An-Najah National University
Faculty of Engineering and IT
Energy Engineering & Environment Dept.
Modeling & Simulating Energy Systems
(10656312)

Dr. Mohammed F. Alsayed

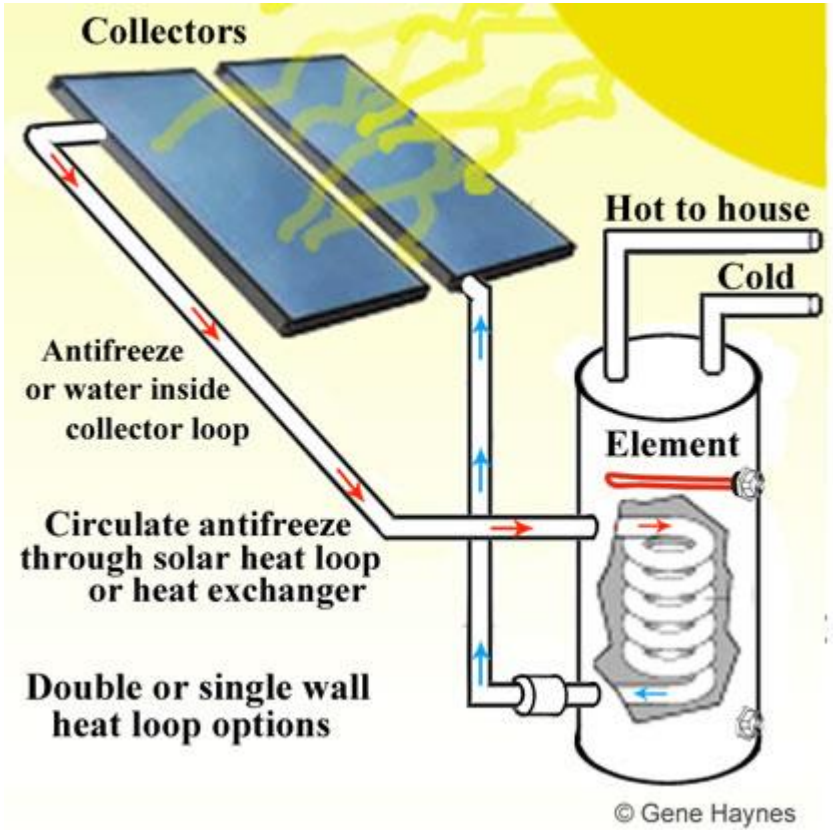
Introduction

- What is energy modeling and simulation?
- Why energy modeling and simulation?
- How to model and simulate energy systems?
- How to become good in energy modeling and simulation?

Lets start with very simple examples

Ex. 1: Residential solar water heater

- What equations and data do we need to model and simulate the system?
- Output required!!



Ex. 1: Residential solar water heater

# of days	month	MJ/m2	Ta	Tm (tap)	L (MJ)
31	January	12.13	7.7	12.7	61.29
28	February	15.12	8.2	13.2	54.77
31	March	19.08	10.4	15.4	57.79
30	April	24.52	15.1	22	47.65
31	May	28.19	19.1	22	49.24
30	June	30.85	21.4	23.5	45.77
31	July	30.24	23.1	18.1	54.29
31	August	28.26	23.1	18.1	54.29
30	September	24.23	21.8	16.8	54.17
31	October	19.01	19.1	23.1	47.82
30	November	13.46	14.1	19.1	51.29
31	December	10.98	9.7	14.7	58.70

Ex. 1: Residential solar water heater

- Conclusions:
- Energy balance & mass balance are essential.
- Data visualization is an art.
- Accuracy is not cheap.

Ex. 2: Building simulation using Energy 3D software

- Download it free from:
- Energy3D (free) download Windows version (freedownloadmanager.org)

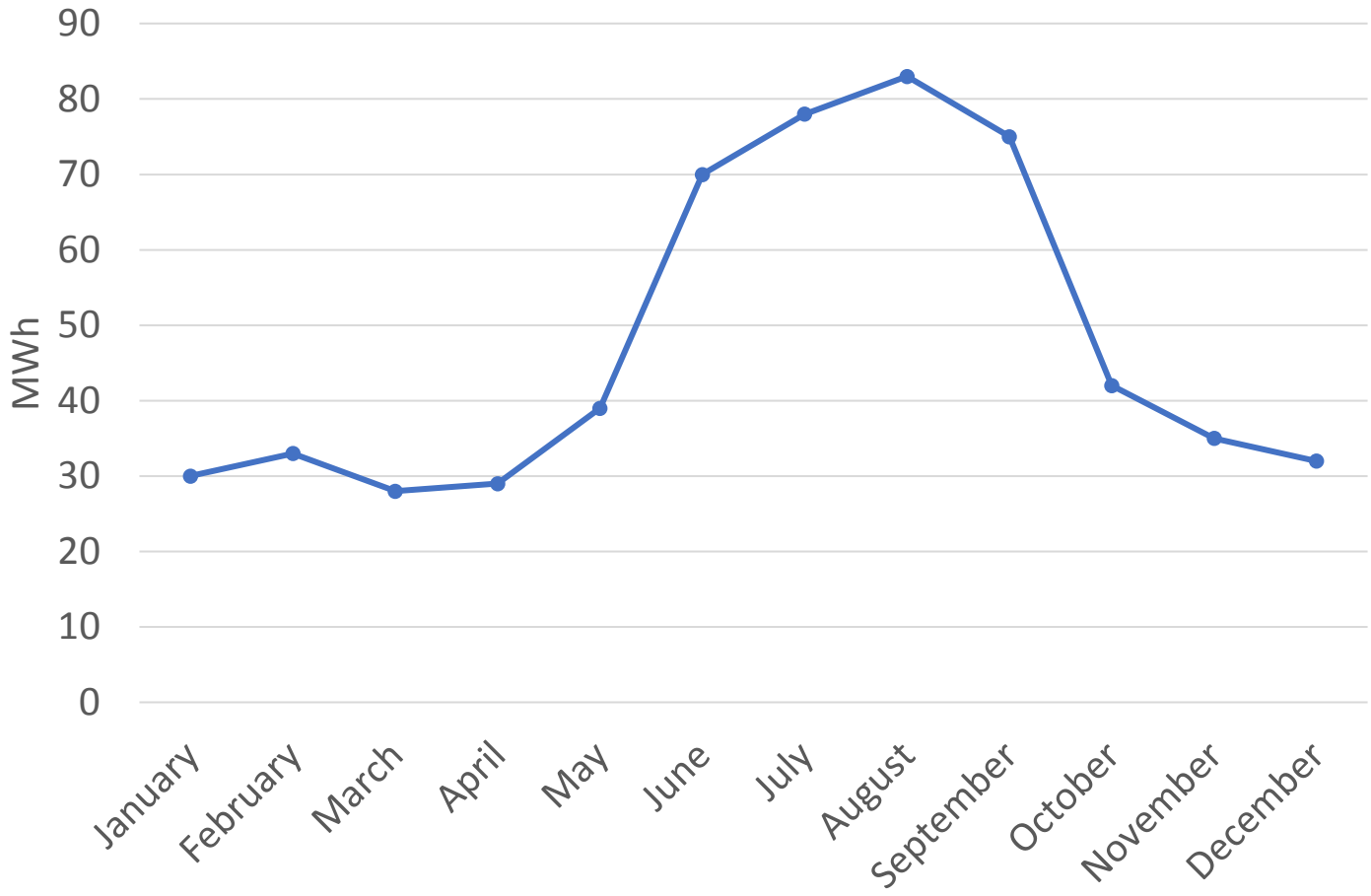
- What do think the main differences between the two approaches?
- Which one you liked more?

Ex. 3: Building AC simulation

- Use linear regression to develop a model.

$$a = \frac{[(\sum y)(\sum x^2) - (\sum x)(\sum xy)]}{[n(\sum x^2) - (\sum x)^2]}$$

$$b = \frac{[n(\sum xy) - (\sum x)(\sum y)]}{[n(\sum x^2) - (\sum x^2)]}$$

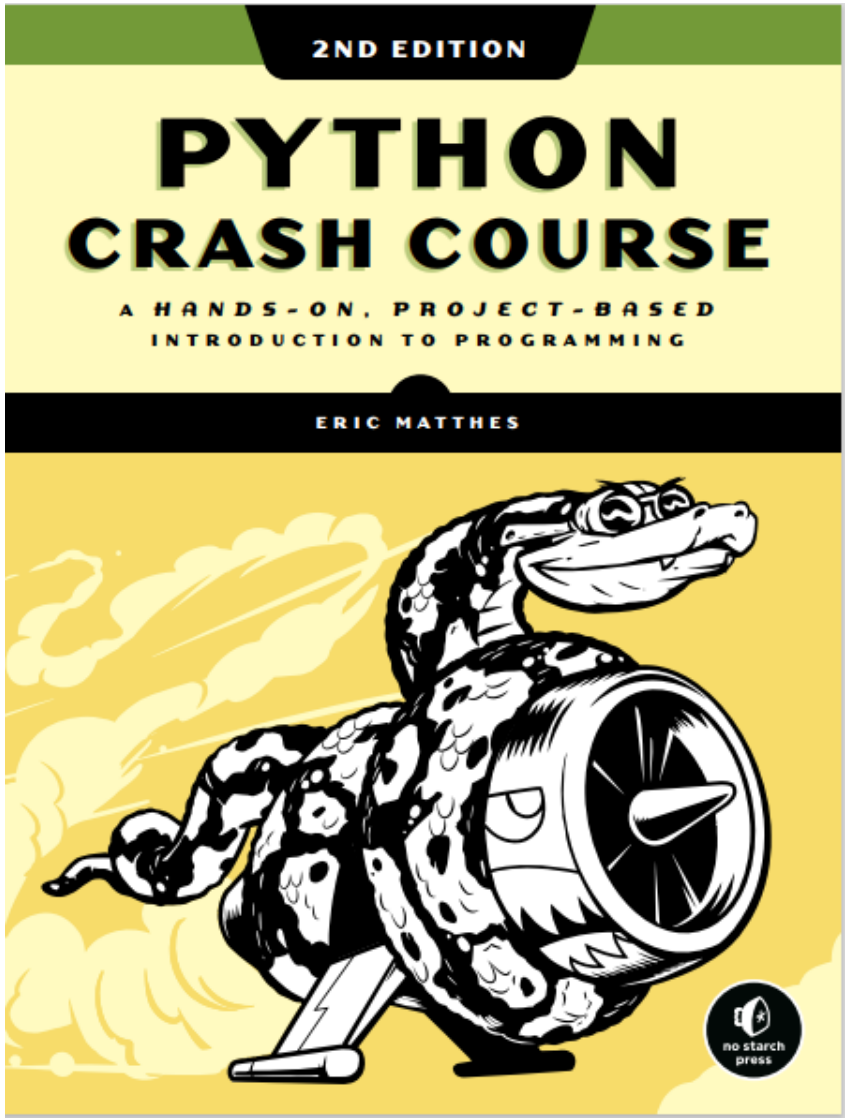


Ex. 4: Population versus energy consumption

- Use the PCBS website to find population and energy consumption historical data. Then, develop suitable modeling equation.


python™

Textbook & Outlines



Textbook and outlines

- **Online Resources**
  - You can find all the supplementary resources for the book online at
  - https://nostarch.com/pythoncrashcourse2e/ or http://ehmatthes.github.io/pcc_2e/.
  - These resources include:
    - Setup instructions These instructions are identical to what's in the book but include active links you can click for all the different pieces.
    - Updates Python, like all languages, is constantly evolving.
    - Solutions to exercises You should spend significant time on your own attempting the exercises in the "Try It Yourself" sections. But if you're stuck and can't make any progress, solutions to most of the exercises are online.
    - Cheat sheets A full set of downloadable cheat sheets for a quick reference to major concepts is also online.

Why Python?

- Python is an incredibly efficient language: your programs will do more in fewer lines of code.
- Python's syntax will also help you write "clean" code. Your code will be easy to read, easy to debug, and easy to extend and build upon compared to other languages.
- Python is also used heavily in scientific fields for academic research and applied work.
- Python community includes an incredibly diverse and welcoming group of people.

Chapter 1: Getting Started

Python Versions

- As of this writing, the latest version is Python 3.7, but everything in this book should run on Python 3.6 or later.

- Appendix A contains a comprehensive guide to installing the latest version of Python on each major operating system as well.

PYTHON installation

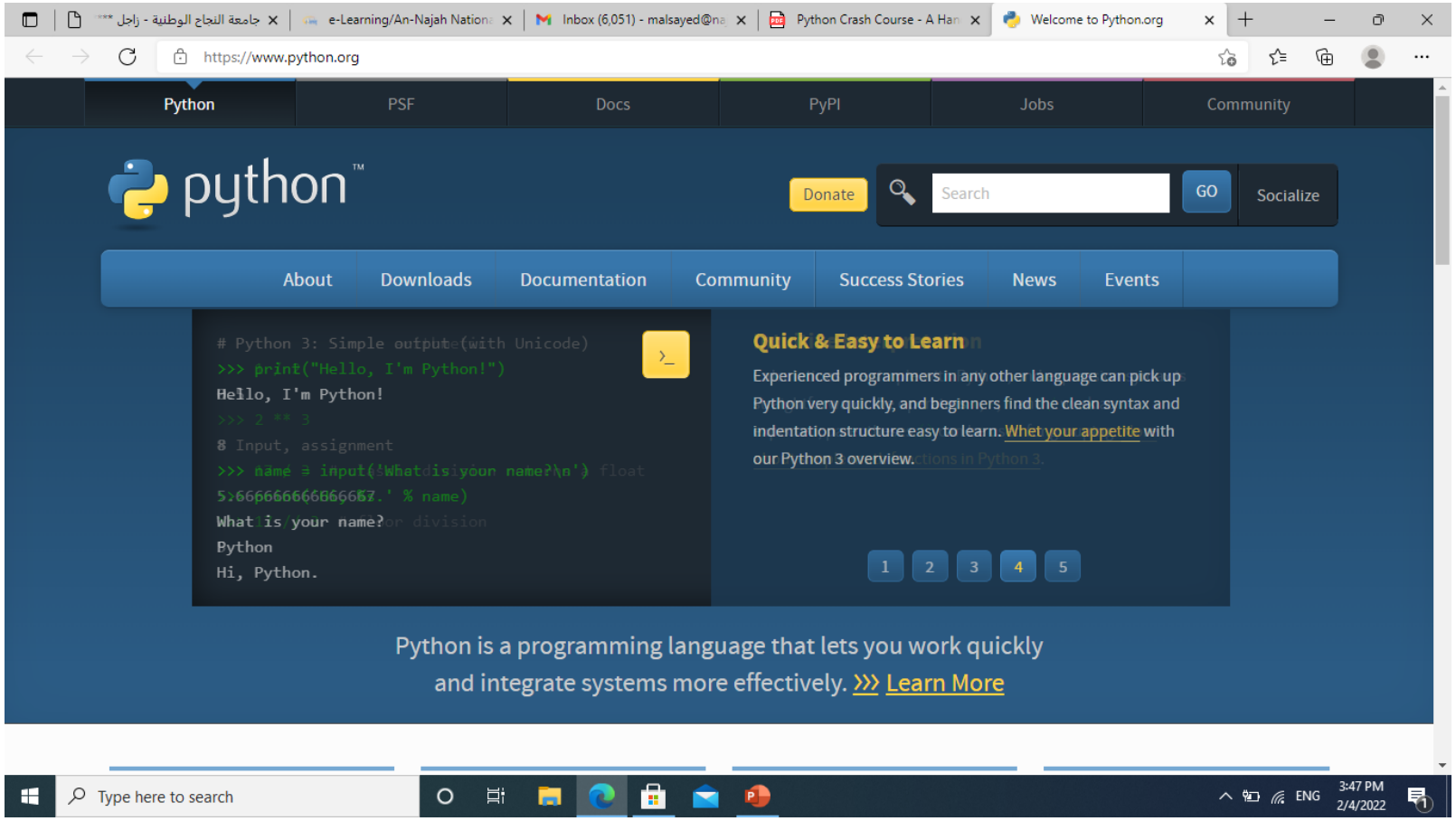
- Python on windows
- Installing Python
  - First, check whether Python is installed on your system. Open a command window by entering command into the Start menu or by holding down the shift key while right-clicking on your desktop and selecting Open command window here from the menu.
  - In the terminal window, enter python in lowercase. If you get a Python prompt (>>>) in response, Python is installed on your system. If you see an error message telling you that python is not a recognized command, Python isn't installed.

PYTHON installation

- In that case, or if you see a version of Python earlier than Python 3.6, you need to download a Python installer for Windows.
- Go to https://python.org/
- Hover over the Downloads link. You should see a button for downloading the latest version of Python. Click the button, which should automatically start downloading the correct installer for your system.

