

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 I: The Absolute Essentials

Chapter 1: Before Learning Anything Else, Learn This

If you are new to coding, do not skip this chapter.

Python was designed to be a simple programming language. In fact, in 1999, Guido van Rossum, the founder of the Python language, sent a funding proposal to DARPA for his *Computer Programming for Everybody* initiative, where he posited the following question:

"What would the world look like if users could program their own computer?"

The proposal went on to make the following claim:

"There is enough (anecdotal) evidence that Python is easy to learn for people who are (nearly) computer-illiterate."

This proposal, which was submitted by van Rossum and the *Python in Education Special Interest Group* [can be found here](https://www.python.org/doc/essays/everybody/) (<https://www.python.org/doc/essays/everybody/>). It is an interesting read and makes some very good points. However, even though Python may be easy to learn, **it is important to learn how to learn Python, and that is the purpose of this chapter.** At times, you will get stuck, and you need to know what to do. There will also be many times when your code won't run properly, even when you've spent hours trying to

debug it. This is very frustrating, and it is something all coders experience from time to time. The good news is that as long as you are not one of the world's most advanced Python coders, someone has already experienced and solved your problem. More importantly, it is very likely that a solution to your problem has been generously shared to the open source community. All you need to know is where to find it. Not knowing can be devastating to your morale. To exemplify this, the following story is from one of my former students, which took place while conducting an analysis for the course: Machine Learning.

Lucas Barros - Masters of Business Analytics - Class of 2019

“While studying in the Masters of Business Analytics program, I was also working part time at the café on campus. It's always good to make some extra money as a graduate student. Learning analytics, especially coding, was a very interesting experience, although sometimes I would lose confidence, questioning whether this was the right field for me. A great example of this is the Game of Thrones Character Prediction analysis project.

We had a dataset based on the book series, which contained around 2000 characters and several features describing each character. It was probably the most stressful project of my life. Throughout this project, sleeping 6 hours a night was a luxury. I spent countless hours working on the dataset: engineering new features, testing out different machine learning algorithms, and trying not to question my life choices too deeply. On the note of life choices, coincidentally, the campus café was looking to hire a new manager and they asked me if I would like the position. I have to be very sincere, after the lack of sleep, the stress of hours of trying to debug my code to no avail, and the process of trying to build an algorithm that predicts reasonably well, the option of giving up and living a more chill life sounded very appealing.

Nonetheless, after some more nights with minimal time to sleep, I found solutions to my coding issues and am proud to say I completed the project. While being one of the most difficult projects I've ever encountered, it was by far one of the most rewarding!”

In this chapter, we will cover five critical resources in an effort to alleviate your long-term coding frustration:

- finding help with the `help()` wrapper
- code complete in Jupyter notebook
- finding answers on the Internet
- talking to humans through code comments

Reminder: If you are new to coding, do not skip this chapter. Time invested here will save you several hours as you move forward.

1.1 Finding Help with the `help()` wrapper

Introduction

One of the most critical functions for programmers at all levels is the **`help()`** function. This is one of the most amazing functions ever written, and you will be using it quite often. To learn more about what it does, search *help* in the **`help()`** function (*Code 1.1.1*).

In []:

```
## Code 1.1.1 ##  
  
help(help)
```

We call the output of *Code 1.1.1* the help function's **documentation**. According to the documentation for **help()**, there are two ways to use this function:

- ~~Calling `help()` at the Python prompt starts an interactive help session.~~
- Calling `help(thing)` prints help for the Python object 'thing'.

Notice how the first bullet point above has been stricken out. That is because we are going to **avoid using interactive help for the time being**. Believe it or not, interactive help sessions can (ironically) cause problems that most beginners are not ready to solve. If you are not comfortable using a command-line interface (i.e. terminal or PowerShell), avoid using interactive help. If you don't know what a command-line interface is, don't worry. For now, we are going to focus on mastering the *help(thing)* option. In *Code 1.1.2*, we are using this to check the documentation for the *print()* function.

Side Note: The Challenge with Interactive Help

After starting an interactive help session, you may run into a situation where you are unable to run any code. This is because your Python kernel is still running interactive help, and it cannot move on until you give it a command to do so. This can be done by typing **quit** into the help search box, but if we accidentally tried to run some code without closing interactive help, our Python kernel might get confused and need to be restarted.

Using help()

In []:

```
▼ ## Code 1.1.2 ##  
  
# Starting an interactive help session  
help(print)
```

Code 1.1.2 generates a very manageable amount of output. Let's dissect each component piece by piece. First, let's add line numbers for easier interpretation.

```
1 | Help on built-in function print in module builtins:  
2 |  
3 | print(...)  
4 |  
5 |     print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)  
6 |  
7 |     Prints the values to a stream, or to sys.stdout by default.  
8 |     Optional keyword arguments:  
9 |     file: a file-like object (stream); defaults to the current sys.stdout.  
10 |     sep: string inserted between values, default a space.  
11 |     end: string appended after the last value, default a newline.  
12 |     flush: whether to forcibly flush the stream.
```

Line	Interpretation
1	States that this is a built-in function .
3	The function being looked up.
5	The arguments for the function.
7	Explains what the function does.
9-12	Explains what each argument does.

Line 1 - Built-in functions

Python comes with a set of default, or **built-in functions**. These are also referred to as **primitives**, and are functions that are so important, they are available without the need to import additional packages (for an example of a package import, see *Code 1.1.3* where we import the package *pandas*). At the time of this writing, the most recent version of Python (*version 3.7.3*) has 69 built-in functions, which can be found in [the Python documentation on Built-In Functions \(https://docs.python.org/3/library/functions.html\)](https://docs.python.org/3/library/functions.html). Python also comes with a series of other built-ins. For example, *built-in types*. According [to the Python documentation on Built-In Types \(https://docs.python.org/3/library/stdtypes.html\)](https://docs.python.org/3/library/stdtypes.html), principal built-in types include numerics, sequences, mappings, classes, instances and exceptions. Think of built-ins as the primary numbers of Python: every function can be broken down into these built-ins. Also, keep in mind that Python was designed to be used for a wide variety of programming tasks, and what is commonly used for business analytics is just one small subset. To avoid information overload, this book will primarily focus on the built-ins, functions, methods, and packages that are most relevant to our purposes. If you're not sure what a function, method, or package is, don't worry. We will go into these details in later chapters.

Line 3 - The function being looked up

As the heading implies, this line specifies what you have looked up. When we called help on the *print()* function in *Code 1.1.2*, this didn't seem to add a lot of value. However, this information becomes extremely useful in other situations, such as in *Code 1.1.3* where *help()* is being called on a user-created object. For now, don't focus on what *Code 1.1.2* is trying to do. Instead, focus on the *help()* function in the last line.

In *Code 1.1.3*, we created an object and called **help()** on it (creating objects will be covered in more detail in **Chapter 2 - Printing, Dynamic Strings, and Escape Sequences**. From the object's name (*my_list_converted*), it is unclear as to what this object actually is. Notice that the help function recognized that this object is a DataFrame and returned its respective documentation. This is the

result when we call help on any named object. Below is a snippet of Line 3 for *Code 1.1.3*

```
class DataFrame(pandas.core.generic.  
NDFrame)
```

In []:

```
▼ ## Code 1.1.3 ##  
  
import pandas as pd  
  
my_list = [[1, 2, 3] , [4, 5, 6]]  
  
my_list_converted = pd.DataFrame(my_list)  
  
help(my_list_converted)
```

Line 5 - Function Arguments

For most functions you will encounter, arguments will come in three forms: *mandatory*, *optional*, and *variable* (also known as **args* and ***kwargs*). Line 5 of *Code 1.1.2* contains mandatory and optional arguments.

Mandatory Arguments

Mandatory arguments are those that do not have a default value. All mandatory arguments must be specified for a function to run. If at least one is missing, the function will throw an error. The easiest way to tell if an argument is mandatory is by checking whether or not it already has a value assigned to it, which is indicated by an equals sign. If no equals sign, then the argument is mandatory. In *Code 1.1.2*, the argument *value* is mandatory.

Optional Arguments

Optional arguments have a default value, which is indicated after an equals sign. If you do not specify anything for these arguments, the default value will be used and the function will run properly. In *Code 1.1.2*, the arguments *sep*, *end*, *file*, *flush* are optional arguments.

Variable Arguments

Variable arguments are slightly more advanced. As their name implies, they allow a function to accept a *variable* number of arguments. This may seem confusing, but the idea behind such an invention is quite remarkable. Let's say, for example, that a programmer wanted to create a function to help organize their grocery list. The programmer may do some research and come to the conclusion that most shopping trips consist of exactly three items and write a code similar to the one displayed in *Code 1.1.4*.

