Introduction to

Python Programming

Introduction to Python Programming

SENIOR CONTRIBUTING AUTHORS
UDAYAN DAS, SAINT MARY'S COLLEGE OF CALIFORNIA
AUBREY LAWSON, WILEY
CHRIS MAYFIELD, JAMES MADISON UNIVERSITY
NARGES NOROUZI, UC BERKELEY

OpenStax
Rice University
6100 Main Street MS-375
Houston, Texas 77005

To learn more about OpenStax, visit https://openstax.org.
Individual print copies and bulk orders can be purchased through our website.

DIGITAL VERSION ISBN-13 **978-1-961584-45-7**
ORIGINAL PUBLICATION YEAR **2024**
1 2 3 4 5 6 7 8 9 10 CJP 24

OpenStax

OpenStax provides free, peer-reviewed, openly licensed textbooks for introductory college and Advanced Placement® courses and low-cost, personalized courseware that helps students learn. A nonprofit ed tech initiative based at Rice University, we're committed to helping students access the tools they need to complete their courses and meet their educational goals.

Rice University

OpenStax is an initiative of Rice University. As a leading research university with a distinctive commitment to undergraduate education, Rice University aspires to path-breaking research, unsurpassed teaching, and contributions to the betterment of our world. It seeks to fulfill this mission by cultivating a diverse community of learning and discovery that produces leaders across the spectrum of human endeavor.

Philanthropic Support

OpenStax is grateful for the generous philanthropic partners who advance our mission to improve educational access and learning for everyone. To see the impact of our supporter community and our most updated list of partners, please visit openstax.org/foundation.

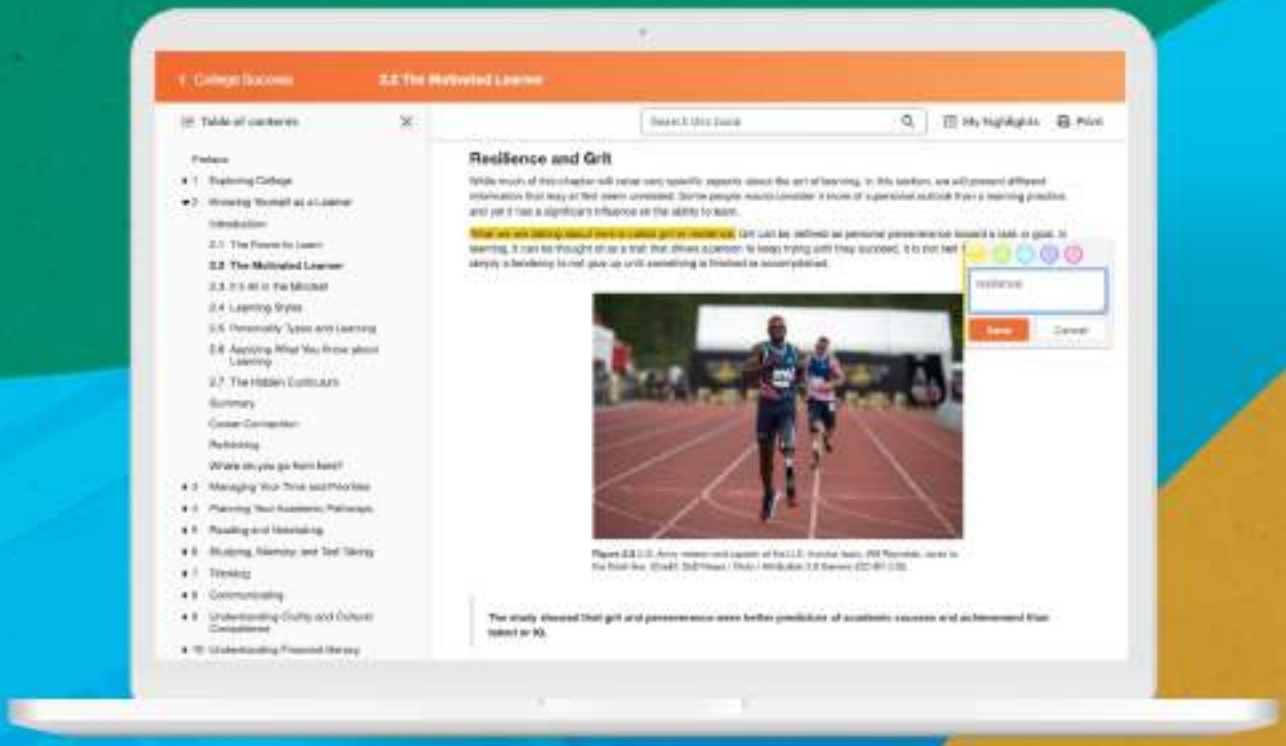
Arnold Ventures
Chan Zuckerberg Initiative
Chegg, Inc.
Arthur and Carlyse Ciocca Charitable Foundation
Digital Promise
Ann and John Doerr
Bill & Melinda Gates Foundation
Girard Foundation
Google Inc.
The William and Flora Hewlett Foundation
The Hewlett-Packard Company
Intel Inc.
Rusty and John Jaggers
The Calvin K. Kazanjian Economics Foundation
Charles Koch Foundation
Leon Lowenstein Foundation, Inc.
The Maxfield Foundation
Burt and Deedee McMurtry
Michelson 20MM Foundation
National Science Foundation
The Open Society Foundations
Jumee Yhu and David E. Park III
Brian D. Patterson USA-International Foundation
The Bill and Stephanie Sick Fund
Steven L. Smith & Diana T. Go
Stand Together
Robin and Sandy Stuart Foundation
The Stuart Family Foundation
Tammy and Guillermo Treviño
Valhalla Charitable Foundation
White Star Education Foundation
Schmidt Futures
William Marsh Rice University