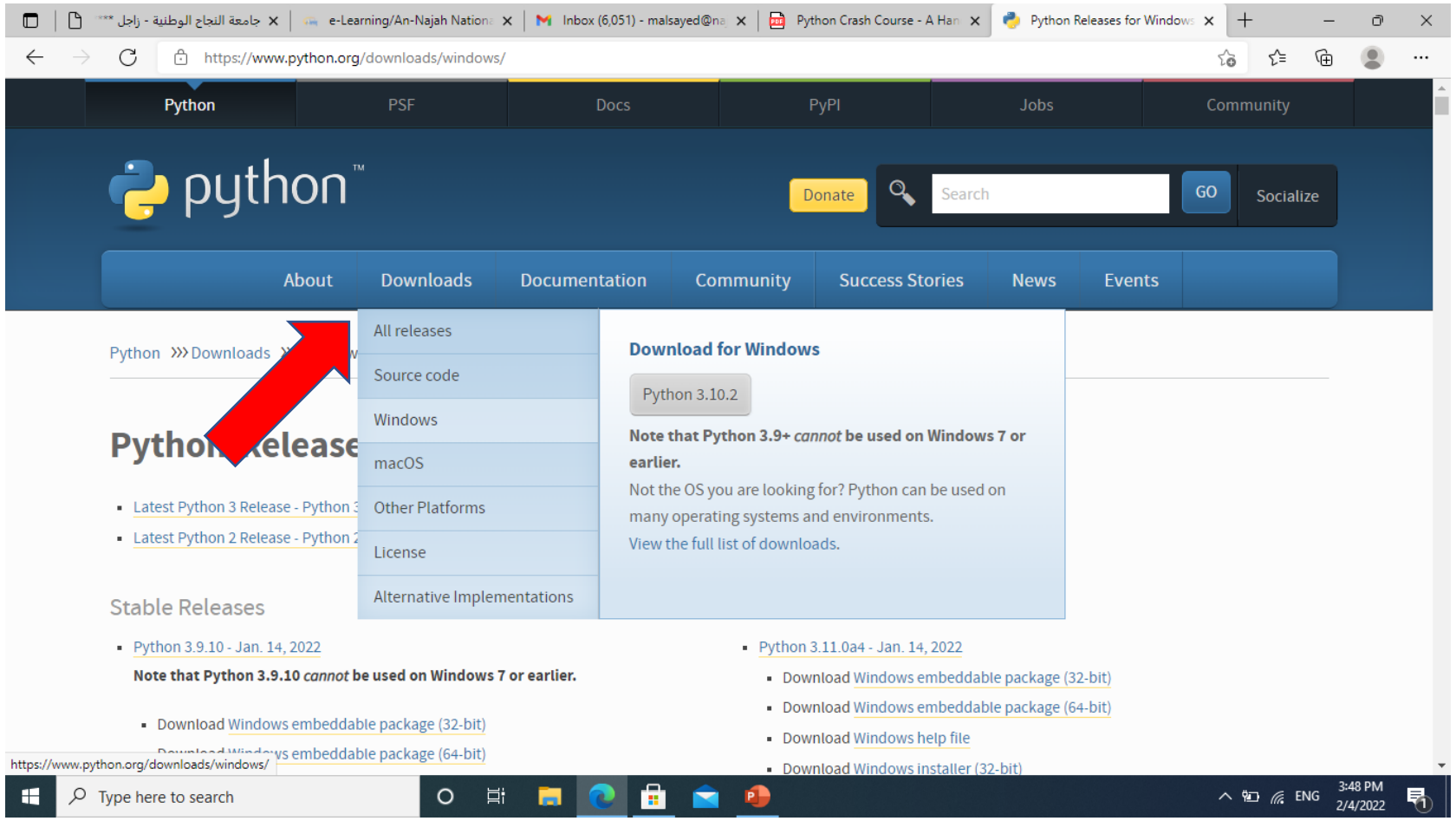
PYTHON installation

- When you click the link, you will go to:



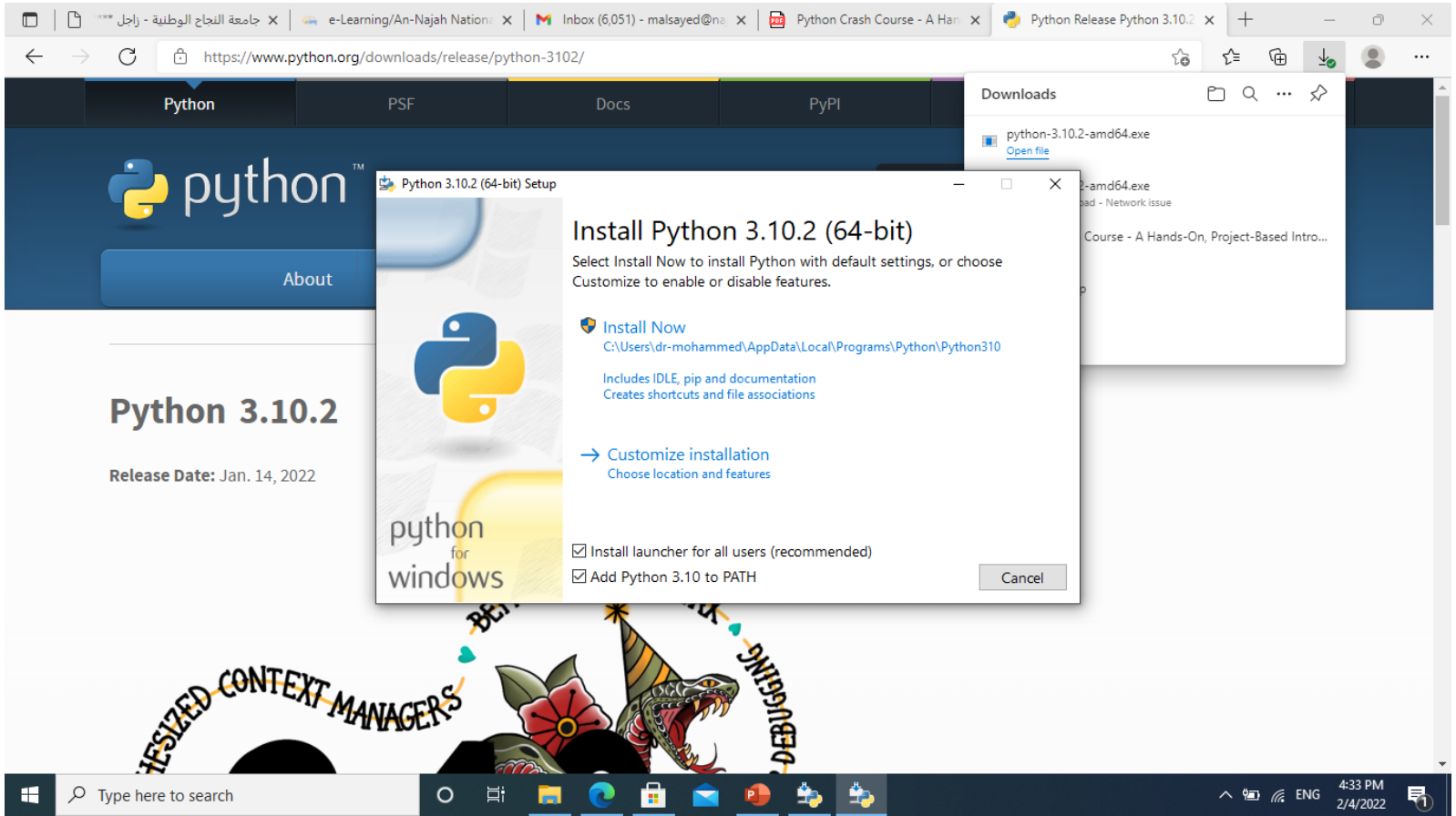
PYTHON installation

- Go to downloads and select ‘’Windows’’



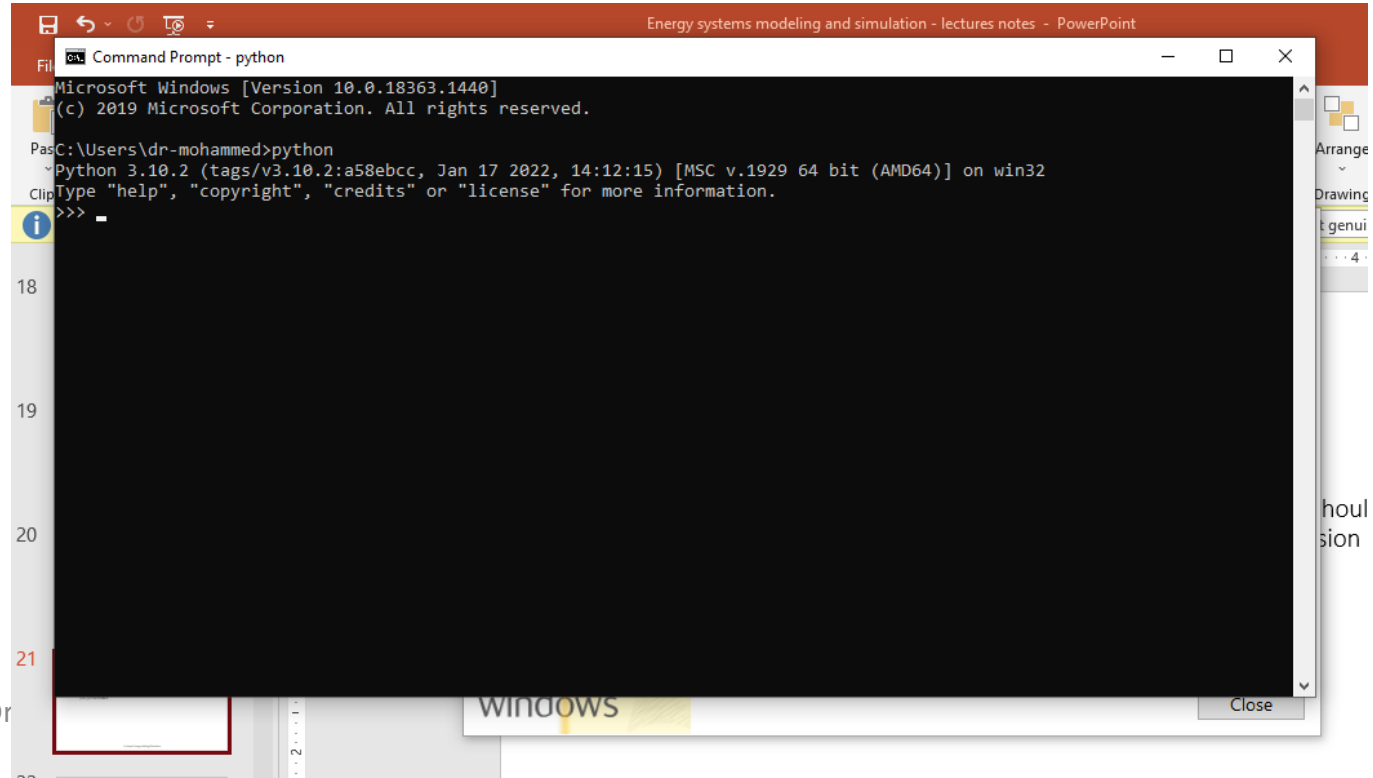
PYTHON installation

- After you've downloaded the file, run the installer. **Make sure** you select the option **Add Python to PATH**, which will make it easier to configure your system correctly.



PYTHON installation

- Running Python in a Terminal Session
  - Open a command window and enter python in lowercase. You should see a Python prompt (>>>), which means Windows has found the version of Python you just installed.



PYTHON installation

- Enter the following line in your Python session, and make sure you see the output Hello Python interpreter!

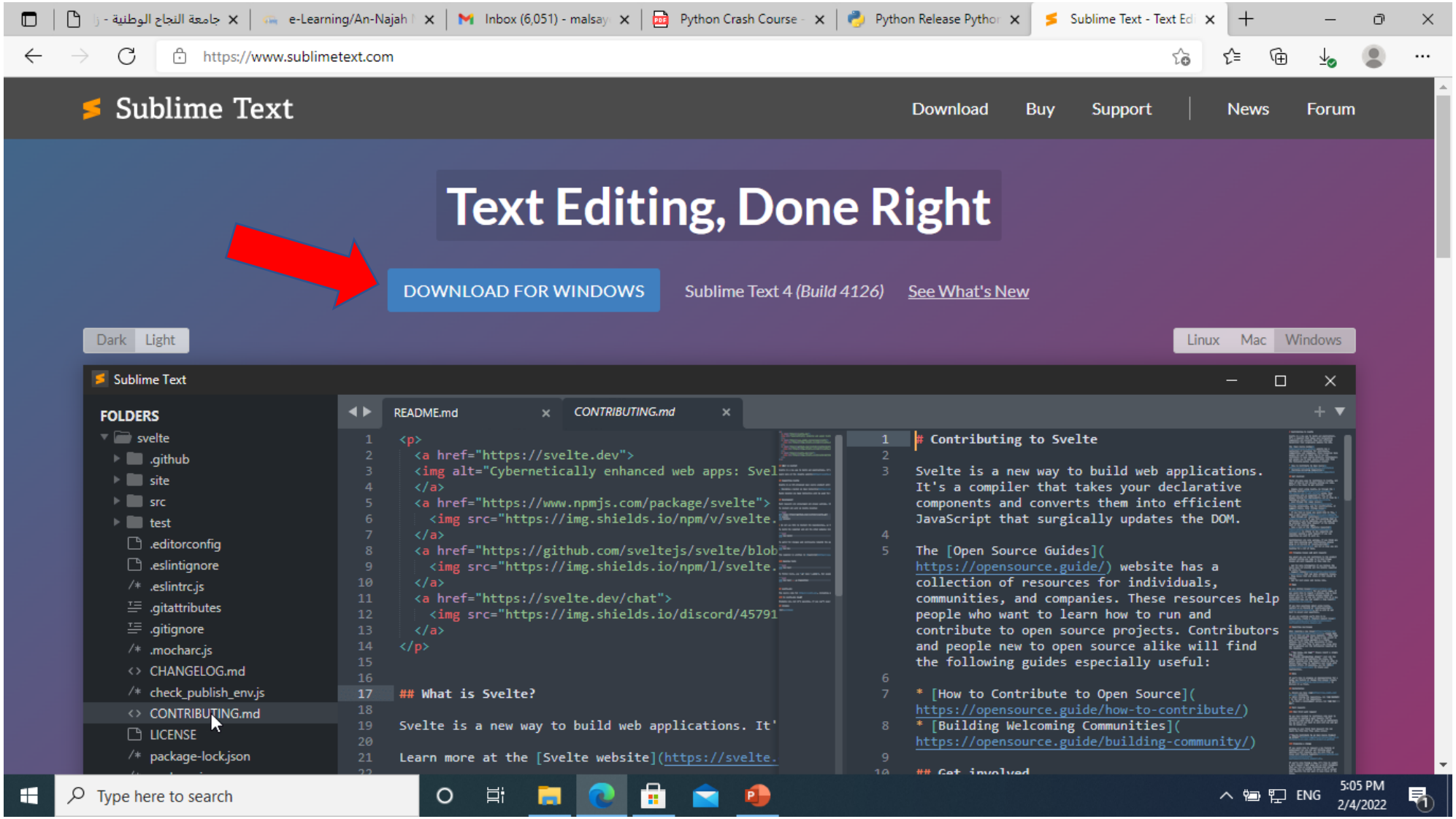
```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

- Any time you want to run a snippet of Python code, open a command window and start a Python terminal session. To close the terminal session, press ctrl-Z and then press enter, or enter the command exit().

PYTHON installation

- Installing Sublime Text
- You can download an installer for Sublime Text at:
- https://sublimetext.com/
- Click the download link and look for a Windows installer. After downloading the installer, run the installer and accept all of its defaults.

PYTHON installation

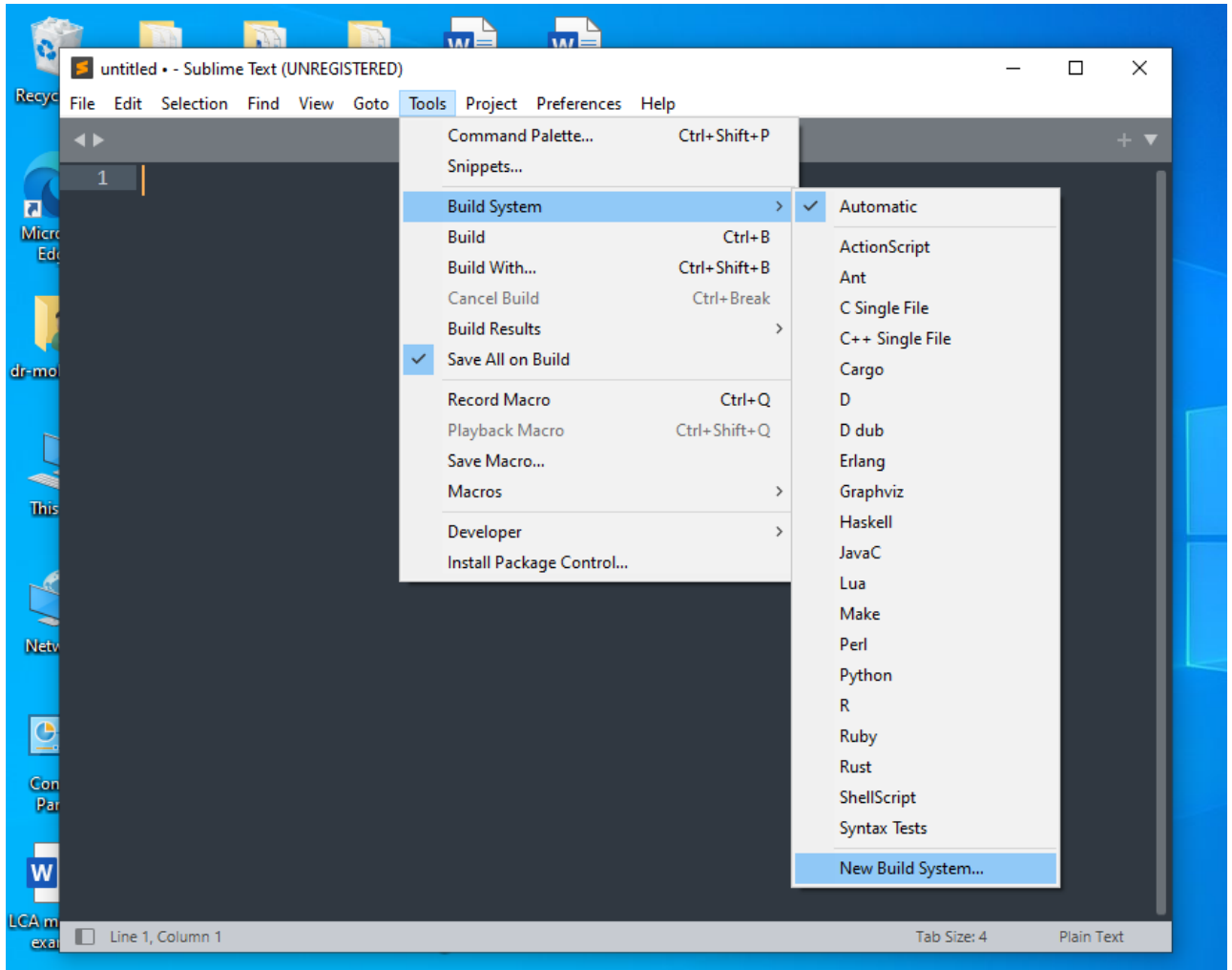


Running a Hello World Program

- Configuring Sublime Text to Use the Correct Python Version
  - If the python command on your system runs Python 3, you won't need to configure anything and can skip to the next section. If you use the python3 command, you'll need to configure Sublime Text to use the correct Python version when it runs your programs.
  - Click the Sublime Text icon to launch it, or search for Sublime Text in your system's search bar and then launch it. Go to **Tools → Build System → New Build System**, which will open a new configuration file for you. Delete what you see and enter the following:

```
{
    "cmd": ["python3", "-u", "$file"],
}
```

Running a Hello World Program



Running a Hello World Program

- For easy copy and paste:
  - "cmd": ["python3", "-u", "$file"],

- This code tells Sublime Text to use your system's python3 command when running your Python program files. Save the file as **Python3.sublime-build** in the default directory that Sublime Text opens when you choose Save.

Running a Hello World Program

- Running hello_world.py
  - Before you write your first program, make a folder called ***python_work*** somewhere on your system for your projects.
  - It's best to use lowercase letters and underscores for spaces in file and folder names, because Python uses these naming conventions.
  - Open Sublime Text, and save an empty Python file (**File → Save As**) called hello_world.py in your python_work folder.
  - The extension .py tells Sublime Text that the code in your file is written in Python, which tells it how to run the program and highlight the text in a helpful way. After you've saved your file, enter the following line in the text editor:

```
print("Hello Python world!")
```

Running a Hello World Program

- If the python command works on your system, you can run your program by selecting **Tools → Build** in the menu or by pressing **ctrl-B**.
- If you had to configure Sublime Text in the previous section, select **Tools → Build System** and then select **Python**.
- From now on you'll be able to select **Tools → Build** or just press **ctrl-B** to run your programs.