A challenge arises in that the function in *Code 1.1.4* requires exactly three items to work properly as all arguments are mandatory. In other words, each shopping list needs to be exactly three items long. Lessons from Social Life 101 have taught us that the number of items on our shopping list will vary. Therefore, the programmer needs a way to allow for this functionality. This is where variable arguments become very handy.

By changing the arguments to *args* as in Code 1.1.5, our shopping list can be of any length. ***kwargs* is similar to *args* in that it allows for arguments of varying length. However, it operates using keywords, which is a concept for a later chapter. ***kwargs* will become more important when we discuss dictionaries in a later chapter.

Note: Try not to get caught up in the syntax of Code 1.1.5. It will be explained in later chapters.

In []:

```
▼ ## Code 1.1.4 ##
▼ def shopping_list(item_1, item_2, item_3):
    print("Shopping List:")
    print(item_1)
    print(item_2)
    print(item_3)

shopping_list('bananas', 'oranges', 'grapes')
```

In []:

```
▼ ## Code 1.1.5 ##
▼ def shopping_list(*args):
    print("Shopping List:")
    for item in args:
        print(item)
▼ shopping_list('bananas', 'oranges',
               'grapes', 'pears', 'apples')
```

Line 7 - What the Function Does

Line 7 of Code 1.1.2 states that the print function:

Prints the values to a stream, or to sys.stdout by default.

The programmers designing this function decided that this was the best way to explain what the *print()* function does. To someone less technical, this explanation may do more harm than good. An important concept to keep in mind is that:

**Programmers like to write in ways that other programmers can understand.
If you are not a programmer, you are not their target audience.**

This is a disadvantage for those of us that did not study software engineering or a similar subject. Luckily, there are ways to mitigate this disadvantage, which are discussed in Section 1.2. If you consider yourself to be less technical than a software engineer, please remember that as you advance in Python, your understanding of technical concepts will grow.

Lines 9-12 - What each argument does.

This section is very important as it will save you several hours of time when learning how to code. By reading and understanding the arguments, you will be able to do many things with only a handful of functions. Understanding a few functions at a detailed level is far more efficient than trying to memorize the basics of

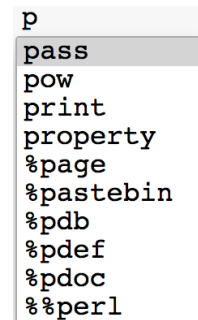
several functions. You will find that you can do more with less, and will have built a strong foundation and set good coding habits.

1.2 Code Complete in Jupyter Notebook

Jupyter Notebook, like many other interfaces, offers a code complete feature. This can come in very handy, and in this section we will discuss using code complete to enhance our abilities to use `help()`. This is a great way to explore a new package and develop an understanding as to its methods.

To activate code complete, simply start typing in a coding block and press `<tab>` on the keyboard. If this doesn't work, check your keyboard shortcuts in Jupyter's menu under *Help > Keyboard Shortcuts*. For example, if we open up a coding cell and type the letter `p`, followed by pressing `<tab>`, we get the result in *Figure 1.1*. In this case, Jupyter's code complete tool recognized the letter `p` and returned every currently-available syntax that starts with this letter.

Much of the syntax displayed in *Figure 1.1* are beyond our current scope, but with this feature, we can dive into new packages and syntax with ease, as exemplified in *Figures 1.2a through 1.2d*.



```
p
pass
pow
print
property
%page
%pastebin
%pdb
%pdef
%pdoc
%perl
```

Figure 1.1: Code complete results.

The steps throughout *Figures 1.2a through 1.2d* are using a technique called **chaining**, or linking multiple Python syntax together using a dot (i.e. a `"."`), to explore part of the *pandas* package. According to its [documentation from pydata.org \(https://pandas.pydata.org/\)](https://pandas.pydata.org/) *pandas* is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. It is a very powerful package for business analytics, and it is a very large. Anaconda has kindly included it in their Python 3 installation, and all we need to do is **import** it.

Importing Packages in Python 3

Importing packages is easy. To import *pandas*, we simply write:

```
import pandas
```

Conventionally, *pandas* should be imported as *pd*, so we modify the above code as follows:

```
import pandas as pd
```

We can also import specific modules and functions from a package with a minor adjustment to our syntax. For example, if we just wanted to import **DataFrame** (i.e. Python's version of a spreadsheet) instead of the entire *pandas* package, we could do so as follows:

```
from pandas import DataFrame
```

To illustrate the value of the line of code above: Have you ever noticed that when you get close to filling up your computer's hard drive it becomes very slow? The same concept applies to our Python environment. Every time something is imported, our Python environment slows down. This is why taking a minimalist approach by only importing what we need is a good practice. Since our current task involves exploring the *pandas* package, we will import the entire thing.

After *pandas* has been imported (as *pd*), it becomes available in our Python environment. If we run *help(pd)*, Python will return some documentation on *pandas*. If we add a dot to our syntax and press *tab* on the keyboard as in *Figure 1.2(a)*, code complete displays all of the wonderful tools *pandas* has to offer. Some of these tools have deeper levels, including *np*, which stands for *numpy*. This is another key package for business analytics, and it is included in *pandas* (*pandas* is built on top of *numpy*).

If we chain *np.* onto our code and press *tab*, code complete will display all of the wonderful tools available in this package, as in *Figure 1.2(b)*.

This drill-down process can continue until we are at the deepest level of a package. *Figure 1.2(c)* drills down one level deeper by chaining *absolute.* onto our code, and *Figure 1.2(d)* drills down one level deeper by chaining *accumulate.* Since there are no further levels to drill down into, code complete does not display a popup window when trying to extend the chain beyond *pd.np.absolute.accumulate.*

Much of the syntax discovered in each step of this chaining example is beyond our current scope, and seeing it for the first time can be overwhelming. Keep in mind: Python was designed for a wide array of programming tasks. Having so many things pre-built and maintained is going to save you an immense amount of time without costing you a penny. This is one of the key benefits of Python as well as other open source programming languages.

Below is a summary of the chaining steps:

1. *pandas* was imported as *pd*

```
import pandas as pd
help(pd.)
NaT
notna
notnull
np
np_array_datetime64_compat
np_datetime64_compat
offsets
option_context
options
pandas
```

Figure 1.2(a): Exploring the pandas package (1 of 4).

```
import pandas as pd
help(pd.np.)
abs
absolute
absolute_import
add
add_docstring
add_newdoc
add_newdoc_ufunc
alen
all
allclose
```

Figure 1.2(b): Exploring the pandas package (2 of 4).

2. a `help()` wrapper was created
3. Code complete was called on `pd.` to access its available syntax
4. `np` was selected, chaining together `pd.np`
5. Code complete was called on `pd.np.` to access its available syntax
6. `absolute` was selected, chaining together `pd.np.absolute.`
7. Code complete was called on `pd.np.absolute.` to access its available syntax
8. `absolute` was selected, chaining together `pd.np.absolute.accumulate`

```
import pandas as pd
help(pd.np.absolute.)
```

autocomplete suggestions:

- accumulate
- at
- identity
- nargs
- nin
- nout
- ntypes
- outer
- reduce
- reduceat

Figure 1.2(c): Exploring the pandas package (3 of 4).

```
import pandas as pd
help(pd.np.absolute.accumulate)
```

Figure 1.2(d): Exploring the pandas package (4 of 4).

Figure 1.3 displays each of these steps side-by-side. Its final code can be run in Code 1.2.1.

```
import pandas as pd
help(pd.)
```

autocomplete suggestions:

- NaT
- notna
- notnull
- np
- np.array_datetime64_compat
- np.datetime64_compat
- offsets
- option_context
- options
- pandas

```
import pandas as pd
help(pd.np.)
```

autocomplete suggestions:

- abs
- absolute
- absolute_import
- add
- add_docstring
- add_newdoc
- add_newdoc_ufunc
- alen
- all
- allclose

```
import pandas as pd
help(pd.np.absolute.)
```

autocomplete suggestions:

- accumulate
- at
- identity
- nargs
- nin
- nout
- ntypes
- outer
- reduce
- reduceat

```
import pandas as pd
help(pd.np.absolute.accumulate)
```

Figure 1.3: Code complete results after method chaining.

In []:

▼ ## Code 1.2.1 ##

```
import pandas as pd
help(pd.np.absolute.accumulate)
```

In []:

▼ ## Space to practice using help() ##

1.3 Finding Answers on the Internet

Note: This is by no means an exhaustive list, and there are many great resources that are not mentioned here. If you find a different resource that explains code in a way that fits well with your learning style, use it. These two resources are mentioned as I have found them incredibly helpful in my coding journey.