openstax™

Study where you want, what you want, when you want.

When you access your book in our web view, you can use our new online highlighting and note-taking features to create your own study guides.

Our books are free and flexible, forever.
Get started at openstax.org/details/books/introduction-python-programming



Access. The future of education.
openstax.org

RICE

CONTENTS

Preface

About OpenStax

OpenStax is part of Rice University, which is a 501(c)(3) nonprofit charitable corporation. As an educational initiative, it's our mission to improve educational access and learning for everyone. Through our partnerships with philanthropic organizations and our alliance with other educational resource companies, we're breaking down the most common barriers to learning. Because we believe that everyone should and can have access to knowledge.

About OpenStax Resources

Customization

Introduction to Python Programming is licensed under a Creative Commons Attribution 4.0 International (CC BY) license, which means that you can distribute, remix, and build upon the content, as long as you provide attribution to OpenStax and its content contributors.

Because our books are openly licensed, you are free to use the entire book or select only the sections that are most relevant to the needs of your course. Feel free to remix the content by assigning your students certain chapters and sections in your syllabus, in the order that you prefer. You can even provide a direct link in your syllabus to the sections in the web view of your book.

Instructors also have the option of creating a customized version of their OpenStax book. Visit the Instructor Resources section of your book page on OpenStax.org for more information.

Art attribution

In *Introduction to Python Programming*, most photos and third-party illustrations contain attribution to their creator, rights holder, host platform, and/or license within the caption. Because the art is openly licensed, anyone may reuse the art as long as they provide the same attribution to its original source. To maximize readability and content flow, some art does not include attribution in the text. This art is part of the public domain or under a CC0 or similar license, and can be reused without attribution. For illustrations (e.g. graphs, charts, etc.) that are not credited, use the following attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license.

Errata

All OpenStax textbooks undergo a rigorous review process. However, like any professional-grade textbook, errors sometimes occur. Since our books are web-based, we can make updates periodically when deemed pedagogically necessary. If you have a correction to suggest, submit it through the link on your book page on OpenStax.org. Subject matter experts review all errata suggestions. OpenStax is committed to remaining transparent about all updates, so you will also find a list of past and pending errata changes on your book page on OpenStax.org.