```
Hello Python world!
[Finished in 0.1s]
```

Troubleshooting

- When a program contains a significant error, Python displays a traceback, which is an error report.
- Step away from your computer, take a short break, and then try again.
- Start over again.
- Ask someone else to follow the steps in this chapter, on your computer or a different one, and watch what they do carefully.
- Find someone who knows Python and ask them to help you get set up.
- The setup instructions in this chapter are also available through the book's companion website at https://nostarch.com/pythoncrashcourse2e/.
- Ask for help online. **Appendix C** provides a number of resources, such as forums and live chat sites, where you can ask for solutions from people who've already worked through the issue you're currently facing.

Running Python Programs from a Terminal

- Most of the programs you write in your text editor you'll run directly from the editor.
- You can do this on any system with Python installed if you know how to access the directory where the program file is stored.
- To try this, make sure you've saved the hello_world.py file in the python_work folder on your desktop.

Running Python Programs from a Terminal

- On Windows
  - You can use the terminal command cd, for change directory, to navigate through your filesystem in a command window. The command dir, for directory, shows you all the files that exist in the current directory. Open a new terminal window and enter the following commands to run hello_world.py:

```
❶ C:\> cd Desktop\python_work
❷ C:\Desktop\python_work> dir
  hello_world.py
❸ C:\Desktop\python_work> python hello_world.py
  Hello Python world!
```

Try It Yourself

- **1-1. python.org**: Explore the Python home page (https://python.org/) to find topics that interest you.
- **1-2. Hello World Typos**: Open the hello_world.py file you just created. Make a typo somewhere in the line and run the program again. Can you make a typo that generates an error? Can you make sense of the error message? Can you make a typo that doesn't generate an error? Why do you think it didn't make an error?
- **1-3. Infinite Skills**: If you had infinite programming skills, what would you build? You're about to learn how to program. If you have an end goal in mind, you'll have an immediate use for your new skills; now is a great time to draft descriptions of what you want to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. Take a few minutes now to describe three programs you want to create.

Chapter 2: Variables and Simple Data Types

Variables

- Let’s try using a variable in hello_world.py. Add a new line at the beginning of the file, and modify the second line:

```
message = "Hello Python world!"
print(message)
```

- Run this program to see what happens. You should see the same output you saw previously:

```
Hello Python world!
```

Variables

- Let’s expand on this program by modifying hello_world.py to print a second message. Add a blank line to hello_world.py, and then add two new lines of code:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

- Now when you run hello_world.py, you should see two lines of output:

```
Hello Python world!
Hello Python Crash Course world!
```

Naming and Using Variables

- Be sure to keep the following variable rules in mind:
  - Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable message_1 but not 1_message.
  - Spaces are not allowed in variable names, but underscores can be used to separate words in variable names.
  - Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word print. (See “Python Keywords and Built-in Functions” on page 471.)
  - Variable names should be short but descriptive. For example, name is better than n, student_name is better than s_n, and name_length is better than length_of_persons_name.
  - Be careful when using the lowercase letter l and the uppercase letter O because they could be confused with the numbers 1 and 0.

Naming and Using Variables

- Note:
- The Python variables you're using at this point should be lowercase. You won't get errors if you use uppercase letters, but uppercase letters in variable names have special meanings that we'll discuss in later chapters

Avoiding Name Errors When Using Variables

- We'll write some code that generates an error on purpose.

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

- The Python interpreter doesn't spellcheck your code, but it does ensure that variable names are spelled consistently

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

Variables Are Labels

- Variables are often described as boxes you can store values in. This idea can be helpful the first few times you use a variable, but it isn't an accurate way to describe how variables are represented internally in Python.
- It's much better to think of variables as labels that you can assign to values. You can also say that a variable references a certain value.

Good Advice

- The best way to understand new programming concepts is to try using them in your programs. If you get stuck while working on an exercise in this book, try doing something else for a while. If you're still stuck, review the relevant part of that chapter.
- If you still need help, see the suggestions in Appendix C.

Try It Yourself

- Write a separate program to accomplish each of these exercises. Save each program with a filename that follows standard Python conventions, using lowercase letters and underscores, such as simple_message.py and simple_messages.py.
- **2-1. Simple Message**: Assign a message to a variable, and then print that message.
- **2-2. Simple Messages**: Assign a message to a variable, and print that message. Then change the value of the variable to a new message, and print the new message.

Strings

- A string is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
'This is also a string.'
```

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Changing Case in a String with Methods

- Try

```
name = "ada lovelace"
print(name.title())
```

- A method is an action that Python can perform on a piece of data. The dot (.) after name in name.title() tells Python to make the title() method act on the variable name.
- The title() method changes each word to title case, where each word begins with a capital letter. This is useful because you'll often want to think of a name as a piece of information. For example, you might want your program to recognize the input values Ada, ADA, and ada as the same name, and display all of them as Ada.

Changing Case in a String with Methods

- Try also

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

- The lower() method is particularly useful for storing data. Many times you won't want to trust the capitalization that your users provide, so you'll convert strings to lowercase before storing them. Then when you want to display the information, you'll use the case that makes the most sense for each string.

Using Variables in Strings

- In some situations, you'll want to use a variable's value inside a string.

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
print(full_name)
```

- These strings are called ***f-strings***. The ***f*** is for format, because Python formats the string by replacing the name of any variable in braces with its value.

Using Variables in Strings

- Try

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
print(f"Hello, {full_name.title()}!")
```

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
message = f"Hello, {full_name.title()}!"
print(message)
```

Adding Whitespace to Strings with Tabs or Newlines

- To add a tab to your text, use the character combination \t:

```
>>> print("Python")
Python
>>> print("\tPython")
    Python
```

- To add a newline in a string, use the character combination \n:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

Adding Whitespace to Strings with Tabs or Newlines

- You can combine it together

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Stripping Whitespace

- Extra whitespace can be confusing in your programs. To programmers **'python'** and **'python '** look pretty much the same. But to a program, they are two different strings.
- Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right end of a string, use the rstrip() method.

```
>>> favorite_language = 'python '
>>> favorite_language
'python '
>>> favorite_language.rstrip()
'python'
>>> favorite_language
'python '
```

Stripping Whitespace

- When you ask Python for this value in a terminal session, you can see the space at the end of the value. When the rstrip() method acts on the variable favorite_language, this extra space is removed. However, it is only removed **temporarily**.
- To remove the whitespace from the string **permanently**, you have to associate the stripped value with the variable name:

```
>>> favorite_language = 'python '
>>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

Stripping Whitespace

- You can also strip whitespace from the left side of a string using the **lstrip()** method, or from both sides at once using **strip()**:

```
>>> favorite_language = ' python '
>>> favorite_language.rstrip()
' python'
>>> favorite_language.lstrip()
'python '
>>> favorite_language.strip()
'python'
```

- In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

Avoiding Syntax Errors with Strings

- A syntax error occurs when Python doesn't recognize a section of your program as valid Python code.
- For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.

Avoiding Syntax Errors with Strings

- Develop the following program and call it ***apostrophe.py***

```
message = "One of Python's strengths is its diverse community."
print(message)
```

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

Try It Yourself

- Save each of the following exercises as a separate file with a name like name_cases.py. If you get stuck, take a break or see the suggestions in Appendix C.
- **2-3. Personal Message**: Use a variable to represent a person's name, and print a message to that person. Your message should be simple, such as, "Hello Eric, would you like to learn some Python today?"
- **2-4. Name Cases:** Use a variable to represent a person's name, and then print that person's name in lowercase, uppercase, and title case.

Try It Yourself

- **2-5. Famous Quote**: Find a quote from a famous person you admire. Print the quote and the name of its author. Your output should look something like the following, including the quotation marks:
  - Albert Einstein once said, “A person who never made a mistake never tried anything new.”
- **2-6. Famous Quote 2**: Repeat Exercise 2-5, but this time, represent the famous person’s name using a variable called famous_person. Then compose your message and represent it with a new variable called message. Print your message.

Try It Yourself

- **2-7. Stripping Names:** Use a variable to represent a person's name, and include some whitespace characters at the beginning and end of the name. Make sure you use each character combination, "\t" and "\n", at least once. Print the name once, so the whitespace around the name is displayed. Then print the name using each of the three stripping functions, lstrip(), rstrip(), and strip().

Numbers

- Let’s first look at how Python manages integers, because they’re the simplest to work with.
- You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Numbers

- Python uses two multiplication symbols to represent exponents:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

Numbers

- Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations.

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

- The spacing in these examples has no effect on how Python evaluates the expressions.

Floats

- Python calls any number with a decimal point a float.

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

Integers and Floats

- When you divide any two numbers, even if they are integers that result in a whole number, you'll always get a float.
- If you mix an integer and a float in any other operation, you'll get a float as well.

```
>>> 4/2
2.0
```

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

Underscores in Numbers

- When you're writing long numbers, you can group digits using underscores to make large numbers more readable:

```
>>> universe_age = 14_000_000_000
```

Multiple Assignment

- You can assign values to more than one variable using just a single line.

```
>>> x, y, z = 0, 0, 0
```

- You need to separate the variable names with commas, and do the same with the values, and Python will assign each value to its respectively positioned variable. As long as the number of values matches the number of variables, Python will match them up correctly.

Constants

- A constant is like a variable whose value stays the same throughout the life of a program.
- Python doesn't have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed.
- When you want to treat a variable as a constant in your code, make the name of the variable all capital letters.

```
MAX_CONNECTIONS = 5000
```

Try It Yourself

- **2-8. Number Eight**: Write addition, subtraction, multiplication, and division operations that each result in the number 8. Be sure to enclose your operations in print() calls to see the results. You should create four lines that look like this:
  - print(5+3)
  - Your output should simply be four lines with the number 8 appearing once on each line.
- **2-9. Favorite Number**: Use a variable to represent your favorite number. Then, using that variable, create a message that reveals your favorite number. Print that message.

Comments

- A comment allows you to write notes in English within your programs.
- In Python, the hash mark (**#**) indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter.

```
# Say hello to everyone.
print("Hello Python people!")
```

Try It Yourself

- **2-10. Adding Comments:** Choose two of the programs you've written, and add at least one comment to each. If you don't have anything specific to write because your programs are too simple at this point, just add your name and the current date at the top of each program file. Then write one sentence describing what the program does.

The Zen of Python

- Python community's philosophy is contained in "The Zen of Python" by Tim Peters.

TRY IT YOURSELF

2-11. Zen of Python: Enter `import this` into a Python terminal session and skim through the additional principles.

Chapter 3: Introducing Lists

What Is a List?

- A list is a collection of items in a particular order.
- You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family.

- In Python, square brackets ([]) indicate a list, and individual elements in the list are separated by commas.

What Is a List?

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

- Because this isn’t the output you want your users to see, let’s learn how to access the individual items in a list.

Accessing Elements in a List

- You can access any element in a list by telling Python the position, or index.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
```

```
trek
```

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

Index Positions Start at 0, Not 1

- Python considers the first item in a list to be at position 0, not position 1.
- The second item in a list has an index of 1. Using this counting system, you can get any element you want from a list by subtracting one from its position in the list.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

```
cannondale
specialized
```

Index Positions Start at 0, Not 1

- Python has a special syntax for accessing the last element in a list. By asking for the item at index -1, Python always returns the last item in the list:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

- This convention extends to other negative index values as well. The index -2 returns the second item from the end of the list, the index -3 returns the third item from the end, and so forth.

Using Individual Values from a List

- You can use individual values from a list just as you would any other variable.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
message = f"My first bicycle was a {bicycles[0].title()}."

print(message)
```

Try It Yourself

- Try these short programs to get some firsthand experience with Python's lists. You might want to create a new folder for each chapter's exercises to keep them organized.
- **3-1. Names**: Store the names of a few of your friends in a list called names. Print each person's name by accessing each element in the list, one at a time.
- **3-2. Greetings:** Start with the list you used in Exercise 3-1, but instead of just printing each person's name, print a message to them. The text of each message should be the same, but each message should be personalized with the person's name.
- **3-3. Your Own List:** Think of your favorite mode of transportation, such as a motorcycle or a car, and make a list that stores several examples. Use your list to print a series of statements about these items, such as "I would like to own a Honda motorcycle."

Changing, Adding, and Removing Elements

- **Modifying Elements in a List**
  - The syntax for modifying an element is similar to the syntax for accessing an element in a list.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles[0] = 'ducati'
print(motorcycles)
```

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

Changing, Adding, and Removing Elements

- **Adding Elements to a List**
  - You might want to add a new element to a list for many reasons.
  - **Appending Elements to the End of a List**
    - The simplest way to add a new element to a list is to append the item to the list. When you append an item to a list, the new element is added to the end of the list.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles.append('ducati')
print(motorcycles)
```

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

Changing, Adding, and Removing Elements

- The append() method makes it easy to build lists dynamically.
- For example, you can start with an empty list and then add items to the list using a series of append() calls. Using an empty list, let's add the elements 'honda', 'yamaha', and 'suzuki' to the list:

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

```
['honda', 'yamaha', 'suzuki']
```

Changing, Adding, and Removing Elements

- **Inserting Elements into a List**
  - You can add a new element at any position in your list by using the insert() method.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

motorcycles.insert(0, 'ducati')
print(motorcycles)
```

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Changing, Adding, and Removing Elements

- **Removing Elements from a List**
  - Removing an Item Using the del Statement
    - If you know the position of the item you want to remove from a list, you can use the del statement.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[0]
print(motorcycles)
```

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

Changing, Adding, and Removing Elements

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```



```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

Changing, Adding, and Removing Elements

- **Removing an Item Using the pop() Method**
  - For example, you might want to get the x and y position of an alien that was just shot down, so you can draw an explosion at that position.
  - In a web application, you might want to remove a user from a list of active members and then add that user to a list of inactive members.
- The pop() method removes the last item in a list, but it lets you work with that item after removing it.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

popped_motorcycle = motorcycles.pop()
print(motorcycles)
print(popped_motorcycle)
```

Changing, Adding, and Removing Elements

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

```
The last motorcycle I owned was a Suzuki.
```

Changing, Adding, and Removing Elements

- **Popping Items from any Position in a List**
  - You can use pop() to remove an item from any position in a list by including the index of the item you want to remove in parentheses.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

first_owned = motorcycles.pop(0)
print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

```
The first motorcycle I owned was a Honda.
```

Changing, Adding, and Removing Elements

- **Remember** that each time you use pop(), the item you work with is no longer stored in the list.
- If you're unsure whether to use the del statement or the pop() method, **here's a simple way to decide**: when you want to delete an item from a list and not use that item in any way, use the del statement; if you want to use an item as you remove it, use the pop() method.

Changing, Adding, and Removing Elements

- **Removing an Item by Value**
  - Sometimes you won't know the position of the value you want to remove from a list. If you only know the value of the item you want to remove, you can use the remove() method.