Stack Overflow

There are many great coding resources available on the Internet. At the time of this writing, [Stack Overflow](https://stackoverflow.com) ([www.stackoverflow.com](https://stackoverflow.com)) is one of the most popular. According to its website:

Stack Overflow is a question and answer site for professional and enthusiast programmers.

In other words, Stack Overflow is a place where coders ask questions to coders. It is a wonderful place to find information to help get you through your coding challenges, and is also a place where you can post your own questions so that the coding community can help you out. This is one of the key benefits of the platform, and it is free of charge.

Caution: The relevance of your search results can be hit or miss. Sometimes you will have trouble articulating your question in a way that gets good results. Other times you may find solutions to your question, but they are too complicated for your current coding ability. Every now and then, you will be more confused than when you started your search. This is normal. Keep in mind that Stack Overflow was designed so that coders could ask questions to coders, regardless of their programming level. You will find some very advanced solutions to problems that you never even knew existed. This leads us to Google, our second suggested resource.

Working with a Search Engine

It goes without saying that many of your coding questions can be resolved via a search engine. However, it is a good idea to reflect on why search engines are a good resource. Most notably, they allow us to ask questions the way that a human would ask them. We can use this to our advantage, thus alleviating any disadvantages we may have from not being the target audience of a programmer.

Ask questions with a search engine the way that you would ask someone at your programming level.

Interestingly enough, in many cases after experiencing poor results when searching on Stack Overflow, you will find that the top search result in a search engine is a page on Stack Overflow. Think about this. Search engines were designed in a way that is friendly to humans. The search functions in Stack Overflow was designed in a way that it is friendly to programmers, which is a subset of humans with a sophisticated understanding of programming jargon. When this happens, click on the Stack Overflow page and take note of how the question was phrased. This is a good way to improve your ability to find the information you need.

Challenge for the Reader: If you are serious about learning Python, register with Stack Overflow and start writing answers to other people's questions. This will intensify your learning speed. Stack Overflow also assists new contributors in that they let other coders know that they are new to the platform. This means that other coders will realize you are trying your best, and in most cases they will be more than willing to give you feedback.

1.4 Talking to Humans through Comments

Even though you may have very little coding experience, try reading the following code and understanding what the programmer was trying to accomplish:

In []:

```
▼ ## Code 1.3.1 ##  
  
import pandas as pd  
  
▼ original_df = pd.DataFrame([[None, 2, 3],  
                             [4, None, 6],  
                             [7, 8, None]])  
  
df_mean = pd.DataFrame.copy(original_df)  
  
▼ for col in df_mean:  
▼     if df_mean[col].isnull().any():  
        col_mean = df_mean[col].mean()  
        df_mean[col] = df_mean[col].fillna(col_mean).round(2)
```

We will be using a slightly-modified version of this code later in this book. If you're new to coding, it can be very intimidating. We learned in *Section 1.1* that we can use the `help()` function to understand what each line of code means. This is a good idea, especially if we are planning to use this code for a similar analysis. From reading the code, however, it is difficult to intuitively understand what this code is trying to do. Wouldn't it be great if the person who wrote this code gave us some additional documentation, in human language, that outlined what was happening step-by-step?

In the coding world, we provide this additional documentation via **comments**. Comments allow humans to talk to other humans within their code. They are very special in that when we write them, the computer knows we are talking to other humans and ignores our writing. This gives us the freedom to write anything we need to in order to clearly explain our code to others. In this section, we will cover two of the most widely-used forms of comments:

- hashtag comments (#)
- triple-quote comments (""" """)

Hashtag Comments (#)

As the name implies, hashtag comments are denoted by, well, a hashtag (#). The essential purpose of hashtag comments is that they help others understand what you are doing, even if you are doing it wrong. If there is a discrepancy between your hashtag comments and your code, more experienced coders are likely to pick that up and give you feedback. If we add hashtag comments to our code, it



In []:

```
▼ ## Code 1.4.2 ##

# Importing packages
import pandas as pd # data science essentials

# Creating a DataFrame that has missing values
▼ original_df = pd.DataFrame([[None, 2, 3],
                             [4, None, 6],
                             [7, 8, None]])

# Creating a copy of the original dataset so that I don't destroy the
# original.
# I'm planning to impute missing values on this dataset using the mean.
# This is why I called this df_mean instead of something else.
df_mean = pd.DataFrame.copy(original_df)

# This is a loop that looks for missing values in a column.
# For each column with missing values, use the mean to fill in the missing
# values.
▼ for col in df_mean:
▼     if df_mean[col].isnull().any():
        col_mean = df_mean[col].mean()
        df_mean[col] = df_mean[col].fillna(col_mean).round(2)
```

Although the functionality of *Code 1.4.2* is identical to *Code 1.4.1*, it is much easier to understand what each line is doing and the rationale behind why it was coded. Again, the computer does not care that these lines are here. It will see each '#' and know to skip anything on that line written after it.

Stand-Alone Strings

Now that we know Python ignores hashtags, it is not surprising that Python ignores other things as well. More accurately, there are some things that Python reads but does not know what to do with, and thus it does not affect our code. One such example is **stand-alone strings**. Take for example *Code 1.4.3*, which is a modified version of *Code 1.4.2*:

In []:

```
▼ # Code 1.4.3 ##

# Importing packages
import pandas as pd # data science essentials

# Creating a DataFrame that has missing values
▼ original_df = pd.DataFrame([[None, 2, 3],
                             [4, None, 6],
                             [7, 8, None]])

▼ ("""Creating a copy of the original dataset so that I don't destroy the
    original. I'm planning to impute missing values on this dataset using the
    mean. This is why I called this df_mean instead of something else.""")
df_mean = pd.DataFrame.copy(original_df)

▼ ("""This is a loop that looks for missing values in a column.
    For each column with missing values, use the mean to fill in the missing
    values.""")
▼ for col in df_mean:
▼     if df_mean[col].isnull().any():
        col_mean = df_mean[col].mean()
        df_mean[col] = df_mean[col].fillna(col_mean).round(2)
```

1.4 Talking to Humans through Comments

Even though you may have very little coding experience, try reading the following code and understanding what the programmer was trying to accomplish:

Notice how in *Code 1.4.3* we replaced the longer hashtag comments with triple-quote strings. This is a way to provide comments to humans on multiple lines of code in a more readable way. Technically speaking, Python will recognize these as lines of code and then try to run them. However, since there is no *print()* wrapper around these statements and they are not assigned to objects, Python has no use for them and immediately forgets that they exist. In other words, since Python has not been instructed to do anything with the triple quote statements, it ignores them. Finally, notice also that we added parentheses around each of the triple quotes. This is a good practice as it helps to organize our code and to prevent unwanted errors.

Note: There may be a time when someone reviewing your code tells you that using stand-alone strings (i.e. strings that aren't part of an object or a *print()* statement) to make comments is a bad practice. There is some rationale to this, as since Python runs stand-alone strings as code, it takes a tiny bit longer for your code to process. In fields like algorithmic trading, "a tiny bit longer" can mean missed opportunities, and thus such comments should be avoided. In business analytics, however, the purpose of most analyses is to gain insights and make recommendations. This takes a thorough and thoughtful exploration of the data, as well as constant communication with stakeholders to truly understand the problem you are trying to solve. In this sense, **your goal is to communicate as clearly as possible**. If this is best done by using stand-alone strings, use them.

Code 1.4.4(a) and *Code 1.4.4(b)* demonstrate the additional processing time from the use of stand-alone strings. As can be observed, the use of stand-alone strings had a very minor, if any, impact on the processing time of our code.

In []:

```
▼ ## Code 1.4.4(a) ##

import time
start_time = time.time()

# importing packages
import pandas as pd # data science essentials

# creating a DataFrame that has missing values
original_df = pd.DataFrame([[None, 2, 3],
                             [4, None, 6],
                             [7, 8, None]])

df_mean = pd.DataFrame.copy(original_df)

for col in df_mean:
    if df_mean[col].isnull().any():
        col_mean = df_mean[col].mean()
        df_mean[col] = df_mean[col].fillna(col_mean)

print("My program took", time.time() - start_time)
```

In []:

```
▼ ## Code 1.4.4(b) ##

import time
start_time = time.time()

# importing packages
import pandas as pd # data science essentials

# creating a DataFrame that has missing values
original_df = pd.DataFrame([[None, 2, 3],
                             [4, None, 6],
                             [7, 8, None]])

df_mean = pd.DataFrame.copy(original_df)

# Creating a copy of the original dataset
# original. I'm planning to impute missing values with the
# mean. This is why I called this df_mean
df_mean = pd.DataFrame.copy(original_df)

# This is a loop that looks for missing values. For each column with missing values, use the mean to fill them.
for col in df_mean:
    if df_mean[col].isnull().any():
        col_mean = df_mean[col].mean()
        df_mean[col] = df_mean[col].fillna(col_mean)

print("My program took", time.time() - start_time)
```

Side Note: Code Processing Time

The code used to time the difference between the two codes was found via [this page on Stack Overflow] (<https://stackoverflow.com/questions/12444004/how-long-does-my-python-application-take-to-run>). The template provided via the Stack Overflow page is shown in Code 1.3.5.

