Chapter 1
Install Python via Anaconda and Spyder

Even if you have never coded before, this chapter will guide you through to install the Python software you'll need for the book to start running Python scripts. We'll be using Anaconda and Spyder. We'll discuss the advantages of making this choice.

No matter whether you are using Windows, Mac, or Linux, we'll guide you through the installation process based on your operating system. Once done, you'll learn how to start coding in the Spyder editor. We'll discuss basic Python rules and operations after that.

Introducing Anaconda and Spyder

There are many different ways to install Python and run scripts, in this book we'll use Anaconda and Spyder. Anaconda is an open-source Python distribution, package and environment manager. It is user-friendly and provides for the easy installation of many useful Python modules that can be otherwise quite a pain to compile and install yourself. We start by downloading the Anaconda distribution of Python that comes bundled with Spyder. Spyder is a full-featured integrated development environment (IDE) for actually writing scripts. Spyder comes with many useful features such as automatic code completion, automatic debugging, code suggestions, and warnings.

Install Anaconda and Spyder

Python is a cross-platform programming language, meaning that you can run Python scripts no matter which operating system you have (whether you use Windows, Mac, or Linux). However, the installation of software and modules can be slightly different. I'll show you how to install different modules in your operating system. Once these are properly installed, Python code works the same in different operating systems.

Install Anaconda and Spyder in Windows

To install Anaconda in Windows, go to *https://www.anaconda.com/products/individual* and download the latest version of Python 3 for Windows.

I recommend using the graphical installer instead of the command line installer, especially for beginners, to avoid mistakes. Make sure you download the appropriate 32- or 64-bit package for your machine. Run the installer and follow the instructions all the way through.

Find and open the Anaconda navigator, and you should see a screen like Figure 1-1 (if you need to, search for *Anaconda navigator* in the search bar).

INSERT FIGURE 1-1 HERE

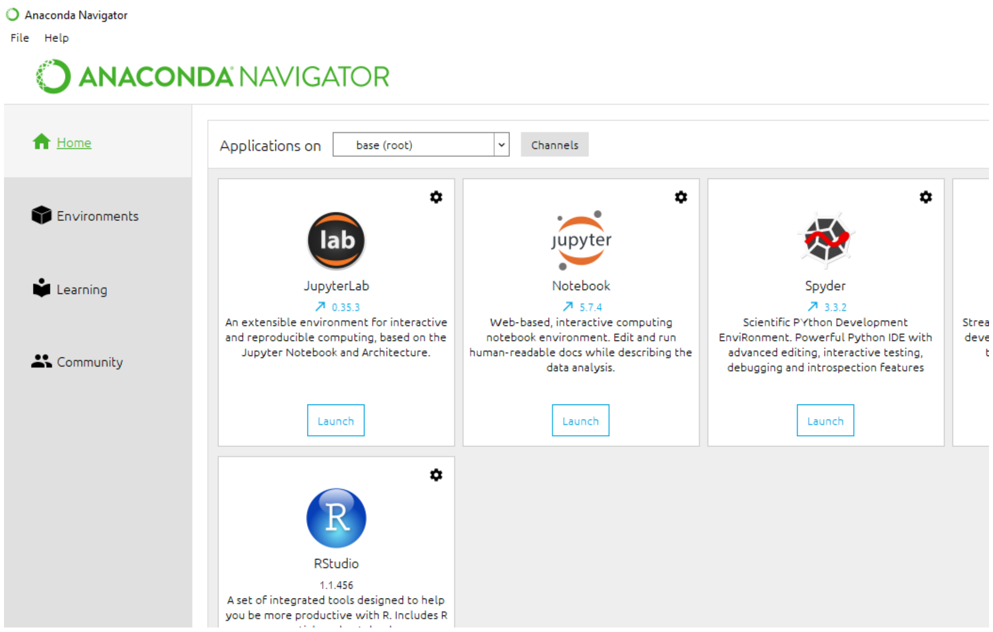

Anaconda Navigator
File Help
ANACONDA NAVIGATOR
Home
Environments
Learning
Community
Applications on
base (root)
Channels
JupyterLab
0.35.3
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.
Launch
Notebook
5.7.4
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch
Spyder
3.3.2
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch
RStudio
1.1.456
A set of integrated tools designed to help you be more productive with R. Includes R

Figure 1-1 The Anaconda navigator

Click the **Launch** button under the Spyder icon. If Spyder is not already installed, click **Install** to install the Spyder development environment, then when it finishes, click **Launch**.