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

motorcycles.remove('ducati')
print(motorcycles)
```

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

Changing, Adding, and Removing Elements

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

too_expensive = 'ducati'
motorcycles.remove(too_expensive)
print(motorcycles)
print(f"\nA {too_expensive.title()} is too expensive for me.")
```

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']

A Ducati is too expensive for me.
```

Changing, Adding, and Removing Elements

- **Note**: the remove() method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to make sure all occurrences of the value are removed. You'll learn how to do this in Chapter 7.

Try It Yourself

- The following exercises are a bit more complex than those in Chapter 2, but they give you an opportunity to use lists in all of the ways described.
- **3-4. Guest List:** If you could invite anyone, living or deceased, to dinner, who would you invite? Make a list that includes at least three people you'd like to invite to dinner. Then use your list to print a message to each person, inviting them to dinner.

Try It Yourself

- **3-5. Changing Guest List:** You just heard that one of your guests can't make the dinner, so you need to send out a new set of invitations. You'll have to think of someone else to invite.
  - Start with your program from Exercise 3-4. Add a print() call at the end of your program stating the name of the guest who can't make it.
  - Modify your list, replacing the name of the guest who can't make it with the name of the new person you are inviting.
  - Print a second set of invitation messages, one for each person who is still in your list.

Try It Yourself

- **3-6. More Guests**: You just found a bigger dinner table, so now more space is available. Think of three more guests to invite to dinner.
  - Start with your program from Exercise 3-4 or Exercise 3-5. Add a print() call to the end of your program informing people that you found a bigger dinner table.
  - Use insert() to add one new guest to the beginning of your list.
  - Use insert() to add one new guest to the middle of your list.
  - Use append() to add one new guest to the end of your list.
  - Print a new set of invitation messages, one for each person in your list.

Try It Yourself

- **3-7. Shrinking Guest List**: You just found out that your new dinner table won't arrive in time for the dinner, and you have space for only two guests.
- Start with your program from Exercise 3-6. Add a new line that prints a message saying that you can invite only two people for dinner.
- Use pop() to remove guests from your list one at a time until only two names remain in your list. Each time you pop a name from your list, print a message to that person letting them know you're sorry you can't invite them to dinner.
- Print a message to each of the two people still on your list, letting them know they're still invited.
- Use del to remove the last two names from your list, so you have an empty list. Print your list to make sure you actually have an empty list at the end of your program.

Organizing a List

- Often, your lists will be created in an unpredictable order, because you can’t always control the order in which your users provide their data.
- Although this is unavoidable in most circumstances, you’ll frequently want to present your information in a particular order.
- Sometimes you’ll want to preserve the original order of your list, and other times you’ll want to change the original order.

Sorting a List Permanently with the sort() Method

- Imagine we have a list of cars and want to change the order of the list to store them alphabetically.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort()
print(cars)
```

```
['audi', 'bmw', 'subaru', 'toyota']
```

Sorting a List Permanently with the sort() Method

- You can also sort this list in reverse alphabetical order by passing the argument reverse=True to the sort() method.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

```
['toyota', 'subaru', 'bmw', 'audi']
```

Sorting a List Temporarily with the sorted() Function

- To maintain the original order of a list but present it in a sorted order, you can use the sorted() function. The sorted() function lets you display your list in a particular order but doesn't affect the actual order of the list.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

print("Here is the original list:")
print(cars)

print("\nHere is the sorted list:")
print(sorted(cars))

print("\nHere is the original list again:")
print(cars)
```

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']

Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']

Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']
```

Sorting a List Temporarily with the sorted() Function

- The sorted() function can also accept a reverse=True argument if you want to display a list in reverse alphabetical order.

- Note: Sorting a list alphabetically is a bit more complicated when all the values are not in lowercase. There are several ways to interpret capital letters when determining a sort order, and specifying the exact order can be more complex than we want to deal with at this time. However, most approaches to sorting will build directly on what you learned in this section.

Printing a List in Reverse Order

- To reverse the original order of a list, you can use the reverse() method.
- The reverse() method changes the order of a list permanently, but you can revert to the original order anytime by applying reverse() to the same list a second time.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

Finding the Length of a List

- You can quickly find the length of a list by using the len() function.

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

Try It Yourself

- **3-8. Seeing the World:** Think of at least five places in the world you'd like to visit.
- Store the locations in a list. Make sure the list is not in alphabetical order.
- Print your list in its original order. Don't worry about printing the list neatly, just print it as a raw Python list.
- Use sorted() to print your list in alphabetical order without modifying the actual list.
- Show that your list is still in its original order by printing it.
- Use sorted() to print your list in reverse alphabetical order without changing the order of the original list.
- Show that your list is still in its original order by printing it again.
- Use reverse() to change the order of your list. Print the list to show that its order has changed.
- Use reverse() to change the order of your list again. Print the list to show it's back to its original order.
- Use sort() to change your list so it's stored in alphabetical order. Print the list to show that its order has been changed.
- Use sort() to change your list so it's stored in reverse alphabetical order. Print the list to show that its order has changed.

Try It Yourself

- **3-9. Dinner Guests**: Working with one of the programs from Exercises 3-4 through 3-7 (page 42), use len() to print a message indicating the number of people you are inviting to dinner.
- **3-10. Every Function**: Think of something you could store in a list. For example, you could make a list of mountains, rivers, countries, cities, languages, or anything else you'd like. Write a program that creates a list containing these items and then uses each function introduced in this chapter at least once.

Avoiding Index Errors When Working with Lists

- One type of error is common to see when you're working with lists for the first time. Let's say you have a list with three items, and you ask for the fourth item:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

```
Traceback (most recent call last):
  File "motorcycles.py", line 2, in <module>
    print(motorcycles[3])
IndexError: list index out of range
```

Avoiding Index Errors When Working with Lists

- Keep in mind that whenever you want to access the last item in a list you use the index -1. This will always work, even if your list has changed size since the last time you accessed it.
- The only time this approach will cause an error is when you request the last item from an empty list.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

```
'suzuki'
```

```
motorcycles = []
print(motorcycles[-1])
```

Avoiding Index Errors When Working with Lists

```
Traceback (most recent call last):
  File "motorcyles.py", line 3, in <module>
    print(motorcycles[-1])
IndexError: list index out of range
```

- **Note**: if an index error occurs and you can't figure out how to resolve it, try printing your list or just printing the length of your list. Your list might look much different than you thought it did, especially if it has been managed dynamically by your program. Seeing the actual list, or the exact number of items in your list, can help you sort out such logical errors.

Try It Yourself

- 3-11. Intentional Error: If you haven’t received an index error in one of your programs yet, try to make one happen. Change an index in one of your programs to produce an index error. Make sure you correct the error before closing the program.

Chapter 4: Working With Lists

What it is about?

- In Chapter 3 you learned how to make a simple list, and you learned to work with the individual elements in a list.
- In this chapter you'll learn how to loop through an entire list using just a few lines of code regardless of how long the list is.
- Looping allows you to take the same action, or set of actions, with every item in a list.
- As a result, you'll be able to work efficiently with lists of any length, including those with thousands or even millions of items.

Looping Through an Entire List

- Let’s use a for loop to print out each name in a list of magicians:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician)
```

```
alice
david
carolina
```

A Closer Look at Looping

- For example, here's a good way to start a for loop for a list of cats, a list of dogs, and a general list of items:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

- Using singular and plural names can help you identify whether a section of code is working with a single element from the list or the entire list.

Doing More Work Within a for Loop

- You can do just about anything with each item in a for loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
```

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

Doing More Work Within a for Loop

- You can also write as many lines of code as you like in the for loop.
- Every indented line following the line for magician in magicians is considered inside the loop, and each indented line is executed once for each value in the list.
- Therefore, you can do as much work as you like with each value in the list.

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Doing Something After a for Loop

- Start the new line without ‘indentation’’

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

print("Thank you, everyone. That was a great magic show!")
```

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

Avoiding Indentation Errors

- Python uses indentation to determine how a line, or group of lines, is related to the rest of the program.
- Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure.
- As you begin to write code that relies on proper indentation, you'll need to watch for a few common indentation errors.
- For example, people sometimes indent lines of code that don't need to be indented or forget to indent lines that need to be indented.

Avoiding Indentation Errors

- **Forgetting to Indent**
  - Always indent the line after the for statement in a loop. If you forget, Python will remind you:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
print(magician)
```

```
  File "magicians.py", line 3
    print(magician)
        ^
IndentationError: expected an indented block
```

Avoiding Indentation Errors

- **Forgetting to Indent Additional Lines**
  - Sometimes your loop will run without any errors but won't produce the expected result. This can happen when you're trying to do several tasks in a loop and you forget to indent some of its lines.

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Avoiding Indentation Errors

- **Indenting Unnecessarily**
  - If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

```
message = "Hello Python world!"
    print(message)
```

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

Avoiding Indentation Errors

- **Indenting Unnecessarily After the Loop**
  - Sometimes this prompts Python to report an error, but often this will result in a logical error.

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

    print("Thank you everyone, that was a great magic show!")
```

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you everyone, that was a great magic show!
```

Avoiding Indentation Errors

- **Forgetting the Colon**
  - The colon at the end of a for statement tells Python to interpret the next line as the start of a loop.
  - If you accidentally forget the colon, you'll get a syntax error because Python doesn't know what you're trying to do.

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians
    print(magician)
```

Try It Yourself

- **4-1. Pizzas**: Think of at least three kinds of your favorite pizza. Store these pizza names in a list, and then use a for loop to print the name of each pizza.
  - Modify your for loop to print a sentence using the name of the pizza instead of printing just the name of the pizza. For each pizza you should have one line of output containing a simple statement like I like pepperoni pizza.
  - Add a line at the end of your program, outside the for loop, that states how much you like pizza. The output should consist of three or more lines about the kinds of pizza you like and then an additional sentence, such as I really love pizza!

Try It Yourself

- **4-2. Animals**: Think of at least three different animals that have a common characteristic. Store the names of these animals in a list, and then use a for loop to print out the name of each animal.
  - Modify your program to print a statement about each animal, such as A dog would make a great pet.
  - Add a line at the end of your program stating what these animals have in common. You could print a sentence such as Any of these animals would make a great pet!

Making Numerical Lists

- **Using the range() Function**
  - Python’s range() function makes it easy to generate a series of numbers.
  - For example, you can use the range() function to print a series of numbers like this:

```
for value in range(1, 5):
    print(value)
```

```
1
2
3
4
```

Making Numerical Lists

- **Using the range() Function – continued**
  - The range() function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide.
  - Because it stops at that second value, the output never contains the end value, which would have been 5 in this case.
  - To print the numbers from 1 to 5, you would use range(1, 6):

```
for value in range(1, 6):
    print(value)
```

```
1
2
3
4
5
```

Making Numerical Lists

- **Using the range() Function – continued**
  - If your output is different than what you expect when you're using
  - range(), try adjusting your end value by 1.
  - You can also pass range() only one argument, and it will start the sequence of numbers at 0.
  - For example, range(6) would return the numbers from 0 through 5.

Making Numerical Lists

- **Using range() to Make a List of Numbers**
  - In the example in the previous section, we simply printed out a series of numbers. We can use list() to convert that same set of numbers into a list.

```
numbers = list(range(1, 6))
print(numbers)
```

```
[1, 2, 3, 4, 5]
```

Making Numerical Lists

- **Using range() to Make a List of Numbers - continued**
  - We can also use the range() function to tell Python to skip numbers in a given range. If you pass a third argument to range(), Python uses that value as a step size when generating numbers.

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

```
[2, 4, 6, 8, 10]
```

Making Numerical Lists

- **Using range() to Make a List of Numbers - continued**
  - You can create almost any set of numbers you want to using the range()function.
  - For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10).
  - In Python, two asterisks (**) represent exponents.

```
squares = []
for value in range(1, 11):
    square = value ** 2
    squares.append(square)

print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Making Numerical Lists

- **Simple Statistics with a List of Numbers**
  - The examples in this section use short lists of numbers in order to fit easily on the page. They would work just as well if your list contained a million or more numbers.

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

Making Numerical Lists

- **List Comprehensions**
  - The approach described earlier for generating the list squares consisted of using three or four lines of code.
  - A list comprehension allows you to generate this same list in just one line of code.
  - A list comprehension combines the for loop and the creation of new elements into one line, and automatically appends each new element.

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Try It Yourself

- **4-3. Counting to Twenty**: Use a for loop to print the numbers from 1 to 20, inclusive.
- **4-4. One Million**: Make a list of the numbers from one to one million, and then use a for loop to print the numbers. (If the output is taking too long, stop it by pressing ctrl-C or by closing the output window.)
- **4-5. Summing a Million**: Make a list of the numbers from one to one million, and then use min() and max() to make sure your list actually starts at one and ends at one million. Also, use the sum() function to see how quickly Python can add a million numbers.

Try It Yourself

- **4-6. Odd Numbers**: Use the third argument of the range() function to make a list of the odd numbers from 1 to 20. Use a for loop to print each number.
- **4-7. Threes**: Make a list of the multiples of 3 from 3 to 30. Use a for loop to print the numbers in your list.
- **4-8. Cubes**: A number raised to the third power is called a cube. For example, the cube of 2 is written as 2**3 in Python. Make a list of the first 10 cubes (that is, the cube of each integer from 1 through 10), and use a for loop to print out the value of each cube.
- **4-9. Cube Comprehension**: Use a list comprehension to generate a list of the first 10 cubes.

Working with Part of a List

- In Python, you can also work with a specific group of items in a list, which Python calls a ***slice***.
- **Slicing a List**
  - To make a slice, you specify the index of the first and last elements you want to work with. As with the range() function, Python stops one item before the second index you specify.

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[0:3])
```

```
['charles', 'martina', 'michael']
```

Working with Part of a List

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

```
['martina', 'michael', 'florence']
```

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

```
['charles', 'martina', 'michael', 'florence']
```

Working with Part of a List

- A similar syntax works if you want a slice that includes the end of a list.

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

```
['michael', 'florence', 'eli']
```

Working with Part of a List

- Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice players [-3:]:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

- This prints the names of the last three players and would continue to work as the list of players changes in size.

- **Note**: You can include a third value in the brackets indicating a slice. If a third value is included, this tells Python how many items to skip between items in the specified range.

Working with Part of a List

- **Looping Through a Slice**
  - You can use a slice in a for loop if you want to loop through a subset of the elements in a list.

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
for player in players[:3]:
    print(player.title())
```

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Working with Part of a List

- **Copying a List**
  - To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index ([:]). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list.

```
my_foods = ['pizza', 'falafel', 'carrot cake']
friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

Working with Part of a List

- To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favorite foods.

```
my_foods = ['pizza', 'falafel', 'carrot cake']
friend_foods = my_foods[:]

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

Working with Part of a List

- **Be careful**: If we had simply set friend_foods equal to my_foods, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice.

```
my_foods = ['pizza', 'falafel', 'carrot cake']

# This doesn't work:
friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Working with Part of a List

- Instead of storing a copy of my_foods in friend_foods, we set friend_foods equal to my_foods. This syntax actually tells Python to associate the new variable friend_foods with the list that is already associated with my_foods, so now both variables point to the same list.

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

Working with Part of a List

- **Note**: Don’t worry about the details in this example for now. Basically, if you’re trying to work with a copy of a list and you see unexpected behavior, make sure you are copying the list using a slice, as we did in the first example.

Try it yourself

- **4-10. Slices:** Using one of the programs you wrote in this chapter, add several lines to the end of the program that do the following:
  - Print the message ***The first three items in the list are***:. Then use a slice to print the first three items from that program's list.
  - Print the message **Three items from the middle of the list are**:. Use a slice to print three items from the middle of the list.
  - Print the message **The last three items in the list are**:. Use a slice to print the last three items in the list.

Try it yourself

- **4-11. My Pizzas**, Your Pizzas: Start with your program from Exercise 4-1 (page 56). Make a copy of the list of pizzas, and call it friend_pizzas. Then, do the following:
  - Add a new pizza to the original list.
  - Add a different pizza to the list friend_pizzas.
  - Prove that you have two separate lists. Print the message **My favorite pizzas are:**, and then use a for loop to print the first list.
  - Print the message **My friend's favorite pizzas are:**, and then use a for loop to print the second list. Make sure each new pizza is stored in the appropriate list.
- **4-12. More Loops**: All versions of foods.py in this section have avoided using for loops when printing to save space. Choose a version of foods.py, and write two for loops to print each list of foods.

Tuples

- The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change.
- Python refers to values that cannot change as immutable, and an immutable list is called a tuple.

Tuples

- **Defining a Tuple**
  - A tuple looks just like a list except you use parentheses instead of square brackets.
  - Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

```
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
```

```
200
50
```

Tuples

- Let’s see what happens if we try to change one of the items in the tuple dimensions.

```
dimensions = (200, 50)
dimensions[0] = 250
```

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

Tuples

- Note: Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple with one element, you need to include a trailing comma:

```
my_t = (3,)
```

- It doesn't often make sense to build a tuple with one element, but this can happen when tuples are generated automatically.

Tuples

- **Looping Through All Values in a Tuple**

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

```
200
50
```

Tuples

- **Writing over a Tuple**
  - Although you can't modify a tuple, you can assign a new value to a variable that represents a tuple.