In []:

```
▼ ## Code 1.4.5 ##

import time
start_time = time.time()

print("My program took", time.time() - start_time, "to run")
```

1.5 Summary

`/\ / \ | \ | / \ | \) / \ | \ | | | / \ | \ | / \ | / \ , \ / | \ | \ > | \ / ~ \ | \ / | \ / ~ \ | | \ / | \ | \ . \ .`

Hats off to you for finishing the first chapter of this book. You are now ready to move forward given your understanding of:

- finding help with the `help()` wrapper
- code complete in Jupyter notebook
- finding answers on the Internet
- talking to humans through code comments

Chapter 2: Printing, Dynamic Strings, and Escape Sequences

Without a doubt, **`print()`** are one of the most fundamental functions in virtually any programming language. Although it is fundamental, it is something that you absolutely need to master. This chapter is dedicated to this function, as well as other very useful tools that will prove invaluable as you move forward. More specifically, this chapter covers:

- the **`print()`** function
- objects in Python
- working with a function's optional arguments
- enabling and escaping the meanings of special characters
- developing **`print()`** statements that are dynamic

2.1 The Fundamentals of `print()` Statements

Print statements are one of the most useful and fundamental operations in Python. We rarely want to print the output of every line of code that we write, and will even encounter several programs that do not print anything. The **`print()`** command helps us to manage what output gets printed, and what output should be suppressed.

What is printing?

To clarify, to *print* simply means to *display*. When we call the **print()** function, we are telling Python to display something. When we call **print()** in Jupyter Notebook, by default, Python displays its results directly under the cell containing the **print()** statement.

Basic Functionality

There is a lot we can do with **print()** statements, and this section is dedicated to mastering the fundamentals. Below is a table of the syntax covered in this section:

Table 2.1: Common **print()** statement syntax.

Syntax	Description
<code>print()</code>	outputs an argument to the console
<code>' '</code>	wrapper to print strings
<code>" "</code>	wrapper to print strings
<code>""" """</code>	wrapper to print strings on multiple lines
<code>f" {var} "</code>	dynamically calls variables/objects into print strings
<code>f""" {var} """</code>	dynamically calls variables/objects into print strings on multiple lines

As you can see from *Table 2.1* above, multiple wrappers are available to construct a print statement. A **wrapper** is, well, syntax that wraps around other syntax. For example, in the statement:

```
print('Hello world!')
```

two wrappers are present (the parentheses and the quotation marks). In this section, when we say wrapper, we are referring to the style of quotation marks that is wrapped around what we intend to print.

In *Codes 2.1.1(a)* through *2.1.1(c)*, three such methods are presented. Each of these codes results in the same output. This may have you wondering why Python 3 allows for several wrappers to do the same thing. You may also be wondering why we are learning three wrappers if each one results in the same output. There are two reasons for this:

- 1. *Hello world!* is a special case where all three wrapper techniques result in the same output. As we move forward, you will experience the advantages of having multiple ways to do (almost) the same thing.
- 2. You need to be capable enough to read other people's code, and they may use any combination of these wrappers.

In []:

```
## Code 2.1.1(a) ##  
  
print('Hello world!')
```

In []:

```
## Code 2.1.1(b) ##  
  
print("Hello world!")
```


In []:

```
▼ ## Code 2.1.1(c) ##  
  
print("""Hello world!""")
```

Effective Use of Print Wrappers

Suppose we wanted to print the following two statements:

"I told my wife that she's drawing her eyebrows too high," said the husband.

"She looked surprised."

Code 2.1.2 utilizes single, double, and triple quote wrappers in an attempt to print the first of these statements. Notice the differences in syntax coloring between each of these techniques. Using the single-quote wrapper, the string to be printed is cut off at the apostrophe in *she's*, and then picks up again at the end of the quote. Technically, this **print()** statement contains two strings: 1) "*I told my wife that she* (wrapped in single-quotes), and 2) *said the husband.*" (starting with a double-quote but never ending). The remaining text is not recognized as a string. Although this is clearly not what we had intended, it is important to note that this print statement will fail for three reasons:

1. The Python interpreter will reach the end of the first string and look for either a comma or a closing of the print statement. Since instead it will find the letter *s*, it does not know what to do and throws a syntax error.
2. The letter *s* (as well as the other words in-between the strings) are currently not defined as objects in our environment.
3. The second string is open. It needs to be closed with another double quote.

These "bugs" have been fixed in Code 2.1.3. Still there are much easier (and more effective) ways to solve such issues. Such remedies are offered in Code 2.1.4.

In []:

```
▼ ## Code 2.1.2 ##  
  
# Single-quote wrapper  
print('I told my wife that she's drawing her eyebrows too high," said the husband.')
```

(Note: A vertical red dashed line is present in the original image, indicating a syntax error at the closing single quote.)

```
# Double-quote wrapper  
print("""I told my wife that she's drawing her eyebrows too high," said the husband.""")
```

(Note: A vertical red dashed line is present in the original image, indicating a syntax error at the closing double quote.)

```
# Triple-quote wrapper  
print("""""I told my wife that she's drawing her eyebrows too high," said the husband.""")
```

In []:

```
▼ ## Code 2.1.3 ##

# Defining objects (covered in Section 2.2)
s      = 's'
drawing = 'drawing'
her     = 'her'
eyebrows = 'eyebrows'
too     = 'too'
high    = 'high'

# (Mostly) Debugged single-quote wrapper
print("I told my wife that she", s, drawing, her, eyebrows, too, high, "said the husband")
```

In []:

```
▼ ## Code 2.1.3 ##

# Defining objects (covered in Section 2.2)
s      = 's'
drawing = 'drawing'
her     = 'her'
eyebrows = 'eyebrows'
too     = 'too'
high    = 'high'

# (Mostly) Debugged single-quote wrapper
print("I told my wife that she", s, drawing, her, eyebrows, too, high, "said the husband")
```

In []:

```
▼ ## Code 2.1.4 ##

# Single-quote wrapper with escape sequence(s) (covered in Section 2.4)
print("I told my wife that she\'s drawing her eyebrows too high," said the husband.)

# Double-quote wrapper with escape sequence(s) (covered in Section 2.4)
print("\\"I told my wife that she's drawing her eyebrows too high,\" said the husband.")

# Combination of single and double quote wrappers
print("I told my wife that", "she's drawing her eyebrows too", 'high,' said the husband.)
```

Success! The first line of our two statements is debugged. Now it's time to print our second statement. We could opt to write two print statements, or we could utilize the primary advantage of the triple quote wrapper, which allows us to print on multiple lines. This wrapper wrapper also simplifies debugging, as by using it our

strings won't break if apostrophes or quotation marks are present. Remember, we are trying to print the following:

```
"I told my wife that she's drawing her eyebrows too high," said the husband.
```

```
"She looked surprised."
```

Triple quote wrappers also offer the opportunity to format print layouts in a similar manner to how we would do so in Microsoft Word or a similar software. I have found that a good practice when using triple-quotes is to give structure the wrapper such that it sits on its own lines of code. This is exemplified in *Code 2.1.5*.

In []:

```
▼ ## Code 1.2.5 ##  
  
print(  
    """  
    "I told my wife that she's drawing her eyebrows too high," said the husband.  
  
    "She looked surprised."  
    """)
```

The structure of *Code 1.2.5* allows for creative customization of our print statement. Below is an open coding block for you to experiment with different spacing.

In []:

```
▼ ## Code 1.2.6 ##  
  
print(  
    """  
    "I told my wife that she's drawing her eyebrows too high," said the husband.  
    "She...  
        looked...  
        surprised."  
    """)
```

Printing v. Sample Outputs

Notice that in *Code 2.1.1(d)*, we simply ran 'Hello world!' without a **print()** wrapper. At first glance, it appears that we have attained the same result as in *Codes 2.1.1(a) through 2.1.1(c)*. However, notice what happens when we add additional syntax to our code block, as in *Code 2.1.2*. Ah ha! Now things are different! Instead of outputting *Hello world!*, Python decided to output the result of `*1 + 1*`. In other words,

In []:

```
▼ ## Code 2.1.1(d) ##  
  
'Hello world!'
```

Python outputted the final line of code that it processed.