Install Anaconda and Spyder in Mac OS

To install Python via Anaconda for macOS, go to *https://www.anaconda.com/products/individual*, scroll down and download the latest version of Python 3 for Mac. Choose the graphical installer and follow the instructions through.

Open the Anaconda navigator by searching for *Anaconda navigator* in Spotlight Search. The screen for the Anaconda navigator in Mac OS should look something like Figure 1-1, perhaps with slight differences.

To launch Spyder, click the **Launch** button under the Spyder icon (if you see an **Install** button instead, click it to install Spyder first).

Install Anaconda and Spyder in Linux

The installation of Anaconda and Spyder in Linux involves more steps than for other operating systems. First, go to *https://www.anaconda.com/products/individual*, scroll down and find the latest Linux version. Choose the appropriate x86 or Power8 and Power9 package. Click and download the latest installer bash script. For example, the installer bash script during my installation was *https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh*. This link may change over time, but we'll use this version as our example below.

By default, the installer bash script is downloaded and saved to the *Downloads* folder on your computer. You should then install Anaconda as follows:

```
bash ~/Downloads/Anaconda3-2020.02-Linux-x86_64.sh
```

After pressing the ENTER key, you'll be prompted to review and approve the license agreement. The last question in the installation process is as follows:

```
installation finished.
Do you wish the installer to prepend the Anaconda3 install location to PATH
in your /home/mark/.bashrc ? [yes|no]
[no] >>>
```

You should type in `yes` and press the ENTER key in order to use the `conda` command to open Anaconda in a terminal.

WARNING: Since the default choice is `no` *in this step, it's easy to make a mistake by pressing the ENTER key without typing in* `yes`*. If that were to happen, enter the following command in the terminal:*

```
gedit /home/{your user name here and file path}/.bashrc
```

Here you need to enter your actual user name in the path. My username is `mark`*, so the full path is* /home/mark/.bashrc*. Once you execute this line of command, the* .bashrc *file should open, and you should enter this next line as a new line at the end of the file:*

```
export PATH=/home/{your user name here}/anaconda3/bin:$PATH
```

Then save and close the file.

Now you need to activate the installation by executing the following command line:

```
source ~/.bashrc
```

To open Anaconda navigator, enter the following command in a terminal:

```
anaconda-navigator
```

You should see the Anaconda navigator on your machine, similar to Figure 1-1. To launch Spyder, click the **Launch** button under the Spyder icon (if you see an **Install** button instead, click it to install Spyder first).

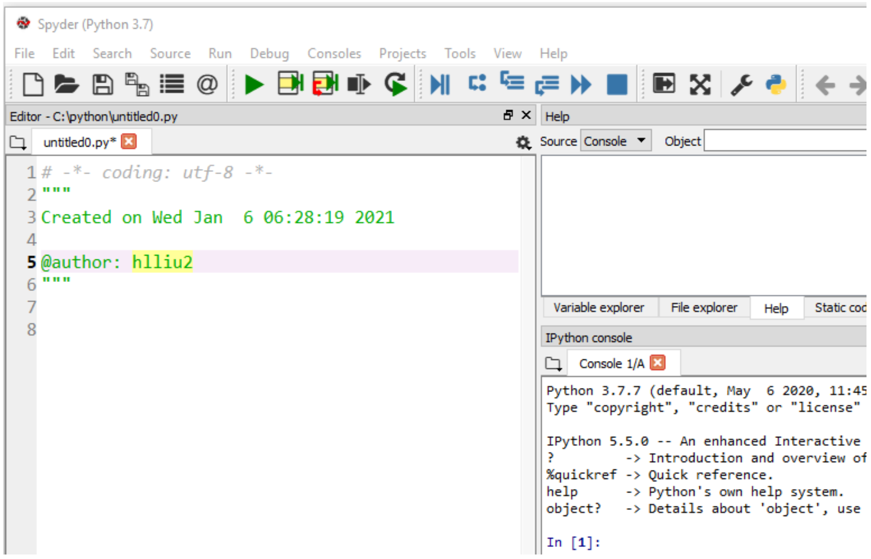
Using Spyder

To get you up and running, we'll build a really simple script in Spyder, then I'll run through a few basic things that'll be useful to know before you start coding for real.