```
dimensions = (200, 50)
print("Original dimensions:")
for dimension in dimensions:
    print(dimension)

dimensions = (400, 100)
print("\nModified dimensions:")
for dimension in dimensions:
    print(dimension)
```

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

Try it yourself

- **4-13. Buffet**: A buffet-style restaurant offers only five basic foods. Think of five simple foods, and store them in a tuple.
  - Use a for loop to print each food the restaurant offers.
  - Try to modify one of the items, and make sure that Python rejects the change.
  - The restaurant changes its menu, replacing two of the items with different foods. Add a line that rewrites the tuple, and then use a for loop to print each of the items on the revised menu.

Styling Your Code

- **The Style Guide**
  - When someone wants to make a change to the Python language, they write a Python Enhancement Proposal (PEP). One of the oldest PEPs is PEP 8, which instructs Python programmers on how to style their code.
  - The Python style guide was written with the understanding that code is read more often than it is written.
  - Given the choice between writing code that's easier to write or code that's easier to read, Python programmers will almost always encourage you to write code that's easier to read.

Styling Your Code

- **The Style Guide – cont.**
- The following guidelines will help you write clear code from the start:
  - **Indentation**
    - PEP 8 recommends that you use four spaces per indentation level. Using four spaces improves readability while leaving room for multiple levels of indentation on each line.
  - **Line Length**
    - Many Python programmers recommend that each line should be less than 80 characters.
    - **Note**: Appendix B shows you how to configure your text editor so it always inserts four spaces each time you press the tab key and shows a vertical guideline to help you follow the 79-character limit.

Styling Your Code

- **The Style Guide – cont.**
  - **Blank Lines**
    - To group parts of your program visually, use blank lines. You should use blank lines to organize your files, but don’t do so excessively. By following the examples provided in this book, you should strike the right balance. For example, if you have five lines of code that build a list, and then another three lines that do something with that list, it’s appropriate to place a blank line between the two sections.

Try it yourself

- **4-14. PEP 8**: Look through the original PEP 8 style guide at https://python.org/dev/peps/pep-0008/. You won't use much of it now, but it might be interesting to skim through it.
- **4-15. Code Review**: Choose three of the programs you've written in this chapter and modify each one to comply with PEP 8:
  - Use four spaces for each indentation level. Set your text editor to insert four spaces every time you press tab, if you haven't already done so (see Appendix B for instructions on how to do this).
  - Use less than 80 characters on each line, and set your editor to show a vertical guideline at the 80th character position.
  - Don't use blank lines excessively in your program files.

Chapter 5: IF statements

A Simple Example

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

```
Audi
BMW
Subaru
Toyota
```

Conditional Tests

- **Checking for Equality**
  - Most conditional tests compare the current value of a variable to a specific value of interest.

```
>>> car = 'bmw'
>>> car == 'bmw'
True
```

```
>>> car = 'audi'
>>> car == 'bmw'
False
```

Conditional Tests

- **Ignoring Case When Checking for Equality**
  - Testing for equality is case sensitive in Python.

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Conditional Tests

- **Ignoring Case When Checking for Equality - continued**
  - In the previous example, this test would return True no matter how the value 'Audi' is formatted because the test is now case insensitive.
  - The lower() function doesn’t change the value that was originally stored in car, so you can do this kind of comparison without affecting the original variable.

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
>>> car
'Audi'
```

Conditional Tests

- **Ignoring Case When Checking for Equality - continued**
  - How can this be useful?
  - Websites enforce certain rules for the data that users enter in a manner similar to this. For example, a site might use a conditional test like this to ensure that every user has a truly unique username, not just a variation on the capitalization of another person’s username. When someone submits a new username, that new username is converted to lowercase and compared to the lowercase versions of all existing usernames. During this check, a username like 'John' will be rejected if any variation of 'john’ is already in use.

Conditional Tests

- **Checking for Inequality**
  - The exclamation point represents not, as it does in many programming languages.

```
requested_topping = 'mushrooms'

if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

```
Hold the anchovies!
```

Conditional Tests

- **Numerical Comparisons**

```
>>> age = 18
>>> age == 18
True
```

```
answer = 17

if answer != 42:
    print("That is not the correct answer. Please try again!")
```

```
That is not the correct answer. Please try again!
```

Conditional Tests

- **Numerical Comparisons - continued**
  - You can include various mathematical comparisons in your conditional statements as well.

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking Multiple Conditions

- **Using and to Check Multiple Conditions**
  - To improve readability, you can use parentheses around the individual tests, but they are not required.

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 22
>>> age_0 >= 21 and age_1 >= 21
True
```

```
(age_0 >= 21) and (age_1 >= 21)
```

Checking Multiple Conditions

- **Using or to Check Multiple Conditions**

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Checking Whether a Value Is in a List

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
>>> 'mushrooms' in requested_toppings
True
>>> 'pepperoni' in requested_toppings
False
```

Checking Whether a Value Is Not in a List

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'

if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")
```

```
Marie, you can post a response if you wish.
```

Boolean Expressions

- As you learn more about programming, you'll hear the term Boolean expression at some point. A Boolean expression is just another name for a conditional test. A Boolean value is either True or False, just like the value of a conditional expression after it has been evaluated.
- Boolean values are often used to keep track of certain conditions, such as whether a game is running or whether a user can edit certain content on a website.
- Boolean values provide an efficient way to track the state of a program or a particular condition that is important in your program.

```
game_active = True
can_edit = False
```

Try It Yourself

- **5-1. Conditional Tests**: Write a series of conditional tests. Print a statement describing each test and your prediction for the results of each test. Your code should look something like this:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

  - Look closely at your results, and make sure you understand why each line evaluates to True or False.
  - Create at least ten tests. Have at least five tests evaluate to True and another five tests evaluate to False.

Try It Yourself

- **5-2. More Conditional Tests**: You don’t have to limit the number of tests you create to ten. If you want to try more comparisons, write more tests and add them to conditional_tests.py. Have at least one True and one False result for each of the following:
  - Tests for equality and inequality with strings
  - Tests using the lower() method
  - Numerical tests involving equality and inequality, greater than and less than, greater than or equal to, and less than or equal to
  - Tests using the and keyword and the or keyword
  - Test whether an item is in a list
  - Test whether an item is not in a list

if Statements

- **Simple if Statements**
  - The simplest kind of if statement has one test and one action.
  - If the conditional test evaluates to True, Python executes the code following the if statement.
  - If the test evaluates to False, Python ignores the code following the if statement.

```
if conditional_test:
    do something
```

if Statements

- **Simple if statements – continued**
  - Indentation plays the same role in if statements as it did in for loops.

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
```

```
You are old enough to vote!
```

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

```
You are old enough to vote!
Have you registered to vote yet?
```

if-else Statements

- Often, you'll want to take one action when a conditional test passes and a different action in all other cases.
- Python's if-else syntax makes this possible. It allows you to define an action or set of actions that are executed when the conditional test fails.

if-else Statements

```
age = 17
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

The if-elif-else Chain

- Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's if-elif-else syntax. Python executes only one block in an if-elif-else chain. It runs each conditional test in order until one passes.
- When a test passes, the code following that test is executed and Python skips the rest of the tests.

The if-elif-else Chain

```
age = 12

if age < 4:
    print("Your admission cost is $0.")
elif age < 18:
    print("Your admission cost is $25.")
else:
    print("Your admission cost is $40.")
```

```
Your admission cost is $25.
```

The if-elif-else Chain

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40

print(f"Your admission cost is ${price}.")
```

Using Multiple elif Blocks

- You can use as many elif blocks in your code as you like.

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
else:
    price = 20

print(f"Your admission cost is ${price}.")
```

Omitting the else Block

- Python does not require an else block at the end of an if-elif chain. Sometimes an else block is useful; sometimes it is clearer to use an additional elif statement that catches the specific condition of interest.

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f"Your admission cost is ${price}.")
```

Testing Multiple Conditions

- The if-elif-else chain is powerful, but it's only appropriate to use when you just need one test to pass. As soon as Python finds one test that passes, it skips the rest of the tests.
- This behavior is beneficial, because it's efficient and allows you to test for one specific condition.
- However, sometimes it's important to check all of the conditions of interest. In this case, you should use a series of simple if statements with no elif or else blocks.
- This technique makes sense when more than one condition could be True, and you want to act on every condition that is True.

Testing Multiple Conditions

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

Testing Multiple Conditions

- The previous code would not work properly if we used an if-elif-else block, because the code would stop running after only one test passes.

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

```
Adding mushrooms.

Finished making your pizza!
```

Try It Yourself

- **5-3. Alien Colors #1**: Imagine an alien was just shot down in a game. Create a variable called alien_color and assign it a value of 'green', 'yellow', or 'red'.
  - Write an if statement to test whether the alien’s color is green. If it is, print a message that the player just earned 5 points.
  - Write one version of this program that passes the if test and another that fails. (The version that fails will have no output.)

Try It Yourself

- **5-4. Alien Colors #2**: Choose a color for an alien as you did in Exercise 5-3, and write an if-else chain.
  - If the alien's color is green, print a statement that the player just earned 5 points for shooting the alien.
  - If the alien's color isn't green, print a statement that the player just earned 10 points.
  - Write one version of this program that runs the if block and another that runs the else block.

Try It Yourself

- **5-5. Alien Colors #3:** Turn your if-else chain from Exercise 5-4 into an if-elif-else chain.
  - If the alien is green, print a message that the player earned 5 points.
  - If the alien is yellow, print a message that the player earned 10 points.
  - If the alien is red, print a message that the player earned 15 points.
  - Write three versions of this program, making sure each message is printed for the appropriate color alien.

Try It Yourself

- **5-6. Stages of Life:** Write an if-elif-else chain that determines a person's stage of life. Set a value for the variable age, and then:
  - If the person is less than 2 years old, print a message that the person is a baby.
  - If the person is at least 2 years old but less than 4, print a message that the person is a toddler.
  - If the person is at least 4 years old but less than 13, print a message that the person is a kid.
  - If the person is at least 13 years old but less than 20, print a message that the person is a teenager.
  - If the person is at least 20 years old but less than 65, print a message that the person is an adult.
  - If the person is age 65 or older, print a message that the person is an elder.

Try It Yourself

- **5-7. Favorite Fruit**: Make a list of your favorite fruits, and then write a series of independent if statements that check for certain fruits in your list.
  - Make a list of your three favorite fruits and call it favorite_fruits.
  - Write five if statements. Each should check whether a certain kind of fruit is in your list. If the fruit is in your list, the if block should print a statement, such as You really like bananas!

Using if Statements with Lists

- You can do some interesting work when you combine lists and if statements. You can watch for special values that need to be treated differently than other values in the list. You can manage changing conditions efficiently, such as the availability of certain items in a restaurant throughout a shift. You can also begin to prove that your code works as you expect it to in all possible situations.

Using if Statements with Lists

- **Checking for Special Items**

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

Using if Statements with Lists

- **Checking for Special Items - continued**
  - But what if the pizzeria runs out of green peppers? An if statement inside the for loop can handle this situation appropriately.

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Using if Statements with Lists

- **Checking That a List Is Not Empty**

```
requested_toppings = []

if requested_toppings:
    for requested_topping in requested_toppings:
        print(f"Adding {requested_topping}.")
    print("\nFinished making your pizza!")
else:
    print("Are you sure you want a plain pizza?")
```

```
Are you sure you want a plain pizza?
```

Using if Statements with Lists

- **Using Multiple Lists**

```
available_toppings = ['mushrooms', 'olives', 'green peppers',
                      'pepperoni', 'pineapple', 'extra cheese']

requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping in available_toppings:
        print(f"Adding {requested_topping}.")
    else:
        print(f"Sorry, we don't have {requested_topping}.")

print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

Try It Yourself

- **5-8. Hello Admin**: Make a list of five or more usernames, including the name 'admin'. Imagine you are writing code that will print a greeting to each user after they log in to a website. Loop through the list, and print a greeting to each user:
  - If the username is 'admin', print a special greeting, such as Hello admin, would you like to see a status report?
  - Otherwise, print a generic greeting, such as Hello Jaden, thank you for logging in again.
- **5-9. No Users**: Add an if test to hello_admin.py to make sure the list of users is not empty.
  - If the list is empty, print the message We need to find some users!
  - Remove all of the usernames from your list, and make sure the correct message is printed.

Try It Yourself

- **5-10. Checking Usernames**: Do the following to create a program that simulates how websites ensure that everyone has a unique username.
  - Make a list of five or more usernames called current_users.
  - Make another list of five usernames called new_users. Make sure one or two of the new usernames are also in the current_users list.
  - Loop through the new_users list to see if each new username has already been used. If it has, print a message that the person will need to enter a new username. If a username has not been used, print a message saying that the username is available.
  - Make sure your comparison is case insensitive. If 'John' has been used, 'JOHN' should not be accepted. (To do this, you'll need to make a copy of current_users containing the lowercase versions of all existing users.)

Try It Yourself

- **5-11. Ordinal Numbers**: Ordinal numbers indicate their position in a list, such as 1st or 2nd. Most ordinal numbers end in th, except 1, 2, and 3.
  - Store the numbers 1 through 9 in a list.
  - Loop through the list.
  - Use an if-elif-else chain inside the loop to print the proper ordinal ending for each number. Your output should read "1st 2nd 3rd 4th 5th 6th 7th 8th 9th", and each result should be on a separate line.

Styling Your if Statements

- In every example in this chapter, you've seen good styling habits. The only recommendation PEP 8 provides for styling conditional tests is to use a single space around comparison operators, such as ==, >=, <=.

```
if age < 4:
```

is better than:

```
if age<4:
```

Try It Yourself

- **5-12. Styling if statements**: Review the programs you wrote in this chapter, and make sure you styled your conditional tests appropriately.
- **5-13. Your Ideas**: At this point, you're a more capable programmer than you were when you started this book. Now that you have a better sense of how real-world situations are modeled in programs, you might be thinking of some problems you could solve with your own programs. Record any new ideas you have about problems you might want to solve as your programming skills continue to improve. Consider games you might want to write, data sets you might want to explore, and web applications you'd like to create.

Chapter 6: Dictionaries

Introduction

- Understanding dictionaries allows you to model a variety of real-world objects more accurately. You'll be able to create a dictionary representing a person and then store as much information as you want about that person. You can store their name, age, location, profession, and any other aspect of a person you can describe.
- A Simple Dictionary

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

```
green
5
```

Working with Dictionaries

- A dictionary in Python is a collection of key-value pairs. Each key is connected to a value, and you can use a key to access the value associated with that key.
- In Python, a dictionary is wrapped in braces, {}.
- *A key-value pair* is a set of values associated with each other.
- When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas.
- You can store as many key-value pairs as you want in a dictionary.

Accessing Values in a Dictionary

- To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets [].

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

```
green
```

```
alien_0 = {'color': 'green', 'points': 5}

new_points = alien_0['points']
print(f"You just earned {new_points} points!")
```

```
You just earned 5 points!
```

Adding New Key-Value Pairs

- Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time.

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

alien_0['x_position'] = 0
alien_0['y_position'] = 25
print(alien_0)
```

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

Starting with an Empty Dictionary

- It’s sometimes convenient, or even necessary, to start with an empty dictionary and then add each new item to it.

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

```
{'color': 'green', 'points': 5}
```

Modifying Values in a Dictionary

- To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key.

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

```
The alien is green.
The alien is now yellow.
```

Modifying Values in a Dictionary

- For a more interesting example, let's track the position of an alien that can move at different speeds.

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")

# Move the alien to the right.
# Determine how far to move the alien based on its current speed.
if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # This must be a fast alien.
    x_increment = 3

# The new position is the old position plus the increment.
alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")
```

```
Original x-position: 0
New x-position: 2
```

Removing Key-Value Pairs

- When you no longer need a piece of information that's stored in a dictionary, you can use the del statement to completely remove a key-value pair.

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

- Note: Be aware that the deleted key-value pair is removed permanently.

A Dictionary of Similar Objects

- You can also use a dictionary to store one kind of information about many objects.
- When you know you'll need more than one line to define a dictionary, press enter after the opening brace. Then indent the next line one.
- Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair and indent it one level so it aligns with the keys in the dictionary.
- It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }
```

A Dictionary of Similar Objects

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}.")
```

Using get() to Access Values

- Using keys in square brackets to retrieve the value you're interested in from a dictionary might cause one potential problem: if the key you ask for doesn't exist, you'll get an error.

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
KeyError: 'points'
```

Using get() to Access Values

- The get() method requires a key as a first argument. As a second optional argument, you can pass the value to be returned if the key doesn’t exist.
- If the key 'points' exists in the dictionary, you’ll get the corresponding value. If it doesn’t, you get the default value. In this case, points doesn’t exist, and we get a clean message instead of an error

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

```
No point value assigned.
```

Using get() to Access Values

- Note:
- If you leave out the second argument in the call to get() and the key doesn’t exist, Python will return the value None. The special value None means “no value exists.” This is not an error: it’s a special value meant to indicate the absence of a value.

Try It Yourself

- **6-1. Person**: Use a dictionary to store information about a person you know. Store their first name, last name, age, and the city in which they live. You should have keys such as first_name, last_name, age, and city. Print each piece of information stored in your dictionary.
- **6-2. Favorite Numbers**: Use a dictionary to store people's favorite numbers. Think of five names, and use them as keys in your dictionary. Think of a favorite number for each person, and store each as a value in your dictionary. Print each person's name and their favorite number. For even more fun, poll a few friends and get some actual data for your program.

Try It Yourself

- **6-3. Glossary**: A Python dictionary can be used to model an actual dictionary. However, to avoid confusion, let's call it a glossary.
  - Think of five programming words you've learned about in the previous chapters. Use these words as the keys in your glossary, and store their meanings as values.
  - Print each word and its meaning as neatly formatted output. You might print the word followed by a colon and then its meaning, or print the word on one line and then print its meaning indented on a second line. Use the newline character (\n) to insert a blank line between each word-meaning pair in your output.

Looping Through a Dictionary

- Looping Through All Key-Value Pairs

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }

for key, value in user_0.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

```
for k, v in user_0.items()
```

Looping Through a Dictionary

- Looping Through All Key-Value Pairs – continued

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

for name, language in favorite_languages.items():
    print(f"{name.title()}'s favorite language is {language.title()}.")
```

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.
```

Looping Through a Dictionary

- **Looping Through All the Keys in a Dictionary**
  - The keys() method is useful when you don’t need to work with all of the values in a dictionary.

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

for name in favorite_languages.keys():
    print(name.title())
```

```
Jen
Sarah
Edward
Phil
```

Looping Through a Dictionary

- **Looping Through All the Keys in a Dictionary – continued**
  - Looping through the keys is actually the default behavior when looping through a dictionary, so this code would have exactly the same output if you wrote . . .

```
for name in favorite_languages:
```

  - Rather than

```
for name in favorite_languages.keys():
```

  - You can choose to use the keys() method explicitly if it makes your code easier to read, or you can omit it if you wish.