Outputting is not the same as printing! If we created a program consisting just the two lines of code in *Code 2.1.2*, the program would run behind the scenes and not generate any output. Jupyter Notebook, in a way, is doing us a favor by showing us some sort of output. However, we need to be mindful that we do not confuse this with something being printed.

To make things more interesting, let's see what happens when we add a print wrapper around *'Hello world!'* and keep the other lines of code the same (*Code 2.1.3*). In this code block, *Hello world!* printed (i.e. appeared on our visual interface) and the result of the calculation $1 + 1$ also appeared. As you may have imagined, if we add any lines of code under $1 + 1$, this output will no longer show up (*Code 2.1.4*). This is exactly why we have **print()** statements!

Code 2.1.5, contains three separate print statements. Wouldn't it be more efficient to combine these into one print statement? This is possible, as the **print()** statement was designed to handle multiple arguments. For example, in *Code 2.1.6*, we are saving different parts of our *Hello world!* statement as objects. When we wrap a **print()** statement around these objects, separating them with a comma, Python returns all of our arguments compiled into one result.

In []:

```
## Code 2.1.2 ##  
  
'Hello world!'  
  
1 + 1
```

In []:

```
## Code 2.1.3 ##  
  
print('Hello world!')  
  
1 + 1
```

In []:

```
## Code 2.1.4 ##  
  
print('Hello world!')  
  
1 + 1  
  
2 + 2
```

In []:

```
## Code 2.1.5 ##  
  
print('Hello world!')  
  
print(1 + 1)  
  
print(2 + 2)
```

2.2 Defining Objects

Several new things have happened in *Code 2.1.6*, including the declaration of three objects (*part_1*, *part_2*, and *part_3*). A good way to think of an object in Python is to think of a suitcase, or better yet, a magic suitcase.

Have you ever flown on an airplane where bringing any luggage other than a carry-on bag was prohibitively expensive (maybe even to the point where the baggage fee was higher than the cost of the actual suitcase you were trying to bring)? Have you, or someone you know, ever tried to avoid any additional baggage fees by filling your carry-on so full that you had to sit on it to get it to close? Have you ever experienced a situation where so many passengers had carry-on bags that there was not enough room in the overhead bins and someone had to put their bag under the plane, thus defeating the purpose of carry-on? If you said "yes" to any of these questions, I feel your pain. If we met in real life, I imagine we could have a detailed conversation on how we

would change airline luggage policies if we were in power.

However, what if instead of focusing on the airlines, we focused on the luggage itself? What if we sat down together and invented a suitcase that would adapt in size based on the amount of things you put into it? What if this suitcase could also magically disappear when you put it in the overhead bin, and would reappear only when you needed it? We would probably need to label our luggage very well so that it doesn't get confused with someone else's, but imagine how much more pleasant air travel would be if such magic suitcases existed.

Objects in Python can be thought of as magic suitcases. They automatically adapt in size, meaning they are exactly as large as they need to be, and no larger. They can also store anything that needs to be stored, and exist only when they are told to exist. They need to be labeled very clearly, as no two objects can have the same name, and object names should be intuitive enough so that we know what's being stored inside them. There are a few rules to naming objects in Python which can be found in [Section 2.3 of the official Python documentation \(https://docs.python.org/3/reference/lexical_analysis.html\)](https://docs.python.org/3/reference/lexical_analysis.html): object names can contain *"the uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9."*

2.3 Working with Optional Arguments

If you look closely at the result in *Code 2.3.1*, it is clear that a space has been added between *world* and the exclamation point. This is because the **print()** statement has an optional argument called **sep**, which stands for separator and has a default value of a single space. If you were unaware of this optional argument, please reference the **help()** documentation for the **print()** statement. If you have no idea how to do this, or are unsure what an optional argument is, please reference the content in **Chapter 1: Before Learning Anything Else, Learn This**.

We can alter this argument by overriding its default value (i.e. setting *sep* equal to something other than ' (space)'). This has been done in *Code 2.3.2*. There is no virtually limit to the way we can separate our **print()** statements. We can also override the default value for the **end** argument, which by default is set to start a new line after each individual **print()** statement. An example of this has been done in *Code 2.3.3*. In this code, the two print statements end with a *<tab>* and an arrow (*-->*), respectively. Overriding optional arguments emphasizes a critical coding

In []:

```
## Code 2.3.1 ##

part_1 = 'Hello'
part_2 = 'world'
part_3 = '!'

print(part_1, part_2, part_3)
```

In []:

```
## Code 2.3.2 ##

part_1 = 'Hello'
part_2 = 'world'
part_3 = '!'

print(part_1, part_2, part_3,
      sep = '') # no space

print(part_1, part_2, part_3,
      sep = '---') # dashes

print(part_1, part_2, part_3,
      sep = '<-*>') # being creative
```

practice:

Read the documentation.

Short, simple and sweet. Reading a function's documentation (i.e. its `help()` files and what's officially posted by the [Python Software Foundation](https://www.python.org/) (<https://www.python.org/>) is the best method to understand what a function is capable of doing. Regardless of the coding language you are trying to learn:

It is better to learn how to do fifty things with one function than to try and learn fifty functions that do relatively the same thing.

This is a philosophy that I live by. Keep this in mind

In []:

```
## Code 2.3.3 ##

part_1 = 'Hello'
part_2 = 'world'
part_3 = '!'

print(part_1, part_2, part_3,
      sep = ' ',
      end = '\n')

print(part_1, part_2, part_3,
      sep = ' ',
      end = ' --> ')

print(part_1, part_2, part_3,
      sep = ' ',
      end = ' <THE END> ')
```

2.4 Escape Sequences

By now, you have been working with several characters in Python that have special meanings (' , # , " , etc.). Such characters are appropriately named **special characters**. Oftentimes, we want to drop the special meaning of a character and use it as text. At other times, we want to take a character that Python normally interprets as text and activate its special meaning. If you're confused, have no fear. There is one simple tool to understand and everything else will come with practice. Essentially, Python has a built in character that we can think of as a light switch: it's job is to turn on or turn off the meaning of special characters.

Backslash (\), the Light Switch for Special Characters

In *Code 2.1.4*, we initiated an escape sequence using a backslash to turn off the special meaning of the apostrophe character ('). By placing a backslash immediately before any such character, Python will ignore its special meaning and interpret it as text. Likewise, the default interpretation of some characters is text, and placing a backslash immediately before such characters activates a special meaning. *Table 2.2* displays some of the most commonly used escape sequences and special meanings for our current level of programming. A complete list can be found in [the official Python documentation](https://docs.python.org/3/reference/lexical_analysis.html) (https://docs.python.org/3/reference/lexical_analysis.html).

Table 2.2: Common escape sequences.

Sequence	Description
\\	escapes the special meaning of the backslash (/)
\'	escapes the special meaning of the single quote wrapper
\"	escapes the special meaning of the double quote wrapper

<code>\n</code>	activates the special meaning of 'n' (new line)
<code>\t</code>	activates the special meaning of 't' (horizontal tab)

Emojis work in the same way.

If you're having trouble with the concept of special characters, think of emojis. *Figure 1.3* is a screen shot of a conversation I was having with myself in WeChat. That may seem like an odd thing to do, but let's look past this weirdness for the time being and talk about the special meaning of the string "Python" in the text.

As can be observed, when "Python" is typed, a special character pops up in one of the predictive text boxes. In this case, it is an emoji that looks like a snake. The same occurs when we type smile, ball, fish, or any of several other strings or phrases. In essence, WeChat is recognizing that the string we have typed can have more than one meaning and is giving us options to choose from. If we hit the *Send* button, WeChat will make a decision on which meaning to use. Sometimes, programs like WeChat will auto-correct our spelling. This enhanced functionality also utilizes the concept of default values.

Like WeChat, Python 3 makes several decisions based on default values. Unlike WeChat, however, it does not give options when a character has more than one meaning. Instead, each special character has a default value that can be overridden. This much like the optional arguments covered in **Section 1.1 of Chapter 1 - Setting Up for Success**. Our light switch (a.k.a. the backslash) allows us to override/activate special characters.