Writing Python in Spyder

As mentioned earlier, Spyder is a full-featured IDE. Let's start with a simple script After you launch the Spyder development environment, you should see a layout like Figure 1-2.

INSERT FIGURE 1-2 HERE



```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  6 06:28:19 2021

@author: hlliu2
"""
```

```
Python 3.7.7 (default, May  6 2020, 11:45
Type "copyright", "credits" or "license"

IPython 5.5.0 -- An enhanced Interactive
?         -> Introduction and overview of
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use

In [1]:
```

Figure 1-2 Figure 1-2: Spyder development environment

The default layout in Spyder has three panels. Spyder comes with several pre-defined layouts, and you can customize layouts according to your preferences. Let's examine the default layout. The left is the Spyder editor in which you can write Python code; the top right is the variable explorer that shows the details of the data generated by your program. As programs become quite complicated, the variable explorer becomes a valuable asset in double checking the values stored in your variables.

The right bottom is the interactive Python (IPython) console that shows the output of the script or execute snippets of python code. The IPython console is also where you would enter input for programs that require user information. It also displays error messages if you make a mistake in your program.

Now let's start coding. Go to the Spyder editor window (default location is on the left) and enter:

```
print("This is my very first Python script!")
```

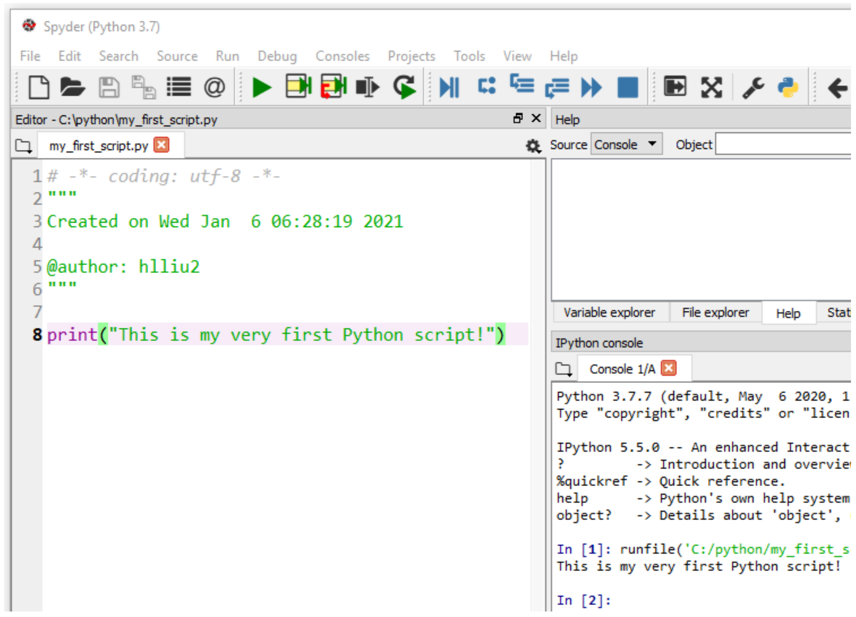
Click **File** ▸ **Save As** and save the file as *my_first_script.py* in your project folder.

There are three ways to run the scripts and they all lead to the same outcome:

1. Go to the **Run** menu and select Run.
2. Press F5 on your keyboard.
3. Press the green triangle icon ▶ in the icons bar.

Run the script and you should see something like Figure 1-3.

INSERT FIGURE 1-3 HERE


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\python\my_first_script.py
my_first_script.py
Help
Source Console Object
Variable explorer File explorer Help Stat
IPython console
Console 1/A

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  6 06:28:19 2021

@author: hlliu2
"""

print("This is my very first Python script!")
```

```
Python 3.7.7 (default, May  6 2020, 1
Type "copyright", "credits" or "licen

IPython 5.5.0 -- An enhanced Interact
?         -> Introduction and overvie
%quickref -> Quick reference.
help      -> Python's own help system
object?   -> Details about 'object',

In [1]: runfile('C:/python/my_first_s
This is my very first Python script!