Looping Through a Dictionary

- **Looping Through All the Keys in a Dictionary – continued**
  - You can access the value associated with any key you care about inside the loop by using the current key.

```
favorite_languages = {
    --snip--
    }

friends = ['phil', 'sarah']
for name in favorite_languages.keys():
    print(name.title())

    if name in friends:
        language = favorite_languages[name].title()
        print(f"\t{name.title()}, I see you love {language}!")
```

Looping Through a Dictionary

- **Looping Through All the Keys in a Dictionary – continued**

```
Hi Jen.
Hi Sarah.
    Sarah, I see you love C!
Hi Edward.

Hi Phil.
    Phil, I see you love Python!
```

Looping Through a Dictionary

- **Looping Through All the Keys in a Dictionary – continued**
  - You can also use the keys() method to find out if a particular person was polled. This time, let's find out if Erin took the poll.

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

if 'erin' not in favorite_languages.keys():
    print("Erin, please take our poll!")
```

```
Erin, please take our poll!
```

Looping Through a Dictionary

- **Looping Through a Dictionary's Keys in a Particular Order**
  - One way to do this is to sort the keys as they're returned in the for loop. You can use the sorted() function to get a copy of the keys in order.

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

for name in sorted(favorite_languages.keys()):
    print(f"{name.title()}, thank you for taking the poll.")
```

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Looping Through a Dictionary

- **Looping Through All Values in a Dictionary**
  - If you are primarily interested in the values that a dictionary contains, you can use the values() method to return a list of values without any keys.

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
for language in favorite_languages.values():
    print(language.title())
```

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

Looping Through a Dictionary

- **Looping Through All Values in a Dictionary – continued**
  - This approach pulls all the values from the dictionary without checking for repeats. That might work fine with a small number of values, but in a poll with a large number of respondents, this would result in a very repetitive list. To see each language chosen without repetition, we can use a set. A set is a collection in which each item must be unique.

```
favorite_languages = {
    --snip--
    }

print("The following languages have been mentioned:")
for language in set(favorite_languages.values()):
    print(language.title())
```

```
The following languages have been mentioned:
Python
C
Ruby
```

Looping Through a Dictionary

- **Looping Through All Values in a Dictionary – continued**
  - **Note**:
    - You can build a set directly using braces and separating the elements with commas:

```
>>> languages = {'python', 'ruby', 'python', 'c'}
>>> languages
{'ruby', 'python', 'c'}
```

    - It's easy to mistake sets for dictionaries because they're both wrapped in braces. When you see braces but no key-value pairs, you're probably looking at a set. Unlike lists and dictionaries, sets do not retain items in any specific order

Try It Yourself

- **6-4. Glossary 2**: Now that you know how to loop through a dictionary, clean up the code from Exercise 6-3 (page 99) by replacing your series of print() calls with a loop that runs through the dictionary's keys and values. When you're sure that your loop works, add five more Python terms to your glossary. When you run your program again, these new words and meanings should automatically be included in the output.
- **6-5. Rivers**: Make a dictionary containing three major rivers and the country each river runs through. One key-value pair might be 'nile': 'egypt'.
  - Use a loop to print a sentence about each river, such as The Nile runs through Egypt.
  - Use a loop to print the name of each river included in the dictionary.
  - Use a loop to print the name of each country included in the dictionary.

Try It Yourself

- **6-6. Polling**: Use the code in favorite_languages.py (page 97).
  - Make a list of people who should take the favorite languages poll. Include some names that are already in the dictionary and some that are not.
  - Loop through the list of people who should take the poll. If they have already taken the poll, print a message thanking them for responding. If they have not yet taken the poll, print a message inviting them to take the poll.

Nesting

- Sometimes you'll want to store multiple dictionaries in a list, or a list of items as a value in a dictionary. This is called nesting. You can nest dictionaries inside a list, a list of items inside a dictionary, or even a dictionary inside another dictionary.

Nesting

- **A List of Dictionaries**

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)
```

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

Nesting

- **A List of Dictionaries – continued**
  - A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example we use range() to create a fleet of 30 aliens.

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

# Show the first 5 aliens.
for alien in aliens[:5]:
    print(alien)
print("...")

# Show how many aliens have been created.
print(f"Total number of aliens: {len(aliens)}")
```

Nesting

- **A List of Dictionaries – continued**

```
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...

Total number of aliens: 30
```

Nesting

- **A List of Dictionaries – continued**

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range (30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Show the first 5 aliens.
for alien in aliens[:5]:
    print(alien)
print("...")
```

Nesting

- **A List of Dictionaries – continued**

```
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...
```

Nesting

- **A List in a Dictionary**
  - Rather than putting a dictionary inside a list, it's sometimes useful to put a list inside a dictionary.

```
# Store information about a pizza being ordered.
pizza = {
    'crust': 'thick',
    'toppings': ['mushrooms', 'extra cheese'],
    }

# Summarize the order.
print(f"You ordered a {pizza['crust']}-crust pizza "
    "with the following toppings:")

for topping in pizza['toppings']:
    print("\t" + topping)
```

Nesting

- **A List in a Dictionary – continued**

```
You ordered a thick-crust pizza with the following toppings:
    mushrooms
    extra cheese
```

Nesting

- **A List in a Dictionary – continued**
  - You can nest a list inside a dictionary any time you want more than one value to be associated with a single key in a dictionary.

```
favorite_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
    }

for name, languages in favorite_languages.items():
    print(f"\n{name.title()}'s favorite languages are:")
    for language in languages:
        print(f"\t{language.title()}")
```

Nesting

- **A List in a Dictionary – continued**

```
Jen's favorite languages are:
    Python
    Ruby

Sarah's favorite languages are:
    C

Phil's favorite languages are:
    Python
    Haskell

Edward's favorite languages are:
    Ruby
    Go
```

Nesting

- **Note:**
- You should not nest lists and dictionaries too deeply. If you're nesting items much deeper than what you see in the preceding examples or you're working with someone else's code with significant levels of nesting, most likely a simpler way to solve the problem exists.

Nesting

- **A Dictionary in a Dictionary**
  - You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do.
  - For example, if you have several users for a website, each with a unique username, you can use the usernames as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username.

Nesting

- **A Dictionary in a Dictionary – continued**

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },

    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },

    }

for username, user_info in users.items():
    print(f"\nUsername: {username}")
    full_name = f"{user_info['first']} {user_info['last']}"
    location = user_info['location']

    print(f"\tFull name: {full_name.title()}")
    print(f"\tLocation: {location.title()}")
```

Nesting

- **A Dictionary in a Dictionary – continued**

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton

Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Try It Yourself

- **6-7. People**: Start with the program you wrote for Exercise 6-1 (page 99). Make two new dictionaries representing different people, and store all three dictionaries in a list called people. Loop through your list of people. As you loop through the list, print everything you know about each person.
- **6-8. Pets**: Make several dictionaries, where each dictionary represents a different pet. In each dictionary, include the kind of animal and the owner's name. Store these dictionaries in a list called pets. Next, loop through your list and as you do, print everything you know about each pet.
- **6-9. Favorite Places**: Make a dictionary called favorite_places. Think of three names to use as keys in the dictionary, and store one to three favorite places for each person. To make this exercise a bit more interesting, ask some friends to name a few of their favorite places. Loop through the dictionary, and print each person's name and their favorite places.

Try It Yourself

- **6-10. Favorite Numbers**: Modify your program from Exercise 6-2 (page 99) so each person can have more than one favorite number. Then print each person's name along with their favorite numbers.
- **6-11. Cities**: Make a dictionary called cities. Use the names of three cities as keys in your dictionary. Create a dictionary of information about each city and include the country that the city is in, its approximate population, and one fact about that city. The keys for each city's dictionary should be something like country, population, and fact. Print the name of each city and all of the information you have stored about it.
- **6-12. Extensions**: We're now working with examples that are complex enough that they can be extended in any number of ways. Use one of the example programs from this chapter, and extend it by adding new keys and values, changing the context of the program or improving the formatting of the output.

Chapter 7: User Input and while Loops

Introduction

- Most programs are written to solve an end user’s problem. To do so, you usually need to get some information from the user.
- To do this, you’ll use the input() function.

How the input() Function Works

- The input() function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it assigns that input to a variable to make it convenient for you to work with.
- Try this:

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

- **Note**: Sublime Text and many other editors don't run programs that prompt the user for input. You can use these editors to write programs that prompt for input, but you'll need to run these programs from a terminal. See "Running Python Programs from a Terminal" on page 12.

How the input() Function Works

- **Writing Clear Prompts**
  - Each time you use the input() function, you should include a clear, easy-to follow prompt that tells the user exactly what kind of information you're looking for.

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

```
Please enter your name: Eric
Hello, Eric!
```

How the input() Function Works

- **Writing Clear Prompts – continued**
  - Sometimes you'll want to write a prompt that's longer than one line. For example, you might want to tell the user why you're asking for certain input. You can assign your prompt to a variable and pass that variable to the input() function. This allows you to build your prompt over several lines, then write a clean input() statement.

```
prompt = "If you tell us who you are, we can personalize the messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print(f"\nHello, {name}!")
```

```
If you tell us who you are, we can personalize the messages you see.
What is your first name? Eric

Hello, Eric!
```

How the input() Function Works

- **Using int() to Accept Numerical Input**
  - When you use the input() function, Python interprets everything the user enters as a string.

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() >= int()
```

How the input() Function Works

- **Using int() to Accept Numerical Input – continued**
  - We can resolve this issue by using the int() function, which tells Python to treat the input as a numerical value.

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age = int(age)
>>> age >= 18
True
```

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

How the input() Function Works

- **The Modulo Operator**
  - A useful tool for working with numerical information is the modulo operator (%), which divides one number by another number and returns the remainder.

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

How the input() Function Works

- **The Modulo Operator – continued**
  - When one number is divisible by another number, the remainder is 0, so the modulo operator always returns 0. You can use this fact to determine if a number is even or odd.

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

```
Enter a number, and I'll tell you if it's even or odd: 42

The number 42 is even.
```

Try It Yourself

- **7-1. Rental Car**: Write a program that asks the user what kind of rental car they would like. Print a message about that car, such as “Let me see if I can find you a Subaru.”
- **7-2. Restaurant Seating**: Write a program that asks the user how many people are in their dinner group. If the answer is more than eight, print a message saying they’ll have to wait for a table. Otherwise, report that their table is ready.
- **7-3. Multiples of Ten**: Ask the user for a number, and then report whether the number is a multiple of 10 or not.

Introducing while Loops

- The for loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the while loop runs as long as, or while, a certain condition is true.

Introducing while Loops

- **The while Loop in Action**
  - You can use a while loop to count up through a series of numbers. For example, the following while loop counts from 1 to 5.

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

```
1
2
3
4
5
```

Introducing while Loops

- **Letting the User Choose When to Quit**
  - We can make the parrot.py program run as long as the user wants by putting most of the program inside a while loop. We'll define a quit value and then keep the program running as long as the user has not entered the quit value.

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)
    print(message)
```

Introducing while Loops

- **Letting the User Choose When to Quit – continued**

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

Introducing while Loops

- **Letting the User Choose When to Quit – continued**
  - The previous program works well, except that it prints the word 'quit' as if it were an actual message. A simple if test fixes this.

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Introducing while Loops

- **Letting the User Choose When to Quit – continued**

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Introducing while Loops

- **Using a Flag**
  - For a program that should run only as long as many conditions are true, you can define one variable that determines whether or not the entire program is active. This variable, called a flag, acts as a signal to the program.
  - We can write our programs so they run while the flag is set to True and stop running when any of several events sets the value of the flag to False. As a result, our overall while statement needs to check only one condition: whether or not the flag is currently True. Then, all our other tests (to see if an event has occurred that should set the flag to False) can be neatly organized in the rest of the program.

Introducing while Loops

- **Using a Flag – continued**

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

- This program has the same output as the previous example where we placed the conditional test directly in the while statement. But now that we have a flag to indicate whether the overall program is in an active state, it would be easy to add more tests (such as elif statements) for events that should cause active to become False. This is useful in complicated programs like games in which there may be many events that should each make the program stop running.

Introducing while Loops

- **Using break to Exit a Loop**
  - To exit a while loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the break statement.
  - The break statement directs the flow of your program; you can use it to control which lines of code are executed and which aren't, so the program only executes code that you want it to, when you want it to.

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

Introducing while Loops

- **Using break to Exit a Loop – continued**

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

- **Note**: You can use the break statement in any of Python's loops. For example, you could use break to quit a for loop that's working through a list or a dictionary.

Introducing while Loops

- **Using continue in a Loop**
  - Rather than breaking out of a loop entirely without executing the rest of its code, you can use the continue statement to return to the beginning of the loop based on the result of a conditional test.

```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue

    print(current_number)
```

```
1
3
5
7
9
```

Introducing while Loops

- **Avoiding Infinite Loops**
  - Every while loop needs a way to stop running so it won't continue to run forever. For example, this counting loop should count from 1 to 5.

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

```
# This loop runs forever!
x = 1
while x <= 5:
    print(x)
```

Introducing while Loops

- **Avoiding Infinite Loops**
  - Every programmer accidentally writes an infinite while loop from time to time, especially when a program's loops have subtle exit conditions. If your program gets stuck in an infinite loop, press ctrl-C or just close the terminal window displaying your program's output.
  - To avoid writing infinite loops, test every while loop and make sure the loop stops when you expect it to.
  - Note:
    - Sublime Text and some other editors have an embedded output window. This can make it difficult to stop an infinite loop, and you might have to close the editor to end the loop. Try clicking in the output area of the editor before pressing ctrl-C, and you should be able to cancel an infinite loop.

Try It Yourself

- **7-4. Pizza Toppings**: Write a loop that prompts the user to enter a series of pizza toppings until they enter a 'quit' value. As they enter each topping, print a message saying you'll add that topping to their pizza.
- **7-5. Movie Tickets**: A movie theater charges different ticket prices depending on a person's age. If a person is under the age of 3, the ticket is free; if they are between 3 and 12, the ticket is $10; and if they are over age 12, the ticket is $15. Write a loop in which you ask users their age, and then tell them the cost of their movie ticket.

Try It Yourself

- **7-6. Three Exits**: Write different versions of either Exercise 7-4 or Exercise 7-5 that do each of the following at least once:
  - Use a conditional test in the while statement to stop the loop.
  - Use an active variable to control how long the loop runs.
  - Use a break statement to exit the loop when the user enters a 'quit' value.
- **7-7. Infinity**: Write a loop that never ends, and run it. (To end the loop, press ctrl-C or close the window displaying the output.)

Using a while Loop with Lists and Dictionaries

- A for loop is effective for looping through a list, but you shouldn’t modify a list inside a for loop because Python will have trouble keeping track of the items in the list.
- To modify a list as you work through it, use a while loop. Using while loops with lists and dictionaries allows you to collect, store, and organize lots of input to examine and report on later.

Using a while Loop with Lists and Dictionaries

- **Moving Items from One List to Another**
  - Consider a list of newly registered but unverified users of a website. After we verify these users, how can we move them to a separate list of confirmed users?
  - One way would be to use a while loop to pull users from the list of unconfirmed users as we verify them and then add them to a separate list of confirmed users.

Using a while Loop with Lists and Dictionaries

- **Moving Items from One List to Another – continued**

```
# Start with users that need to be verified,
#  and an empty list to hold confirmed users.
unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verify each user until there are no more unconfirmed users.
#  Move each verified user into the list of confirmed users.
while unconfirmed_users:
    current_user = unconfirmed_users.pop()

    print(f"Verifying user: {current_user.title()}")
    confirmed_users.append(current_user)

# Display all confirmed users.
print("\nThe following users have been confirmed:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

Using a while Loop with Lists and Dictionaries

- **Moving Items from One List to Another – continued**

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Using a while Loop with Lists and Dictionaries

- **Removing All Instances of Specific Values from a List**
  - In Chapter 3 we used remove() to remove a specific value from a list. The remove() function worked because the value we were interested in appeared only once in the list. But what if you want to remove all instances of a value from a list?

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Using a while Loop with Lists and Dictionaries

- **Filling a Dictionary with User Input**
  - You can prompt for as much input as you need in each pass through a while loop.

```
responses = {}

# Set a flag to indicate that polling is active.
polling_active = True

while polling_active:
    # Prompt for the person's name and response.
    name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday? ")

    # Store the response in the dictionary.
    responses[name] = response

    # Find out if anyone else is going to take the poll.
    repeat = input("Would you like to let another person respond? (yes/ no) ")
    if repeat == 'no':
        polling_active = False

# Polling is complete. Show the results.
print("\n--- Poll Results ---")
for name, response in responses.items():
    print(f"{name} would like to climb {response}.")
```

Using a while Loop with Lists and Dictionaries

- **Filling a Dictionary with User Input – continued**

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.
```

Try It Yourself

- **7-8. Deli**: Make a list called sandwich_orders and fill it with the names of various sandwiches. Then make an empty list called finished_sandwiches. Loop through the list of sandwich orders and print a message for each order, such as I made your tuna sandwich. As each sandwich is made, move it to the list of finished sandwiches. After all the sandwiches have been made, print a message listing each sandwich that was made.
- **7-9. No Pastrami**: Using the list sandwich_orders from Exercise 7-8, make sure the sandwich 'pastrami' appears in the list at least three times. Add code near the beginning of your program to print a message saying the deli has run out of pastrami, and then use a while loop to remove all occurrences of 'pastrami' from sandwich_orders. Make sure no pastrami sandwiches end up in finished_sandwiches.
- **7-10. Dream Vacation**: Write a program that polls users about their dream vacation. Write a prompt similar to If you could visit one place in the world, where would you go? Include a block of code that prints the results of the poll.

Chapter 8: Functions

Introduction

- If you need to perform that task multiple times throughout your program, you don’t need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function.
- You’ll learn to store functions in separate files called modules to help organize your main program files.