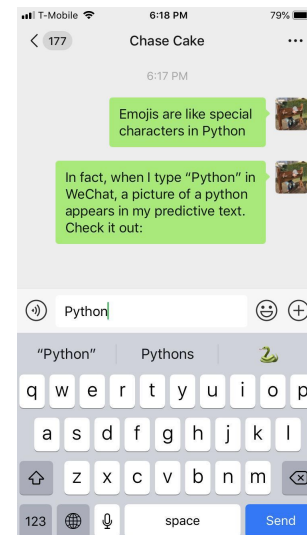


Figure 1.4: Python emoji example in WeChat



Figure 1.5: Smile emoji example in WeChat

2.5 Dynamic Strings

Up to this point, all of our strings have been static in that their components have been fixed. Given this, it would be incredibly hard to design even a simple program, such as one to tell us what time it is. This can be addressed with the use of **dynamic strings**, also known as formatted strings or f-strings. A **dynamic string** is a

string that contains objects (i.e. variables) or calculations. If the string's variables get updated, the string will adapt. These can come in very handy in several applications, such as dynamic reports, dashboards, personalized emails, and much more.

Code 2.5.1 contains a basic program designed to tell us the current time. When I wrote this program, the current time was 19:51. As one can imagine, this is only useful in a very limited context. Functionally, *Code 2.5.2* is equally useless, but exemplifies how to convert a standard *print()* statement into one that is dynamic. As of right now, *Code 2.5.2* creates no benefit beyond what is offered in *Code 2.5.1*, but it is one step closer to developing the program we intend to create.

In []:

```
▼ ## Code 2.5.1 ##  
  
time = '19:51'  
  
print('The time is:', time)
```

In []:

```
▼ ## Code 2.5.2 ##  
  
time = '19:51'  
  
print(f'The time is: {time}')
```

To develop the *time* object into one that varies based on the current time, we need to get into some code that can seem intimidating. However, this code (presented in *Code 2.5.3*) utilizes a very useful technique called **method chaining**.

In []:

```
▼ ## Code 2.5.3 ##  
  
# Importing datetime (where a function for the current time exists)  
import datetime  
  
# Method chaining  
time = str(datetime.datetime.now().strftime("%H:%M"))  
  
# Printing the result  
print(f'The time is: {time}')
```

Let's dissect *Code 2.5.3* piece by piece:

```
1 | ## Code 2.5.3 ##  
2 |  
3 | import datetime  
4 |  
5 | time = str(datetime.datetime.now().strftime("%H:%M"))  
6 |  
7 | print(f'The time is: {time}')
```


Line 3 - Importing datetime

The *datetime* package contains a number of useful functions for, well, manipulating dates and times. **You are not expected to memorize every detail of the *datetime* package.** Many packages have an incredible amount of methods embedded inside of them, and you will learn more about various methods as encounter new situations along your coding journey. Our present situation requires us to find a method that returns the current time.

Line 5 - Method Chaining

After importing *datetime*, we need to access the method *now()* that returns the current time. How did I know about this method? I searched on Google and noticed [this Stack Overflow thread \(https://stackoverflow.com/questions/415511/how-to-get-the-current-time-in-python\)](https://stackoverflow.com/questions/415511/how-to-get-the-current-time-in-python) in the search results. This method is deep within *datetime*, hence all of the dots ('.') being used to access it. When we call *now()*, it returns the exact current time, including the year, month, day, hour, minute, second, and microsecond. This is more detail than we need to develop our program, but we are on the right track. The method after the next dot (*strftime*) allows us to dissect the result of *now()* so that it only returns the current hour and minute. Through this process, we have successfully created a program to tell us the current time!

The process of linking methods together using dots is a technique called **method chaining**. It is similar to the chaining technique used in **Section 1.1 of Chapter 1 - Setting Up for Success**, and is extremely common in Python programming for business analytics. We will learn more about this in later chapters.

Line 7 - Printing the Result

Dynamic strings can be created in one of two ways. In *Code 2.5.3*, the letter 'f' that is placed immediately before the string in the *print()* statement tells Python that this code is meant to be dynamic. Objects to be referenced in dynamic strings need to be placed between curly brackets {}. Any time Python sees a set of curly brackets in a dynamic string, it treats whatever is inside as its own line of code. The second method for creating dynamic strings is the *.format* method, which can be found in [Section 6.1.2 of the Python documentation \(https://docs.python.org/3.4/library/string.html\)](https://docs.python.org/3.4/library/string.html).

Side Note: If you're trying to print curly brackets in a dynamic string, remember that you can use the backslash to escape special characters.

The Advantage of Dynamic Strings

Dynamic strings allow for more than just the inclusion of objects. They also allow programmers to:

- create less objects, thus enhancing the speed of our working environment
- better control the formatting of strings

Section 1.2 of Chapter 1 - Setting Up for Success discussed how our working environment gets slower as more objects and packages get loaded. Therefore, it is beneficial to minimize the amount of objects we create. *Code 2.5.4* is an adaptation of our program in which no objects are created. Instead, code for the *time* object has been embedded within the dynamic string. These minor enhancements can add up to a massive difference!

In []:

```
▼ ## Code 2.5.4 ##

# Importing datetime (where a function for the current time exists)
import datetime

# Method chaining and printing
print(f'The time is: {datetime.datetime.now().strftime("%H:%M")}')
```

Dynamic Strings and Triple-Quotes

Things start to get really interesting when we combine dynamic strings with triple quotes, allowing for the inclusion of objects formatted to our specifications. Below is an adapted version of *Code 2.5.4* that utilizes these two powerful tools.

To emphasize this, the focus of Interactive Workbook 1 is to create a dynamic report using this technique.

In []:

```
▼ ## Code 2.5.5 ##

# Importing datetime (where a function for the current time exists)
import datetime

# Formatted Method chaining and printing
print(
f"""
*****

Today is {datetime.datetime.now().strftime("%A, %B %d")}

▼ The current time is {datetime.datetime.now().strftime("%H:%M")}

        Have a nice day! :)

*****
""")
```

2.6 Summary

```
dP""b8 88""Yb 888888      db      888888      88888 dP"Yb 88""Yb d8b
dP  ` " 88__dP 88__      dPYb      88          88 dP  Yb 88__dP Y8P
Yb  "88 88"Yb 88""      dP__Yb      88          o. 88 Yb  dP 88""Yb `""
YboodP 88  Yb 888888 dP""""Yb      88          "bodP'  YbodP 88oodP (8)
```

You're two chapters deep! Now that we've covered some of the basics, it's time to move into some syntax that is a bit more interesting. By the time you finish the next chapter, you will be ready to build a basic application that interacts with users!

The next chapter will build upon this one, and help solidify your understanding of:

- the **print()** function
- objects in Python
- working with a function's optional arguments
- enabling and escaping the meanings of special characters
- developing **print()** statements that are dynamic

Chapter 3: Interacting with Users and Variable Types

Now that we have an understanding of printing and formatting, it's time to develop our way towards interacting with users. More specifically, this chapter focuses on syntax that allows users to **input()** information that can be used in our programs. This functionality is quite extraordinary as it grants us the ability to collect information and personalize the user experience accordingly.

After completing this chapter, you will be ready to build a basic application that interacts with users. Thus, this chapter concludes with a project that does just that. To get there, we will cover:

- prompting users for input
- converting object types
- writing more efficient code

Note: In **Chapter 1 - Before Learning Anything Else, Learn This**, it was mentioned that interactive help sessions can cause problems that most beginners are not ready to solve. This is due to the *interactivity* of interactive **help()**, as while this is running, your Python [kernel](https://ipython.org/ipython-doc/dev/development/kernels.html) (<https://ipython.org/ipython-doc/dev/development/kernels.html>) cannot move on to process other code. In this chapter, we will learn what to do in such situations.

3.1 Prompting Users

As its name implies, the **input()** function allows us to prompt users to give us input.

Objects created from **input()** allow for interactive and dynamic programs. Such a program has been developed in *Code 3.1.1*. In this code, we are using **input()** to prompt a user to input their name. When this code is run, a special window pops up under the coding cell to prompt users for input. It is important to understand what is happening here, as forgetting to respond to an open **input()** will prevent you from running additional code (*see the note at the beginning of this chapter for more details*). *Figure 3.2* outlines the anatomy of Jupyter Notebook's behavior when a user is prompted for **input()**.

Notice the star (*) status in *Figure 3.2*. This indicates that the kernel is currently busy. There are several reasons why this might occur, including an open user input prompt, a code that hasn't finished processing, or a loop that was never instructed to stop running (covered in a later chapter). As mentioned earlier, while the Python kernel is busy, you cannot move on to process other code. If you run into a situation in Jupyter Notebook where the star won't go away, you can interrupt the kernel by pressing *ESC + i*.

Note: If you're interested in learning more hotkeys for Jupyter Notebook, try pressing *ESC + h*.