In [2]:
```

Figure 1-3 Figure 1-3: Run a script in the Spyder development environment

The output is shown in the IPython console, and as you can see, is a simple printed message that says, “This is my very first Python script!” Congratulations, you have written and successfully run your first Python script!

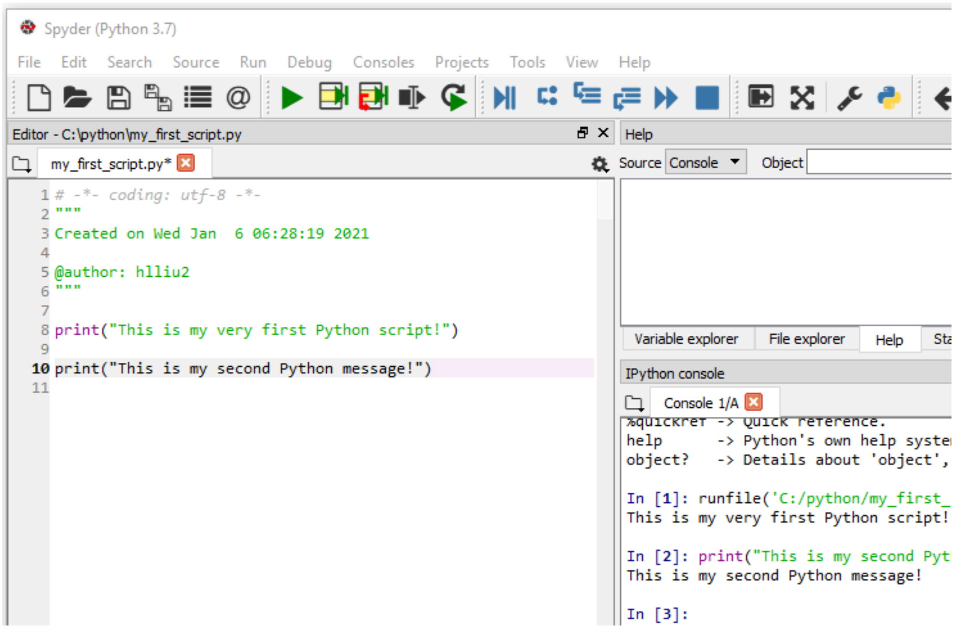
Inspecting Code in Spyder

Spyder has the ability to run code line by line, block by block, or entire scripts at a time. This is useful for carefully following the execution of a script, to verify that it does exactly what you intended for it to do. Go back to the *my_first_script.py* example and add another line:

```
print("This is my second Python message!")
```

Place your cursor over this second line and press F9, and you should see the output shown in Figure 1-4.

INSERT FIGURE 1-4 HERE


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\python\my_first_script.py
my_first_script.py*
Help
Source Console Object
Variable explorer File explorer Help Sta
IPython console
Console 1/A

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  6 06:28:19 2021

@author: hlliu2
"""

print("This is my very first Python script!")

print("This is my second Python message!")

```

```
%quickref -> Quick reference.
help      -> Python's own help syste
object?   -> Details about 'object',

In [1]: runfile('C:/python/my_first_
This is my very first Python script!

In [2]: print("This is my second Pyt
This is my second Python message!

In [3]:
```

Figure 1-4 Figure 1-4: Run just one line of code in Spyder editor

As we can see, only the highlighted line is being executed. The output is:

```
This is my second Python message!
```

Now press the F5 button, and you'll see that every line in the program is executed:

```
This is my very first Python script!
This is my second Python message!
```

To run a particular block of code, highlight the lines that belong to the block and press F9.

Coding in Python

Before we get into the basic coding concepts of Python, there are a few general things to understand. First, Python is case sensitive. You should take great care when it comes to upper and lower cases. Variables *X* and *Y* are different from variables *x* and *y*. A string "Hello" and a string "hello" are also different from each other.

Second, indentations are significant in Python. Non-printing characters like tabs and spaces must be consistently applied throughout a script. If you have experience with another programming language like C or Java, you may notice the lack of brackets and semi-colons, this is by design. Blocks of code are defined by indentation. An unintended space in the code will likely betray your intentions, as we'll see later in this chapter.

Third, Python uses single quotation marks and double quotation marks (mostly) interchangeably. For example, placing a sequence of characters inside single quotes has the same effect as if we put them in double quotes (unless one of the characters is an escape character, or a single quote).

Fourth, Python lets you make notes, known as *comments*. One popular way to write a comment uses the pound sign #. Everything in the same line after # will not be executed. It's good practice to make notes in your scripts so others can more easily understand what the code is doing, and to remind yourself of the decisions you make when you revisit the code after a few weeks or a few months. For example, in the very first line in the script *my_first_script.py*, we have

```
# -*- coding: utf-8 -*-
```

This is purely for information purpose, and Python ignores this line since it starts with #.