Defining a Function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

- In this case, the name of the function is greet_user(), and it needs no information to do its job, so its parentheses are empty.
- Any indented lines that follow def greet_user(): make up the body of the function.
- The text at line 2 is a comment called a docstring, which describes what the function does.
- Docstrings are enclosed in triple quotes, which Python looks for when it generates documentation for the functions in your programs.
- To call a function, you write the name of the function, followed by any necessary information in parentheses. Because no information is needed here, calling our function is as simple as entering greet_user().

Passing Information to a Function

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

```
Hello, Jesse!
```

Arguments and Parameters

- The variable username in the definition of greet_user() is an example of a
- parameter, a piece of information the function needs to do its job.
- The value 'jesse' in greet_user('jesse') is an example of an argument.
- An argument is a piece of information that’s passed from a function call to a function.
- In this case the argument 'jesse' was passed to the function greet_user(), and the value was assigned to the parameter username.
- Note: People sometimes speak of arguments and parameters interchangeably. Don’t be surprised if you see the variables in a function definition referred to as arguments or the variables in a function call referred to as parameters.

Try It Yourself

- 8-1. Message: Write a function called display_message() that prints one sentence telling everyone what you are learning about in this chapter. Call the function, and make sure the message displays correctly.
- 8-2. Favorite Book: Write a function called favorite_book() that accepts one parameter, title. The function should print a message, such as One of my favorite books is Alice in Wonderland. Call the function, making sure to include a book title as an argument in the function call.

Passing Arguments

- Because a function definition can have multiple parameters, a function call may need multiple arguments.
- You can pass arguments to your functions in a number of ways; positional arguments, keyword arguments, and lists and dictionaries of values.

Passing Arguments

- **Positional Arguments**
  - When you call a function, Python must match each argument in the function call with a parameter in the function definition.
  - The simplest way to do this is based on the order of the arguments provided.

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
```

```
I have a hamster.
My hamster's name is Harry.
```

Passing Arguments

- **Multiple Function Calls**
  - You can call a function as many times as needed.

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Passing Arguments

- **Order Matters in Positional Arguments**
  - You can get unexpected results if you mix up the order of the arguments in a function call when using positional arguments.

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

```
I have a harry.
My harry's name is Hamster.
```

Passing Arguments

- **Keyword Arguments**
  - A keyword argument is a name-value pair that you pass to a function.
  - You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion.
- **Note**: When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

Passing Arguments

- **Default Values**
  - When writing a function, you can define a default value for each parameter.
  - If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value.
  - So when you define a default value for a parameter, you can exclude the corresponding argument you'd usually write in the function call.
  - For example, if you notice that most of the calls to describe_pet() are being used to describe dogs, you can set the default value of animal_type to 'dog'. Now anyone calling describe_pet() for a dog can omit that information.

Passing Arguments

- **Default Values – continued**

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

```
I have a dog.
My dog's name is Willie.
```

```
describe_pet('willie')
```

```
describe_pet(pet_name='harry', animal_type='hamster')
```

- **Note**: When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly.

Passing Arguments

- **Equivalent Function Calls**
  - Because positional arguments, keyword arguments, and default values can all be used together, often you'll have several equivalent ways to call a function.

```
def describe_pet(pet_name, animal_type='dog'):
```

```
# A dog named Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# A hamster named Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Passing Arguments

- **Note**: It doesn't really matter which calling style you use. As long as your function calls produce the output you want, just use the style you find easiest to understand.

Passing Arguments

- **Avoiding Argument Errors**
  - When you start to use functions, don't be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work.

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet()
```

```
Traceback (most recent call last):
  File "pets.py", line 6, in <module>
    describe_pet()
TypeError: describe_pet() missing 2 required positional arguments: 'animal_
type' and 'pet_name'
```

Try It Yourself

- **8-3. T-Shirt**: Write a function called make_shirt() that accepts a size and the text of a message that should be printed on the shirt. The function should print a sentence summarizing the size of the shirt and the message printed on it. Call the function once using positional arguments to make a shirt. Call the function a second time using keyword arguments.
- **8-4. Large Shirts**: Modify the make_shirt() function so that shirts are large by default with a message that reads I love Python. Make a large shirt and a medium shirt with the default message, and a shirt of any size with a different message.
- **8-5. Cities:** Write a function called describe_city() that accepts the name of a city and its country. The function should print a simple sentence, such as Reykjavik is in Iceland. Give the parameter for the country a default value. Call your function for three different cities, at least one of which is not in the default country.

Return Values

- A function doesn’t always have to display its output directly. Instead, it can process some data and then return a value or set of values. The value the function returns is called a return value.
- **Returning a Simple Value**

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

```
Jimi Hendrix
```

Return Values

- **Making an Argument Optional**
  - Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to. You can use default values to make an argument optional.

```
def get_formatted_name(first_name, middle_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

Return Values

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Return a full name, neatly formatted."""
    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

```
Jimi Hendrix
John Lee Hooker
```

Return Values

- **Returning a Dictionary**
  - A function can return any kind of value you need it to, including more complicated data structures like lists and dictionaries.

```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix'}
```

Return Values

- Returning a Dictionary – continued
  - You can easily extend this function to accept optional values like a middle name, an age, an occupation, or any other information you want to store about a person.

```
def build_person(first_name, last_name, age=None):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

Return Values

- **Using a Function with a while Loop**

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# This is an infinite loop!
while True:
    print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

Return Values

- **Using a Function with a while Loop – continued**
  - For previous example, we use a simple version of get_formatted_name() that doesn't involve middle names.
  - The while loop asks the user to enter their name, and we prompt for their first and last name separately.
  - But there's one problem with this while loop: We haven't defined a quit condition. Where do you put a quit condition when you ask for a series of inputs?
  - We want the user to be able to quit as easily as possible, so each prompt should offer a way to quit.

Return Values

- **Using a Function with a while Loop – continued**

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break
    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

Return Values

- **Using a Function with a while Loop – continued**

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes

Hello, Eric Matthes!

Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

Try It Yourself

- **8-6. City Names**: Write a function called city_country() that takes in the name of a city and its country. The function should return a string formatted like this:

  "Santiago, Chile"

- Call your function with at least three city-country pairs, and print the values that are returned.

Try It Yourself

- **8-7. Album**: Write a function called make_album() that builds a dictionary describing a music album. The function should take in an artist name and an album title, and it should return a dictionary containing these two pieces of information. Use the function to make three dictionaries representing different albums. Print each return value to show that the dictionaries are storing the album information correctly.
- Use None to add an optional parameter to make_album() that allows you to store the number of songs on an album. If the calling line includes a value for the number of songs, add that value to the album's dictionary. Make at least one new function call that includes the number of songs on an album.

Try It Yourself

- **8-8. User Albums**: Start with your program from Exercise 8-7. Write a while loop that allows users to enter an album's artist and title. Once you have that information, call make_album() with the user's input and print the dictionary that's created. Be sure to include a quit value in the while loop.

Passing a List

- You'll often find it useful to pass a list to a function, whether it's a list of names, numbers, or more complex objects, such as dictionaries. When you pass a list to a function, the function gets direct access to the contents of the list.

```
def greet_users(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

Passing a List

- **Modifying a List in a Function**
  - When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function's body are permanent, allowing you to work efficiently even when you're dealing with large amounts of data.

```
# Start with some designs that need to be printed.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simulate printing each design, until none are left.
#  Move each design to completed_models after printing.
while unprinted_designs:
    current_design = unprinted_designs.pop()

    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

# Display all completed models.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

Passing a List

- **Modifying a List in a Function – continued**

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

- We can reorganize this code by writing two functions, each of which does one specific job. Most of the code won't change; we're just making it more carefully structured.

Passing a List

- **Modifying a List in a Function – continued**

```
def print_models(unprinted_designs, completed_models):
    """
    Simulate printing each design, until none are left.
    Move each design to completed_models after printing.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

def show_completed_models(completed_models):
    """Show all the models that were printed."""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Passing a List

- **Modifying a List in a Function – continued**
  - This program has the same output as the version without functions, but the code is much more organized. The code that does most of the work has been moved to two separate functions, which makes the main part of the program easier to understand.

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Passing a List

- **Preventing a Function from Modifying a List**
  - Sometimes you'll want to prevent a function from modifying a list.
  - For example, say that you start with a list of unprinted designs and write a function to move them to a list of completed models, as in the previous example.
  - You may decide that even though you've printed all the designs, you want to keep the original list of unprinted designs for your records.
  - But because you moved all the design names out of unprinted_designs, the list is now empty, and the empty list is the only version you have; the original is gone.
  - In this case, you can address this issue by passing the function a copy of the list, not the original.
  - Any changes the function makes to the list will affect only the copy, leaving the original list intact.

Passing a List

- **Preventing a Function from Modifying a List – continued**
  - You can send a copy of a list to a function like this:

```
function_name(list_name[:])
```

  - The slice notation [:] makes a copy of the list to send to the function.
  - If we didn't want to empty the list of unprinted designs in printing_models.py, we could call print_models() like this:

```
print_models(unprinted_designs[:], completed_models)
```

Try It Yourself

- **8-9. Messages**: Make a list containing a series of short text messages. Pass the list to a function called show_messages(), which prints each text message.
- **8-10. Sending Messages**: Start with a copy of your program from Exercise 8-9. Write a function called send_messages() that prints each text message and moves each message to a new list called sent_messages as it's printed. After calling the function, print both of your lists to make sure the messages were moved correctly.
- **8-11. Archived Messages**: Start with your work from Exercise 8-10. Call the function send_messages() with a copy of the list of messages. After calling the function, print both of your lists to show that the original list has retained its messages.

Passing an Arbitrary Number of Arguments

- Sometimes you won't know ahead of time how many arguments a function needs to accept. Fortunately, Python allows a function to collect an arbitrary number of arguments from the calling statement.
- For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want.

Passing an Arbitrary Number of Arguments

```
def make_pizza(*toppings):
    """Print the list of toppings that have been requested."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Passing an Arbitrary Number of Arguments

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Passing an Arbitrary Number of Arguments

- **Mixing Positional and Arbitrary Arguments**
  - If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments **must** be placed last in the function definition. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter.

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Passing an Arbitrary Number of Arguments

- **Using Arbitrary Keyword Arguments**
  - Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides.

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

Passing an Arbitrary Number of Arguments

- **Using Arbitrary Keyword Arguments – continued**
  - The definition of build_profile() expects a first and last name, and then it allows the user to pass in as many name-value pairs as they want.
  - The double asterisks before the parameter **user_info cause Python to create an empty dictionary called user_info and pack whatever name-value pairs it receives into this dictionary.
  - Within the function, you can access the key-value pairs in user_info just as you would for any dictionary.

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

Passing an Arbitrary Number of Arguments

- **Using Arbitrary Keyword Arguments – continued**
  - **Note**: You'll often see the generic parameter name *args, which collects arbitrary positional arguments like this.
  - **Note**: You'll often see the parameter name **kwargs used to collect non-specific keyword arguments.

Try It Yourself

- **8-12. Sandwiches**: Write a function that accepts a list of items a person wants on a sandwich. The function should have one parameter that collects as many items as the function call provides, and it should print a summary of the sandwich that's being ordered. Call the function three times, using a different number of arguments each time.
- **8-13. User Profile**: Start with a copy of user_profile.py from page 149. Build a profile of yourself by calling build_profile(), using your first and last names and three other key-value pairs that describe you.

Try It Yourself

- **8-14. Cars:** Write a function that stores information about a car in a dictionary. The function should always receive a manufacturer and a model name. It should then accept an arbitrary number of keyword arguments. Call the function with the required information and two other name-value pairs, such as a color or an optional feature. Your function should work for a call like this one:

  car = make_car('subaru', 'outback', color='blue', tow_package=True)

- Print the dictionary that's returned to make sure all the information was stored correctly.

Storing Your Functions in Modules

- One advantage of functions is the way they separate blocks of code from your main program.
- By using descriptive names for your functions, your main program will be much easier to follow.
- You can go a step further by storing your functions in a separate file called a module and then importing that module into your main program.
- An import statement tells Python to make the code in a module available in the currently running program file.
- When you store your functions in separate files, you can share those files with other programmers without having to share your entire program.
- Knowing how to import functions also allows you to use libraries of functions that other programmers have written.

Storing Your Functions in Modules

- **Importing an Entire Module**
  - To start importing functions, we first need to create a module. A module is a file ending in .py that contains the code you want to import into your Functions program.
  - Let's make a module that contains the function make_pizza().
  - To make this module, we'll remove everything from the file pizza.py except the function make_pizza().

Storing Your Functions in Modules

- **Importing an Entire Module – continued**

pizza.py

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

making_pizzas.py

```
import pizza

❶ pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

- When Python reads this file, the line import pizza tells Python to open the file pizza.py and copy all the functions from it into this program. You don't actually see code being copied between files because Python copies the code behind the scenes just before the program runs. All you need to know is that any function defined in pizza.py will now be available in making_pizzas.py.

Storing Your Functions in Modules

- **Importing an Entire Module – continued**

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Storing Your Functions in Modules

- **Importing an Entire Module – continued**
  - This first approach to importing, in which you simply write import followed by the name of the module, makes every function from the module available in your program. If you use this kind of import statement to import an entire module named module_name.py, each function in the module is available through the following syntax:

```
module_name.function_name()
```

Storing Your Functions in Modules

- **Importing Specific Functions**
  - You can also import a specific function from a module.

```
from module_name import function_name
```

```
from module_name import function_0, function_1, function_2
```

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Storing Your Functions in Modules

- **Importing Specific Functions – continued**
  - With previous syntax, you don’t need to use the dot notation when you call a function. Because we’ve explicitly imported the function make_pizza() in the import statement, we can call it by name when we use the function.

Storing Your Functions in Modules

- **Using as to Give a Function an Alias**
  - If the name of a function you're importing might conflict with an existing name in your program or if the function name is long, you can use a short, unique alias—an alternate name similar to a nickname for the function.
  - You'll give the function this special nickname when you import the function. Here we give the function make_pizza() an alias, mp(), by importing make_pizza as mp.

Storing Your Functions in Modules

- Using as to Give a Function an Alias – continued

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

- The general syntax for providing an alias is:

```
from module_name import function_name as fn
```

Storing Your Functions in Modules

- **Using as to Give a Module an Alias**
  - You can also provide an alias for a module name. Giving a module a short alias, like p for pizza, allows you to call the module's functions more quickly.
  - Calling p.make_pizza() is more concise than calling pizza.make_pizza():

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
import module_name as mn
```

Storing Your Functions in Modules

- **Importing All Functions in a Module**
  - You can tell Python to import every function in a module by using the asterisk (*) operator:

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Storing Your Functions in Modules

- **Importing All Functions in a Module – continued**
  - The asterisk in the import statement tells Python to copy every function from the module pizza into this program file. Because every function is imported, you can call each function by name without using the dot notation. However, it's best not to use this approach when you're working with larger modules that you didn't write: if the module has a function name that matches an existing name in your project, you can get some unexpected results.
  - The best approach is to import the function or functions you want, or import the entire module and use the dot notation.

```
from module_name import *
```

Styling Functions

- Functions should have descriptive names, and these names should use lowercase letters and underscores.
- Descriptive names help you and others understand what your code is trying to do.
- Module names should use these conventions as well.
- Every function should have a comment that explains concisely what the function does.
- This comment should appear immediately after the function definition and use the docstring format.
- In a well-documented function, other programmers can use the function by reading only the description in the docstring. They should be able to trust that the code works as described, and as long as they know the name of the function, the arguments it needs, and the kind of value it returns, they should be able to use it in their programs.

Styling Functions

- If you specify a default value for a parameter, no spaces should be used on either side of the equal sign:

```
def function_name(parameter_0, parameter_1='default value')
```

- The same convention should be used for keyword arguments in function calls:

```
function_name(value_0, parameter_1='value')
```

Styling Functions

- PEP 8 (https://www.python.org/dev/peps/pep-0008/) recommends that you limit lines of code to 79 characters so every line is visible in a reasonably sized editor window.
- If a set of parameters causes a function's definition to be longer than 79 characters, press enter after the opening parenthesis on the definition line. On the next line, press tab twice to separate the list of arguments from the body of the function, which will only be indented one level.

Styling Functions

- Most editors automatically line up any additional lines of parameters to match the indentation you have established on the first line:

```
def function_name(
        parameter_0, parameter_1, parameter_2,
        parameter_3, parameter_4, parameter_5):
    function body...
```

- If your program or module has more than one function, you can separate each by two blank lines to make it easier to see where one function ends and the next one begins.
- All import statements should be written at the beginning of a file. The only exception is if you use comments at the beginning of your file to describe the overall program.

Try It Yourself

- **8-15. Printing Models**: Put the functions for the example printing_models.py in a separate file called printing_functions.py. Write an import statement at the top of printing_models.py, and modify the file to use the imported functions.

Try It Yourself

- **8-16. Imports**: Using a program you wrote that has one function in it, store that function in a separate file. Import the function into your main program file, and call the function using each of these approaches:

```
import module_name
from module_name import function_name
from module_name import function_name as fn
import module_name as mn
from module_name import *
```

Try It Yourself

- **8-17. Styling Functions**: Choose any three programs you wrote for this chapter, and make sure they follow the styling guidelines described in this section.

Chapter 9: Classes

INTRODUCTION

- Object-oriented programming is one of the most effective approaches to writing software. In object-oriented programming you write classes that represent real-world things and situations, and you create objects based on these classes. When you write a class, you define the general behavior that a whole category of objects can have.
- Making an object from a class is called instantiation, and you work with instances of a class. In this chapter you'll write classes and create instances of those classes. You'll specify the kind of information that can be stored in instances, and you'll define actions that can be taken with these instances.

Creating and Using a Class

- You can model almost anything using classes.
- Let’s start by writing a simple class, Dog, that represents a dog—not one dog in particular, but any dog.
- What do we know about most pet dogs? Well, they all have a name and age. We also know that most dogs sit and roll over.
- Those two pieces of information (name and age) and those two behaviors (sit and roll over) will go in our Dog class because they’re common to most dogs.
- This class will tell Python how to make an object representing a dog. After our class is written, we’ll use it to make individual instances, each of which represents one specific dog.

Creating and Using a Class

- Creating the Dog Class

```
class Dog:
    """A simple attempt to model a dog."""

    def __init__(self, name, age):
        """Initialize name and age attributes."""
        self.name = name
        self.age = age

    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        """Simulate rolling over in response to a command."""
        print(f"{self.name} rolled over!")
```

Creating and Using a Class

- The __init__() Method
  - A function that's part of a class is a method. Everything you learned about functions applies to methods as well; the only practical difference for now is the way we'll call methods.
  - The __init__() method at w is a special method that Python runs automatically whenever we create a new instance based on the Dog class.
  - This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names.
  - Make sure to use two underscores on each side of __init__(). If you use just one on each side, the method won't be called automatically when you use your class, which can result in errors that are difficult to identify.

Creating and Using a Class

- **Making an Instance from a Class**
  - Think of a class as a set of instructions for how to make an instance. The class Dog is a set of instructions that tells Python how to make individual instances representing specific dogs.