The second red box in the lower part of *Figure 3.2* is the **user input prompt**. In the case of the figure, anything typed into this prompt will be stored as a **string** in the object *name*. Strings and other variable types will be covered in *Section 3.2*. To reiterate, the Python kernel will continue processing this code block until something is input into this prompt (even something as simple as pressing the *return* on the keyboard). If you accidentally move through your code too quickly and it stops giving you output, look for the star (***) in previous coding blocks and check to see if there is an open user prompt.

Concept Clarification - Kernels

If you're having trouble conceptualizing a kernel, think of it as an escalator. That's right, an escalator.

escalator - power-driven set of stairs arranged like an endless belt that ascend or descend continuously. - [Merriam-Webster \(https://www.merriam-webster.com/dictionary/escalator\)](https://www.merriam-webster.com/dictionary/escalator)

Let's say, for example, people are in a straight line walking towards an escalator to go upstairs. For the sake of this argument, let's assume the escalator is only wide enough to fit one person per step. If all goes according to plan, the line of people will reach the top of the escalator in the order in which they started. However, what if the first person in line reached the bottom of the escalator, was a split second from stepping on, and then realized that they were afraid of riding escalators? According to [fearof.net \(https://www.fearof.net/fear-of-escalators-](https://www.fearof.net/fear-of-escalators-)

In []:

```
## Code 3.1.1 ##  
  
name = input()  
  
print(f'Welcome {name}')
```

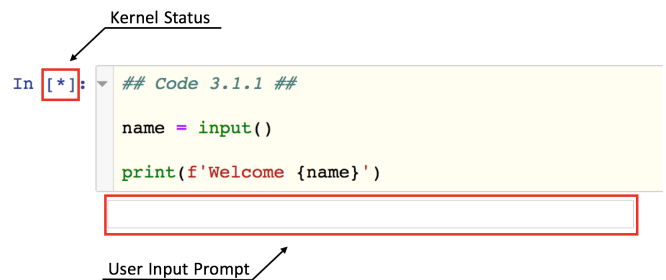


Figure 3.1: the anatomy of Jupyter Notebook's behavior when a user is prompted.

[phobia-escalaphobia/](#)), this is a legitimate phobia known as Escalaphobia, and it affects hundreds of thousands of individuals globally. In this situation, every person in the entire line is stuck until the first person steps on the escalator or moves out of the way.

This is how kernels work, and it brushes upon an important mathematical concept known as [Queueing Theory](https://en.wikipedia.org/wiki/Queueing_theory) (https://en.wikipedia.org/wiki/Queueing_theory). Think of the escalator like a pathway to your processor, and only one line of code can go through the processor at a time. If something is holding up the line, such as in the case of a code block waiting for user input, no code in line behind it can move forward.

Optional Arguments for input()

`input()` has only one optional argument, as can be observed from the `help()` call in *Code 3.1.2*. As a refresher, an optional argument is one that has a default value, denoted with an equals sign (`=`). As can be observed in *Codes 3.1.3* and *3.1.4*, we can utilize this argument by calling its name and providing a value (i.e. `prompt = 'some_text'`), or we can simply type some text and Python will understand what we are trying to do (i.e. `'some_text'`). As its name implies, the optional `prompt` argument *prompts* a user on what they should do.

Do not underestimate the value of an informative prompt.

You cannot expect a user to intuitively understand what they are supposed to input into a prompt. If you require a number with no decimal places, make sure to inform users that this is required. If you require a date in a specific format, make sure to use your **prompt** accordingly. Users are free to put anything into an `input()` prompt, so your directions are incredibly important. We will learn how to control for correct user input in a later chapter.

In *Codes 3.1.3* and *3.1.4*, we have specified that a user should input their name. Notice how this message is followed by a space. Without the trailing space, the output would look like this:

```
Input your name:Python
Welcome Python
```

Figure 3.2: Prompt result with no trailing space.

As can be observed below, the user experience can be slightly improved with the trailing space. This may seem trivial, but little details such as this can make a huge difference in how your program is received by its audience.

In []:

```
## Code 3.1.2 ##

help(input)
```

In []:

```
## Code 3.1.3 ##

# Specifying prompt
name = input(prompt = 'Input your name: ')

print(f'Welcome {name}')
```

In []:

```
## Code 3.1.4 ##

# Not specifying prompt
name = input('Input your name: ')

print(f'Welcome {name}')
```

```
Input your name: Python
Welcome Python
```

Figure 3.3: Prompt result with a trailing space.

3.2 Type Conversions

As stated earlier, a user is free to input anything into an **input()** prompt. Given this, Python needs to be very careful not to make assumptions as to what we are trying to achieve with any given prompt. To address this concern, Python treats data from an **input()** prompt as a [string](https://docs.python.org/3/library/stdtypes.html#str) (<https://docs.python.org/3/library/stdtypes.html#str>). Simply put, a string is a set of characters that can be read as text. This is very important to understand, as even if a number were input into an **input()** prompt, Python would treat it as a string instead of making assumptions on how a program will use this information. This can create bugs if we are not careful. For example, in the code below, we are prompting a user to **input()** a number and then attempting to make a calculation with their input.

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

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

double = number * 2

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

As the object *number* is coded as a string, Python interprets our meaning of the calculation in the object *double* as us wanting to print *number* twice (Code 3.2.1). Although this is not what we intended, such functionality can be quite useful. For example, if we wanted to create a border around our output, as in Code 3.2.2, we could take advantage of how Python interprets strings in multiplication. Such utilization can be observed in Code 3.2.3.

In []:

```
▼ ## Code 3.2.1 ##

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

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

double = number * 2

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

In []:

```
## Code 3.2.2 ##

print("""
*****

Happy birthday!!!!

*****
""")
```

In []:

```
## Code 3.2.3 ##

print(f"""
{'*' * 40}

Happy birthday!!!!

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

As can be observed, *Codes 3.2.2 and 3.2.3* generate the same output. Utilizing string calculations for formatting is especially intriguing when considering how easy it is to ensure we have the same number of stars above and below our *Happy Birthday!!!!* message. As our coding skills become more advanced, we will also see that such a practice can save a huge amount of time when debugging.

Basic Object Types

There are an ever-growing number of object types in Python. This is because Python programmers have an ever-growing number of needs. For example and as a preview of what's to come, the data science package *pandas* is home to the *DataFrame* object type, which is Python's version of an Excel spreadsheet. Looking further out, when instantiating a model object in the machine learning package *scikit-learn*, each model object is of its own type. The rationale behind having so many different object types is that it allows each object to behave the way its programmer intended. You can think of an object type like a template, or even better, a tool to perform a specific task. Some tools are flexible and can perform a diverse set of tasks, while others were designed to solve one and only one need. This is another great example as to why you should read the documentation. The [documentation on data types](https://docs.python.org/3/library/datatypes.html) (<https://docs.python.org/3/library/datatypes.html>) can be bewildering for first-time coders, but it contains a wealth of information as to what each object is designed to do, as well as useful methods to help you use each type to its fullest extent.

Below is a table of the most basic object types. Also note that when in doubt, you can wrap **type()** around an object, and as its name implies, Python will output the object's type. *Code 3.2.4* is an open coding block

In []:

```
## Code 3.2.4 ##

# Open coding block for type() discovery

"""
Objects created thus far:
    + name
    + number
    + double

Use this space to check the types of each of
"""
```



Show Solution

where you can call type on any of the objects we have created in the above codes.

Note: Make sure you have run the coding cells above. If you get an error stating that an object is not defined, it generally means that you forgot to run the coding cell where that object was created.

Wrapper	Interpretation
str()	string - changes object type to text
int()	integer - a number with no decimal places
float()	float - a number with decimal places
bool()	boolean - True or False

In []:

```
## Code 3.2.5 ##

number = input("""
What is your favorite number between 1 and
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 {doub
```

Code 3.2.5 is a replica of Code 3.2.1, with one enhancement: the object *number* is being converted into an integer before the object *double* makes its calculation. Now the program functions in the way we intended! Let's move on to a more advanced example.

3.3 Avoiding the Unnecessary - Writing Efficient Code

In Code 3.2.4, we are attempting to create a code to be used when someone has a birthday.

In this code, we would like to:

- Prompt a user to input their birthday
- Display a happy birthday message, which includes the user's input
- Convert the user's age into days, hours, minutes, and seconds
- Display True if a user's age is even

Code 3.3.1 meets these requirements, assuming a user input their age as instructed by the **input()** prompt. However, the code is a bit inefficient. In particular, it does not take advantage of string calculations, and has several **int()** type conversions spread throughout the code when just one could be used to achieve the same result. Also, the format of the output looks as if it could be improved. The output for days looks appropriate, as depending on the

user's input, it displays trailing decimal places (according to [NASA](https://pumas.jpl.nasa.gov/examples/index.php?id=46) (<https://pumas.jpl.nasa.gov/examples/index.php?id=46>), a year is approximately 365.25 days long).