When you have a comment that can't fit in one line, you can place the comment in triple quotation marks """, and everything between the first set of quotes and the last set will not be executed by the Python program. For example, in lines 2 to 6 in the script *my_first_script.py*, we have:

```
"""
Created on Wed Jan  6 06:28:19 2021

@author: hlliu2
"""
```

All those lines are ignored by Python.

Basic Operations in Python

Python is capable of basic math operations. For example, to calculate 7 multiplied by 123, you enter the following in the Spyder editor:

```
print(7*123)
```

Place your cursor in this line and press F9, you will get an output of 861.

Table 1-1 provides the other basic math operations in Python.

Table 1-1 Table 1-1: Basic math operators.

Operator	Action
+	Addition; print(5+6) will give you a result of 11;
-	Subtraction; print(9-4) will give you a result of 5;
/	Division; print(9/3) will give you a result of 3
**	Exponent; print(5**3) will give you a result of 125
%	Remainder; print(13%5) will give you a result of 3 because 13=5*2+3
//	Integer quotient; print(13//5) will give you a result of 2 because 13=5*2+3

These operations have a precedent, meaning they will execute in a particular order. That order of operations is as follows: parentheses have highest priority, followed by exponents, then multiplication and division, which have the same priority and are executed from left to right. Addition and subtraction have the least priority and are treated equally, so whichever comes first is executed first.

For more complicated mathematical operations such as cosine in trigonometry or the natural logarithm, we need to import modules. In Chapter 2, you'll learn what modules are and how to import them.

Variables

In Python, we assign values to variables. There are four types of variables: floats, strings, integers, and Booleans. Let's talk about them one by one.

Strings:

A string variable is a sequence of characters inside quotation marks. Python is using single or double quotation marks interchangeably.

Here are some examples:

```
Name1 = 'University of Kentucky '
Name2 = "Gatton College"
```

You can find out the type of a variable by using the `type()` function. Enter the following two lines of code in the Spyder editor:

```
print(type(Name1))
print(type(Name2))
```

After execution, you will see the following output:

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

This means both variables are string variables.

You can add or multiply string variables, but not in the traditional mathematical sense. For example, if you run the following two lines of code in the Spyder editor:

```
print(Name1+Name2)
print(Name1*3)
```

You will see the following output:

```
University of Kentucky Gatton College
University of Kentucky University of Kentucky University of Kentucky
```

The plus sign joins two strings together, while multiplying a string by 3 means repeating the characters in the string three times. Note that I deliberately leave an empty space at the end of the variable *Name1*, so that when they join together, there is a space between them.

Floats

The second type of variables are float point numbers (or just floats). This is the equivalent of decimal numbers in mathematics. Here are two examples of floats:

```
x=17.89
y=0.987
```

You can use the `round()` function to decide how many digits you want to keep after the decimal point. Run the following code:

```
print(type(x))
print(type(y))
print(round(x,3))
print(round(y,1))
```

You will have the following output:

```
<class 'float'>
<class 'float'>
17.890
1.0
```

Integers

The third type of variables are integers. Integers are used mainly for indexing purpose in Python. Integers can be either positive, negative, or zero. Here are some examples of integers:

```
a=7
b=-23
```

```
c=0
```

It is important to know that floats always have decimals with them, while integers do not. This is the way for Python to distinguish the two types of variables. Even if you round a float number to 0 digits after decimal, there is still a decimal point and a 0 trailing the number. Run the following code:

```
print(type(a))
print(type(b))
print(type(c))
print(round(7.346,1))
print(round(7.346,0))
```

and you will have the following output:

```
<class 'int'>
<class 'int'>
<class 'int'>
7.3
7.0
```

You will not get an output of 7 from `print(round(7.346,0))` because this is Python's way of separating the integer from the float to help us tell them apart.

Bools

There is a fourth variable type: Booleans, or bools, which are binary variables that can take the value of `True` or `False`. Note that we need to capitalize the first letter in `True` or `False`. Here are some examples. Run these two lines of code:

```
print(4> 5)
print(10>= 6)
```

You will get the following output:

```
False
True
```

The above results show that the logic statement `4>5` is `False` while the logic statement `10>=6` is `True`. Again, bool variables can take only a value of `True` or `False`, which are special values. They are not string variables. Consider the following command lines:

```
print('4>5')
print(type(4>5))
print(type('4>5'))
```

The output will be

```
4>5
<class 'bool'>
<class 'str'>
```

As you can see, once you put `4>5` inside quotation marks, it becomes a string variable instead of a logical statement.