```
class Dog:
    --snip--

my_dog = Dog('Willie', 6)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
```

Creating and Using a Class

- Accessing Attributes
  - To access the attributes of an instance, you use dot notation.

```
my_dog.name
```

```
My dog's name is Willie.
My dog is 6 years old.
```

Creating and Using a Class

- **Calling Methods**
  - After we create an instance from the class Dog, we can use dot notation to call any method defined in Dog.

```
class Dog:
    --snip--

my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

```
Willie is now sitting.
Willie rolled over!
```

Creating and Using a Class

- **Creating Multiple Instances**
  - You can create as many instances from a class as you need.

```
class Dog:
    --snip--

my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

Creating and Using a Class

- **Creating Multiple Instances – continued**
  - You can make as many instances from one class as you need, as long as you give each instance a unique variable name or it occupies a unique spot in a list or dictionary.

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Try It Yourself

- **9-1. Restaurant**: Make a class called Restaurant. The __init__() method for Restaurant should store two attributes: a restaurant_name and a cuisine_type. Make a method called describe_restaurant() that prints these two pieces of information, and a method called open_restaurant() that prints a message indicating that the restaurant is open. Make an instance called restaurant from your class. Print the two attributes individually, and then call both methods.
- **9-2. Three Restaurants**: Start with your class from Exercise 9-1. Create three different instances from the class, and call describe_restaurant() for each instance.

Try It Yourself

- **9-3. Users:** Make a class called User. Create two attributes called first_name and last_name, and then create several other attributes that are typically stored in a user profile. Make a method called describe_user() that prints a summary of the user's information. Make another method called greet_user() that prints a personalized greeting to the user. Create several instances representing different users, and call both methods for each user.

Working with Classes and Instances

- You can use classes to represent many real-world situations. Once you write a class, you'll spend most of your time working with instances created from that class. One of the first tasks you'll want to do is modify the attributes associated with a particular instance.

Working with Classes and Instances

- The Car Class
  - Let's write a new class representing a car. Our class will store information about the kind of car we're working with, and it will have a method that summarizes this information.

```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
```

```
2019 Audi A4
```

Working with Classes and Instances

- **Setting a Default Value for an Attribute**
  - When an instance is created, attributes can be defined without being passed in as parameters.
  - These attributes can be defined in the __init__() method, where they are assigned a default value. Let's add an attribute called odometer_reading that always starts with a value of 0.

Working with Classes and Instances

- Setting a Default Value for an Attribute – continued

```
class Car:

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        --snip--

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

```
2019 Audi A4
This car has 0 miles on it.
```

Working with Classes and Instances

- **Modifying Attribute Values**
  - You can change an attribute's value in three ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method.

Working with Classes and Instances

- **Modifying an Attribute's Value Directly**
  - The simplest way to modify the value of an attribute is to access the attribute directly through an instance.

```
class Car:
    --snip--

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
2019 Audi A4
This car has 23 miles on it.
```

Working with Classes and Instances

- **Modifying an Attribute's Value Through a Method**
  - It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

```
2019 Audi A4
This car has 23 miles on it.
```

Working with Classes and Instances

- **Modifying an Attribute's Value Through a Method – continued**
  - We can extend the method update_odometer() to do additional work every time the odometer reading is modified. Let's add a little logic to make sure no one tries to roll back the odometer reading

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
```

Working with Classes and Instances

- **Incrementing an Attribute's Value Through a Method**
  - Sometimes you'll want to increment an attribute's value by a certain amount rather than set an entirely new value. Say we buy a used car and put 100 miles on it between the time we buy it and the time we register it.

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        --snip--

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

my_used_car = Car('subaru', 'outback', 2015)
print(my_used_car.get_descriptive_name())

my_used_car.update_odometer(23_500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

Working with Classes and Instances

- Incrementing an Attribute's Value Through a Method – continued

```
2015 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

- **Note**: You can use methods like this to control how users of your program update values such as an odometer reading, but anyone with access to the program can set the odometer reading to any value by accessing the attribute directly. Effective security takes extreme attention to detail in addition to basic checks like those shown here.

Try It Yourself

- **9-4. Number Served**: Start with your program from Exercise 9-1 (page 162). Add an attribute called number_served with a default value of 0. Create an instance called restaurant from this class. Print the number of customers the restaurant has served, and then change this value and print it again. Add a method called set_number_served() that lets you set the number of customers that have been served. Call this method with a new number and print the value again. Add a method called increment_number_served() that lets you increment the number of customers who've been served. Call this method with any number you like that could represent how many customers were served in, say, a day of business.

Try It Yourself

- **9-5. Login Attempts**: Add an attribute called login_attempts to your User class from Exercise 9-3 (page 162). Write a method called increment_login _attempts() that increments the value of login_attempts by 1. Write another method called reset_login_attempts() that resets the value of login_attempts to 0. Make an instance of the User class and call increment_login_attempts() several times. Print the value of login_attempts to make sure it was incremented properly, and then call reset_login_attempts(). Print login_attempts again to make sure it was reset to 0.

Inheritance

- You don’t always have to start from scratch when writing a class. If the class you’re writing is a specialized version of another class you wrote, you can use inheritance. When one class inherits from another, it takes on the attributes and methods of the first class. The original class is called the parent class, and the new class is the child class. The child class can inherit any or all of the attributes and methods of its parent class, but it’s also free to define new attributes and methods of its own.

Inheritance

- **The __init__() Method for a Child Class**
  - When you're writing a new class based on an existing class, you'll often want to call the __init__() method from the parent class. This will initialize any attributes that were defined in the parent __init__() method and make them available in the child class.

Inheritance

- The __init__() Method for a Child Class - continued

```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

class ElectricCar(Car):
```

Inheritance

- **The __init__() Method for a Child Class - continued**
  - The super() function is a special function that allows you to call a method from the parent class. This line tells Python to call the __init__() method from Car, which gives an ElectricCar instance all the attributes defined in that method. The name super comes from a convention of calling the parent class a superclass and the child class a subclass

```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
        super().__init__(make, model, year)


my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
```

```
2019 Tesla Model S
```

Inheritance

- **Defining Attributes and Methods for the Child Class**
  - Once you have a child class that inherits from a parent class, you can add any new attributes and methods necessary to differentiate the child class from the parent class.

```
class Car:
    --snip--

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery_size = 75

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Inheritance

- **Overriding Methods from the Parent Class**
  - You can override any method from the parent class that doesn't fit what you're trying to model with the child class.
  - To do this, you define a method in the child class with the same name as the method you want to override in the parent class.
  - Python will disregard the parent class method and only pay attention to the method you define in the child class.

Inheritance

- Overriding Methods from the Parent Class – continued

```
class ElectricCar(Car):
    --snip--

    def fill_gas_tank(self):
        """Electric cars don't have gas tanks."""
        print("This car doesn't need a gas tank!")
```

Inheritance

- **Instances as Attributes**
  - When modeling something from the real world in code, you may find that you're adding more and more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy.
  - In these situations, you might recognize that part of one class can be written as a separate class. You can break your large class into smaller classes that work together.

Inheritance

- Instances as Attributes – continued

```
class Car:
    --snip--

class Battery:
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=75):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")


class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

```
my_tesla.battery.describe_battery()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Inheritance

- Instances as Attributes – continued

```
class Car:
    --snip--

class Battery:
    --snip--

    def get_range(self):
        """Print a statement about the range this battery provides."""
       if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    --snip--

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

Inheritance

- Modeling Real-World Objects
  - As you begin to model more complicated things like electric cars, you'll wrestle with interesting questions. Is the range of an electric car a property of the battery or of the car? If we're only describing one car, it's probably fine to maintain the association of the method get_range() with the Battery class.
  - But if we're describing a manufacturer's entire line of cars, we probably want to move get_range() to the ElectricCar class. The get_range() method would still check the battery size before determining the range, but it would report a range specific to the kind of car it's associated with. Alternatively, we could maintain the association of the get_range() method with the battery but pass it a parameter such as car_model. The get_range() method would then report a range based on the battery size and car model.
  - This brings you to an interesting point in your growth as a programmer. When you wrestle with questions like these, you're thinking at a higher logical level rather than a syntax-focused level. You're thinking not about Python, but about how to represent the real world in code. When you reach this point, you'll realize there are often no right or wrong approaches to modeling real-world situations.
  - Some approaches are more efficient than others, but it takes practice to find the most efficient representations. If your code is working as you want it to, you're doing well!

Try It Yourself

- **9-6. Ice Cream Stand**: An ice cream stand is a specific kind of restaurant. Write a class called IceCreamStand that inherits from the Restaurant class you wrote in Exercise 9-1 (page 162) or Exercise 9-4 (page 167). Either version of the class will work; just pick the one you like better. Add an attribute called flavors that stores a list of ice cream flavors. Write a method that displays these flavors. Create an instance of IceCreamStand, and call this method.

Try It Yourself

- **9-7. Admin**: An administrator is a special kind of user. Write a class called Admin that inherits from the User class you wrote in Exercise 9-3 (page 162) or Exercise 9-5 (page 167). Add an attribute, privileges, that stores a list of strings like "can add post", "can delete post", "can ban user", and so on. Write a method called show_privileges() that lists the administrator's set of privileges. Create an instance of Admin, and call your method.
- **9-8. Privileges**: Write a separate Privileges class. The class should have one attribute, privileges, that stores a list of strings as described in Exercise 9-7. Move the show_privileges() method to this class. Make a Privileges instance as an attribute in the Admin class. Create a new instance of Admin and use your method to show its privileges.

Try It Yourself

- **9-9. Battery Upgrade**: Use the final version of electric_car.py from this section. Add a method to the Battery class called upgrade_battery(). This method should check the battery size and set the capacity to 100 if it isn't already. Make an electric car with a default battery size, call get_range() once, and then call get_range() a second time after upgrading the battery. You should see an increase in the car's range.

Importing Classes

- As you add more functionality to your classes, your files can get long, even when you use inheritance properly. In keeping with the overall philosophy of Python, you'll want to keep your files as uncluttered as possible. To help, Python lets you store classes in modules and then import the classes you need into your main program.

Importing Classes

- Importing a Single Class
  - Let's create a module containing just the Car class.

```
"""A class that can be used to represent a car."""

class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")
```

Importing Classes

- Importing a Single Class – continued

```
def update_odometer(self, mileage):
    """
    Set the odometer reading to the given value.
    Reject the change if it attempts to roll the odometer back.
    """
    if mileage >= self.odometer_reading:
        self.odometer_reading = mileage
    else:
        print("You can't roll back an odometer!")

def increment_odometer(self, miles):
    """Add the given amount to the odometer reading."""
    self.odometer_reading += miles
```

Importing Classes

- Importing a Single Class – continued

```
from car import Car

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
2019 Audi A4
This car has 23 miles on it.
```

Importing Classes

- **Storing Multiple Classes in a Module**
  - You can store as many classes as you need in a single module, although each class in a module should be related somehow.

```
"""A set of classes used to represent gas and electric cars."""

class Car:
    --snip--

class Battery:
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Models aspects of a car, specific to electric vehicles."""
```

Importing Classes

- Storing Multiple Classes in a Module – continued

```
    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

Importing Classes

- Importing Multiple Classes from a Module
  - You can import as many classes as you need into a program file.

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Importing Classes

- **Importing an Entire Module**
  - You can also import an entire module and then access the classes you need using dot notation. This approach is simple and results in code that is easy to read. Because every call that creates an instance of a class includes the module name, you won't have naming conflicts with any names used in the current file.

```
import car

my_beetle = car.Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())

my_tesla = car.ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

Importing Classes

- **Importing All Classes from a Module**
  - You can import every class from a module using the following syntax:

```
from module_name import *
```

  - This method is not recommended for two reasons.
  - First, it's helpful to be able to read the import statements at the top of a file and get a clear sense of which classes a program uses. With this approach it's unclear which classes you're using from the module. This approach can also lead to confusion with names in the file. If you accidentally import a class with the same name as something else in your program file, you can create errors that are hard to diagnose.
  - If you need to import many classes from a module, you're better off importing the entire module and using the module_name.ClassName syntax.

Importing Classes

- Importing a Module into a Module
  - Sometimes you'll want to spread out your classes over several modules to keep any one file from growing too large and avoid storing unrelated classes in the same module. When you store your classes in several modules, you may find that a class in one module depends on a class in another module. When this happens, you can import the required class into the first module.

```
"""A set of classes that can be used to represent electric cars."""

from car import Car

class Battery:
    --snip--

class ElectricCar(Car):
    --snip--
```

Importing Classes

- Importing a Module into a Module – continued

```
"""A class that can be used to represent a car."""

class Car:
    --snip--
```

```
from car import Car
from electric_car import ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Importing Classes

- Using Aliases
  - As you saw in Chapter 8, aliases can be quite helpful when using modules to organize your projects' code. You can use aliases when importing classes as well.
  - As an example, consider a program where you want to make a bunch of electric cars. It might get tedious to type (and read) ElectricCar over and over again. You can give ElectricCar an alias in the import statement:

```
from electric_car import ElectricCar as EC
```

```
my_tesla = EC('tesla', 'roadster', 2019)
```

Importing Classes

- **Finding Your Own Workflow**
  - As you can see, Python gives you many options for how to structure code in a large project. It's important to know all these possibilities so you can determine the best ways to organize your projects as well as understand other people's projects. When you're starting out, keep your code structure simple.
  - Try doing everything in one file and moving your classes to separate modules once everything is working. If you like how modules and files interact, try storing your classes in modules when you start a project. Find an approach that lets you write code that works, and go from there.

Try It Yourself

- **9-10. Imported Restaurant**: Using your latest Restaurant class, store it in a module. Make a separate file that imports Restaurant. Make a Restaurant instance, and call one of Restaurant's methods to show that the import statement is working properly.
- **9-11. Imported Admin**: Start with your work from Exercise 9-8 (page 173). Store the classes User, Privileges, and Admin in one module. Create a separate file, make an Admin instance, and call show_privileges() to show that everything is working correctly.
- **9-12. Multiple Modules**: Store the User class in one module, and store the Privileges and Admin classes in a separate module. In a separate file, create an Admin instance and call show_privileges() to show that everything is still working correctly.

The Python Standard Library

- The Python standard library is a set of modules included with every Python installation.
- Now that you have a basic understanding of how functions and classes work, you can start to use modules like these that other programmers have written.
- You can use any function or class in the standard library by including a simple import statement at the top of your file.
- Let’s look at one module, random, which can be useful in modeling many real-world situations.
- One interesting function from the random module is randint(). This function takes two integer arguments and returns a randomly selected integer between (and including) those numbers.

The Python Standard Library

- Here's how to generate a random number between 1 and 6:

```
>>> from random import randint
>>> randint(1, 6)
3
```

- Another useful function is choice(). This function takes in a list or tuple and returns a randomly chosen element:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

Try It Yourself

- **9-13. Dice**: Make a class Die with one attribute called sides, which has a default value of 6. Write a method called roll_die() that prints a random number between 1 and the number of sides the die has. Make a 6-sided die and roll it 10 times. Make a 10-sided die and a 20-sided die. Roll each die 10 times.
- **9-14. Lottery**: Make a list or tuple containing a series of 10 numbers and five letters. Randomly select four numbers or letters from the list and print a message saying that any ticket matching these four numbers or letters wins a prize.

Try It Yourself

- **9-15. Lottery Analysis**: You can use a loop to see how hard it might be to win the kind of lottery you just modeled. Make a list or tuple called my_ticket. Write a loop that keeps pulling numbers until your ticket wins. Print a message reporting how many times the loop had to run to give you a winning ticket.
- **9-16. Python Module of the Week**: One excellent resource for exploring the Python standard library is a site called Python Module of the Week. Go to https://pymotw.com/ and look at the table of contents. Find a module that looks interesting to you and read about it, perhaps starting with the random module.

Styling Classes

- Class names should be written in CamelCase. To do this, capitalize the first letter of each word in the name, and don’t use underscores.
- Instance and module names should be written in lowercase with underscores between words.
- Every class should have a docstring immediately following the class definition. The docstring should be a brief description of what the class does, and you should follow the same formatting conventions you used for writing docstrings in functions.
- Each module should also have a docstring describing what the classes in a module can be used for. You can use blank lines to organize code, but don’t use them excessively.
- Within a class you can use one blank line between methods, and within a module you can use two blank lines to separate classes.
- If you need to import a module from the standard library and a module that you wrote, place the import statement for the standard library module first. Then add a blank line and the import statement for the module you wrote. In programs with multiple import statements, this convention makes it easier to see where the different modules used in the program come from.

Chapter 10: Files and Exceptions


An-Najah National University
1918 AC
1337 Hijri