However, the results for hours, minutes, and seconds are showing a trailing zero. We can clean this up by implementing an `int()` wrapper around each calculation within the curly brackets in our `print()` statement. To keep our code consistent, we should also put a wrapper around the calculation for *days*, even though it is already functioning properly. This is to help avoid unknown errors, and to also help inform other coders of our intended output.

Such enhancements have been made in *Code 3.3.2*. As can be observed, the *age* object is being converted into an integer immediately after it is defined. Notice how this object does not need to be converted back into a string in the line of code reading *Happy birthday you beautiful...*. Python is doing this for us behind the scenes of the `print()` statement. Design decisions such as this one help make Python a truly remarkable language.

The star borders of the birthday message have been recoded using string calculations. Also, notice the enhanced functionality of the exclamation point in the birthday message. With this addition, the message will print one exclamation point for every year since the user's birth. Finally, the conversions to days, hours, minutes, and seconds have been wrapped with their appropriate type conversion wrappers.

There are many ways to further enhance our code. For example, if a user were to input data of type string, float, or boolean in the `input()` prompt instead of an integer, the *Code 3.3.2* would throw a *Value Error*. It is important to control for such errors, which is a concept known as [exception handling](https://docs.python.org/3/tutorial/errors.html) (<https://docs.python.org/3/tutorial/errors.html>). However, we will save this conversation for a later point in our coding journey.

In []:

```
## Code 3.3.1 ##

# Age input
age = input("""
I hear it's your birthday! How old are you
Please input your age (as a number) in year
""")

# Birthday message
print(f"""
*****

Happy birthday you beautiful {age} year old

That's the equivalent of:
{int(age) * 365.25} days
{int(age) * 365.25 * 24} hours
{int(age) * 365.25 * 24 * 60} minutes
{int(age) * 365.25 * 24 * 60 * 60} seconds

It's {bool(age)} that your age in years is

*****
""")
```

In []:

```
▼ ## Code 3.3.2 ##

# Age input
age = input("""
I hear it's your birthday! How old are you
Please input your age (as a number) in year
""")

# Age conversion
age = int(age)

# Birthday message
print(f"""
{'*' * 40}

Happy birthday you beautiful {age} year old

That's the equivalent of:
{float(age * 365.25)} days
{int(age * 365.25 * 24)} hours
{int(age * 365.25 * 24 * 60)} minutes
{int(age * 365.25 * 24 * 60 * 60)} seconds

It's {bool(age)} that your age in years is

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

Upon critiquing *Code 3.3.2*, you may wonder if the line displaying *True* when a user's age is even adds any value. It seems intuitive that we can assume a user old enough to use a computer will also be at a mathematical level where they have a solid understanding of even and odd numbers. Perhaps we should alter this line of code to provide more useful information, such as whether or not a user was born in a leap year. Although this requires a more detailed knowledge of Python than we have currently covered, it is imperative to consider the following:

It is better to build something that is challenging to code than to build something that is easy but adds no value.

At this point, our coding abilities are limited. However, we should not let this dictate our design. Encountering a coding challenge that is beyond our current capabilities is a prime opportunity to improve. Spend some time doing research. If needed, go back to **Chapter 1: Before Learning Anything Else, Learn This** and refresh yourself on how to learn Python. For now, we will move forward to develop our code to be more valuable to users.

Adding Additional Functionality

In addition to adding a leap year calculation, it would be interesting to more precisely state the number of days in a year. Returning to research from [NASA \(https://pumas.jpl.nasa.gov/examples/index.php?id=46\)](https://pumas.jpl.nasa.gov/examples/index.php?id=46), "the true length of a year on Earth is 365.2422 days." Before reading on, use the open coding block below to try to adjust *Code 3.3.2* to meet these new requirements:

- adjust the number of days per year to 365.2422
- change the 'age in years is even' functionality to one that returns True if a user was born in a leap year
- add any other changes you feel could add value (such as calculating a user's birth year based on their age)

When ready, move on to see a sample solution in *Code 3.3.5* and read the rationale behind it.

In []:

```

## Code 3.3.4 ##

# Open coding block to meet new requirements

# Age input
age = input("""
I hear it's your birthday! How old are you today?
Please input your age (as a number) in years.\t
""")

# Age conversion
age = int(age)

# Birthday message
print(f"""
{'*' * 40}

Happy birthday you beautiful {age} year old{'!' * age}

That's the equivalent of:
{float(age * 365.25)} days
{int(age * 365.25 * 24)} hours
{int(age * 365.25 * 24 * 60)} minutes
{int(age * 365.25 * 24 * 60 * 60)} seconds

It's {bool(age)} that your age in years is an even number.

{'*' * 40}
""")

```

Note: The following code may seem intense, especially when compared to *Code 3.3.2*. Adding a check to see if the user was born in a leap year was not in our original requirements. Thus, much of *Code 3.3.2* needs to be rewritten. This is commonly the case, and emphasizes how important it is to plan out what you would like your code to do ahead of time.

In order to achieve the functionality we desire, several changes need to occur. First, we need to develop a method for calculating someone's birth year given their age. This can be engineered by taking the current year and subtracting the users age. Luckily, a method already exists in the subpackage **date**, which is part of the **datetime** package. Here we can find the method **today()** and chain it to the method **year**. When deciding how to determine if a user was born in a leap year, we have two choices: 1) develop a logic in our code, or 2) use a code that is already developed and available in Python. This is a classic make or buy decision, except that the open source packages in Python are free of charge. Therefore, before attempting to develop our own logic, it is in our best interest to search to see if someone has already solved this challenge. If you are currently unaware of such functionality, take some time to search before moving on.

After some research, you may have discovered that

the **calendar** package has just the method we are looking for (**isleap**). This function returns True if a year is a leap year, and False otherwise. If you were unable to find this in your search, or if you stumbled onto forums where nobody had yet figured this out, this is a good opportunity for you to go back and contribute to the conversation. With **date.today().year** and **isleap**, we are ready to adjust our code.

In addition to these enhancements, we should allow the calculations for hours, minutes, and seconds to **float**, or contain decimal places. Converting objects into floats can have strange formatting results, such as several trailing zeros followed by a seemingly random number (*Code 3.3.4*).

This formatting issue can be resolved using a **round()** wrapper. As can be observed from its **help()** documentation, **round()** has one mandatory argument (a number) and one optional argument (the number of digits to round to). By utilizing this wrapper, our output looks cleaner, providing a better experience for our users. A redeveloped solution, enhanced based on the new requirements, is available in *Code 3.3.5*.

In []:

```
## Code 3.3.4 ##

age = 5

print(f"""
{float(age) * 365.2422} days
""")
```

In []:

```
▼ ## Code 3.3.5 ##

# Importing necessary modules
from calendar import isleap
from datetime import date

# Saving the current year
current_year = date.today().year

# Saving a precise days per year
days_in_year = 365.2422

# Prompting a user for their birth year
age = input("""
I hear it's your birthday! How old are you today?
Please input your age (as a number) in years.\t
""")

# Converting age
age = int(age)

# Calculating birth year
b_year = current_year - age

# Birthday message
print(f"""
{'*' * 40}

Happy birthday you beautiful {age} year old{'!' * age}

That's the equivalent of:
{round(float(age * days_in_year), 4)} days
{round(float(age * days_in_year * 24), 4)} hours
{round(float(age * days_in_year * 24 * 60), 4)} minutes
{round(float(age * days_in_year * 24 * 60 * 60), 4)} seconds

You were born in {b_year}, and it is {isleap(int(b_year))}
that you were born in a leap year.

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

3.4 Summary

— — — — —
| | | | |
— — — — —
| | | | |
— — — — —
| | | | |

You're one step closer to being able to design interactive programs! The basics are out of the way, and you're ready to move on to your first project. It's going to be mad fun!

This chapter has prepared you for:

- prompting users for input
- converting object types
- making your code more efficient

We have also focused our design decisions on creating an optimal user experience. Although this is not directly related to the work as a business analyst, it is an important consideration when designing any sort of technical solution. Later, when we discuss data exploration and insights, this mindset will become critical to success. Even the most groundbreaking data-driven discoveries are likely to be discarded if they are hard to interpret or infeasible to implement. There's plenty to look forward to in the coming chapters, and it is my hope that upon completion of this book, you will not only be capable in business analytics using Python, but also in driving analytics as a core strategic component of business strategy.