Convert Variable Types

You can covert variable types by using functions `str()`, `int()`, `bool()`, and `float()`, if they are convertible to begin with. For example, you can covert the string variable *"17"* to either an integer or a float by using `int("17")` or `float("17")`. However, you cannot covert the string variable *"Kentucky"* to either an integer or a float.

Consider the following lines of code:

```
print(int(17.0))
print(int("88"))
print(int(3.45))
print(str(17.0))
print(float(-4))
```

The outputs from them will be

```
17
88
3
'17.0'
-4.0
```

Bool values `True` and `False` can be converted to integers 1 and 0, respectively. The code

```
print(int(True))
print(int(False))
print(float(True))
print(str(False))
```

will lead to the following output:

```
1
0
1.0
'False'
```

The `bool()` function converts any nonzero value to `True`, and 0 to `False`. Run the following lines of code:

```
print(bool(1))
print(bool(-2))
print(bool(0))
print(bool('hello'))
```

and you will get:

```
True
True
False
True
```

Rules for Variable Names

Not everything can be used as variable names. You must follow the following rules.

A variable name must start with a letter (either upper or lower case) or an underscore ("_"). For example, you cannot use *8python* as a variable name because it starts with a number.

Special characters such as @ or & cannot appear in the variable name, except the underscore.

NOTE: Here is a list of password special characters https://www.owasp.org/index.php/Password_special_characters.

Variable names cannot be Python keywords or Python built-in functions. To find out the list of all keywords, run these two lines of code in the Spyder editor:

```
from keyword import kwlist
print(kwlist)
```

and you will have the following output:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

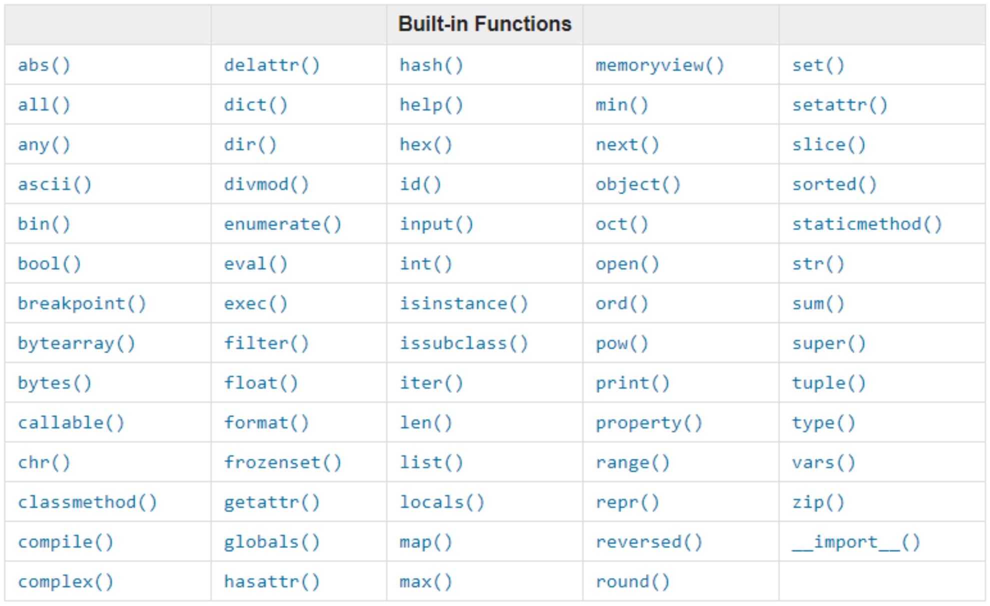
which is the complete list of Python keywords.

Variable names cannot be Python built-in functions. The list of those functions are shown in Figure 1-5, which is obtained from this website on Python documentations:

https://docs.python.org/3/library/functions.html

More information about Python built-in functions and their definitions can be found on the above website.

Insert Figure 1-5 here



		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Figure 1-5: List of Python built-in functions

Loops and Conditional Execution

You'll learn conditional execution and loops in this section.

