,d8b,_  `YbadP' ,dP   8I   Yb,,
d88b,
  `"Y88888888"  dP"   "Y8P""Y8888PP888P"Y8888P'"Y888P'"Y88888P"Y8888P'   8I   `Y88
P""Y8
```