Conditional Execution

The `if` statement in Python allows you to decide which actions to take, based on whether a condition is met or not. Consider the following lines of code:

```
x=5
if x>0:
    print('x is positive')
else:
    print('x is nonpositive')
```

In the above example, `x>0` is the condition. If the condition is met, the program prints a message that says "x is positive." The colon (":") is needed after all conditional executions in Python. If the condition is not met, the program moves to the `else` branch and prints "x is nonpositive."

In case we have more than two possibilities, we can use `elif` in the middle. Consider the following code:

```
x=5
if x>0:
    print('x is positive')
elif x==0:
    print('x is zero')
else:
    print('x is negative')
```

Python uses the double equal sign ('==') as a comparison operator, to distinguish it from value assignments when we use a single equal sign ('='). The above program has three possibilities, and depending on which condition is met, a different message is printed: "x is positive", "x is zero", or "x is negative".

Sometimes we have more than three possibilities. In such cases, we put *if* in front of the first possibility, *else* in front of the last possibility, and *elif* in front of all other possibilities. The following program *ScoreToGrade.py* is one such example:

```
Score = 88
if score>=90:
    print('grade is A')
elif score>=80:
    print('grade is B')
```

```
elif score>=70:
    print('grade is C')
elif score>=60:
    print('grade is D')
else:
    print('grade is F')
```

The program prints out the letter grade based on the value of the score: A if the score is greater or equal to 90; if not, assign a letter grade of B if the score is above 80, and so on.

Loops

One of the main advantages of computers over human beings is that computers can repeat the same tasks many times at a fast rate. Repeating the same task is called loops or iterations in programming. There are two types of iterations in Python: the `while` statement and the `for` statement.

The `while` Loop

Here is an example of an iteration using the `while` statement. The program is *whileloop.py*:

```
n=0
while n<3: # ①
    n=n+1 # ②
     print(n)# ③
print('finished') #④
```

We first assign a value of 0 to the variable *n*. At ❶, the program starts the `while` loop with the condition `n<3`. As long as the condition is met, the loop keeps running. The colon (":") is needed after the condition. All command lines inside the loop must be indented so that the program knows what lines of command to execute for each iteration. In the above case, lines ❷ and ❸ will be executed for each iteration, but not line ❹ (since it is not indented).

In the first iteration, the value of *n* increases from 0 to 1, and the updated value of *n* is printed out. In the second iteration, the value of *n* increases from 1 to 2, and the updated value of *n*, now at 2, is printed out. In the third iteration, the value of *n* increases to 3, and 3 is printed out. When the program goes to the fourth iteration, the condition `n<3` is no longer met, and the loop stops.

After that, line ❹ is executed. As a result, we see the following output from the program *whileloop.py*:

```
1
2
3
finished
```

The `for` Loop

Another way of iteration in Python is to use the `for` statement. The following program *forloop.py* is one example:

```
for n in range(3): #❶
    n=n+1
    print(n)
print('finished')
```

Line ❶ uses `range()`, a built-in function in Python, and `range(3)` produces values 0, 1, and 2. Python uses 0 indexing and the first value it counts is 0, and the second 1, and so on. Line ❶ tells the program to loop through the three values, and execute the next two lines of code for each value.

The output from the program *forloop.py* produces the same output as the program *whileloop.py* we have seen earlier.

Loops in Loops:

You can put a loop in another loop. Here is one example, and let's name the program *LoopInLoop.py*:

```
for letter in ("A","B","C"): # ❶
    for num in (1,2): # ❷
        print(f"this is {letter}{num}")
```

Line ❶ starts the outer loop, and line ❷ starts the inner loop (the loop inside the outer loop). The program takes the first value in the outer loop, and goes through all iterations in the inner loop and prints a message at each iteration. The program then takes the second value in the outer loop, and repeats the process again. The final output from the program *LoopInLoop.py* is:

```
this is A1
this is A2
this is B1
this is B2
this is C1
this is C2
```

We need to indent twice before all command lines inside the inner loop so that the program which line of command is in which loop.

We are using the string formatting approach inside the `print()` function in the above program. The string `f"this is {letter}{num}"` is meant to place the actual value of the variable *letter* in the place of the first pair of curly brackets, and the value of the variable *num* in the place of the second pair of curly brackets.

NOTE: The string formatting approach `f"this is {letter}{num}"` works only in Python versions 3.6 or newer. If you are using an older version of Python, you should use `"this is {0}{1}".format(letter,num)` instead.

You can even put loops inside a loop that is inside another loop. The logic works the same. The program iterates through all values in the innermost loop for each combination of values in the medium and outer loops. The command lines inside the innermost loop are indented three times.

Differences between loop commands `continue`, `break`, and `pass`

Sometimes you want to treat different iterations inside a loop differently. You can do that with `continue`, `break`, or `pass`, based on what you intend to accomplish.

The Loop Command `continue`

The command `continue` tells Python to stop executing the rest of the commands for the current iteration, and go to the next iteration. For example, the program *forloop1.py* below contains a `continue` command:

```
for n in range(3):
    n=n+1
    if n==2:
        continue
```

```
❶    print(n)
print('finished')
```

when the value of *n* is 2, line ❶ will not be executed because the `continue` command asks the program to skip it and go to the next iteration. The output from the above program is:

```
1
3
finished
```

The Loop Command `break`

The command `break` tells Python to break the loop and skip all remaining iterations. For example, the program *forloop2.py* contains a `break` command:

```
for n in range(3):
    n=n+1
    if n==2:
        break
    print(n)
❶  print('finished')
```

when the value of *n* is 2, the whole loop stops and the program goes to line ❶ directly. The output from *forloop2.py* is:

```
1
finished
```

The Loop Command `pass`

The command `pass` tells Python to do nothing. It is created for situations where a command line is needed but no action needs to be taken. It is used along with `try` and `except` quite often, and we will revisit this command later in this book. For example, the program *forloop3.py* contains a `pass` command:

```
for n in range(3):
    n=n+1
    if n==2:
        pass
    print(n)
print('finished')
```

when the value of *n* is 2, no action needs to be taken. Therefore, the output from the above program is:

```
1
2
3
finished
```

which is the same as the output from the program *forloop.py*.

Summary

In this chapter, you learned how to install Python via Anaconda and Spyder. You learned how to run Python programs on your computer, even if you had no prior experience in programming.

We also discussed the basic operations in Python. You learned the four different types of variables and how to convert one type to another, if possible. Finally, you learned conditional executions and loops.

End of Chapter Exercises

1: What is the output from each of the following lines of command? First write down the answer and then run the command line in Spyder and verify.

print(2)

print(3**2)

print(7//3)

print(7/3)

print(7%3)

print(2+2)

print(10*2)

2: Assume:

Name1 = 'Kentucky '

Name2 = "Wildcats"

What is the output from each of the following lines of command? Verify your answers in Spyder.

```
print(type(Name1))
print(type(Name2))
print(Name1+Name2)
print(Name2+Name1)
print(Name2+' @ '+Name1)
print(3* Name2)
```

3: Assume x=3.458 and y=-2.35, what is the result for each of the following?

```
print(type(x))
print(type(y))
print(round(x,2))
print(round(y,1))
print(round(x,0))
```

4: Here are some examples of integers:

```
a=57
b=-3
c=0
```

What is the outcome from each of the following?

```
print(type(b))
print(str(a))
print(float(c))
```

5: What is the outcome from each of the following line of commands?

```
print(type(5==9))
print('8<7')
print(5==9)
print(type('5==9'))
print(type('8<7'))
print(type('True'))
```

6: What is the outcome from each of the following?

```
print(int(-23.0))
print(int("56"))
print(int(-2.35))
print(str(-23.0))
print(float(8))
```

7: What is the outcome for the following?

```
print(int(True))
print(float(False))
print(str(False))
```

8: What is the outcome from each of the following?

```
print(bool(0))
print(bool(-23))
print(bool(17.6))
print(bool('Python'))
```

9: Are the following valid variable names?

```
global
2pirnt
print2
_squ
list
```

10: What is the output from the following lines of code?

```
for letter in ("A", "B", "C"):
    if letter == "B":
        break
    for num in (1, 2):
        print(f"this is {letter} {num}")
```

11: What is the output from the following lines of code?

```
for letter in ("A", "B", "C"):
    if letter == "B":
        continue
    for num in (1, 2):
        print(f"this is {letter} {num}")
```

12: What is the output from the following lines of code?

```
for letter in ("A", "B", "C"):
    if letter == "B":
        pass
    for num in (1, 2):
        print(f"this is {letter}{num}")
```

13: What is the output from the following lines of code?

```
for letter in ("A","B"):
    for num in (1,2):
        print(f"this is {letter}{num}")
```