Format

You can access this textbook for free in web view or PDF through OpenStax.org. The web view is the recommended format because it is the most accessible—including being WCAG 2.1 AA compliant—and most current.

About *Introduction to Python Programming*

Introduction to Python Programming provides a comprehensive foundation in programming concepts and skills, and is aligned to the scope of most introductory courses. A wide array of scenarios, contexts, and problems reflect programming applications in many disciplines and careers. The offering is suitable for a diverse learner audience, including those pursuing computer science, business, science, social science, statistics, data science, and related areas of study and employment.

Introduction to Python Programming is an interactive offering that teaches basic programming concepts, problem-solving skills, and the Python language using hands-on activities. The resource includes a unique, integrated code runner, through which students can immediately apply what they learn to check their understanding. Embedded videos, critical thinking exercises, and explorations of external programming tools and activities all contribute to a meaningful and supportive learning experience.

The content is organized in chapters, with each chapter containing 6-8 sections. Each section follows the pattern:

- Learning objectives
- 1–3 subsections
- Programming practice

The learning objectives are designed to help readers identify the section's focus. Each objective completes the sentence, "By the end of this section you should be able to". The programming practice aligns with the learning objectives and gives readers an opportunity to apply concepts learned in the section.

Pedagogical Foundation

Concise text and video-based animations

Introduction to Python Programming is designed to foster active learning and student engagement. It focuses on interactivity and practice through its integrated code runner, videos, and links to external environments and activities. With that focus, the material is often more concise, with less text and more activity.

Each section's content is organized in subsections. The subsection begins with a concise introduction to the concept, with key term definitions and brief context for its relevance and importance. The concept is then explained in detail using video-based animations and interactive learning questions.

Animation videos use a step-by-step approach to show the execution of Python code. Explanations for each step describe Python syntax, show how fundamental programming concepts are used, illustrate how variables are assigned, emphasize how code executes line by line, apply problem solving to create programs, and more.

CHECKPOINT

Displaying output to the user

Access multimedia content (https://www.openstax.org/r/displaying-output)

Learning questions

After introducing a new concept and explaining the concept with a video-based animation, each subsection includes engagement in the form of learning questions. These questions reinforce the concepts taught, explain concepts in more depth, directly address misconceptions and errors commonly made by new programmers, and teach related concepts.

Explanations are provided for the incorrect answers. Incorrect answers' explanations include why the answer is incorrect and help guide the reader to the correct answer.

Incorrect answer choices typically represent a misconception or are the result of making a common mistake. Even if the correct answer is achieved, readers are encouraged to explore the explanations to gain awareness of these common misconceptions.

Programming practice exercises

Each section ends with 1 or 2 practice programs. This book includes an integrated programming environment, known as the "OpenStax Python Code Runner," which allows readers to write programs directly in the browser.

The code runner requires the reader to pre-enter any input before running a program.

A sample code runner

Conventions used in this book

The following typographical conventions are used throughout the book:

Bold
Indicates vocabulary words when first defined in the chapter.

Italic
Indicates emphasized text, filenames, and file extensions.

`Constant width`
Used for code listings and code elements within paragraphs. Code elements include variable names, Python keywords, etc.

`Constant width bold`
Shows commands or keyboard input that should be typed literally by the user.

Ex:
Abbreviation for "Example:"

About the Authors

Senior Contributing Authors

Senior contributing authors, left to right: Udayan Das, Aubrey Lawson, Chris Mayfield, and Narges Norouzi.

Udayan Das, Saint Mary's College of California

Udayan Das, PhD, is an Associate Professor and Program Director of Computer Science at Saint Mary's College of California. He received his PhD in Computer Science and a master's in Computer Engineering from the Illinois Institute of Technology. His research interests include wireless networks, computer science education and broadening participation in computing, and knowledge graph backed language models for technical document processing. He is also strongly committed to incorporating ethics into computer science and engineering education, and the Computer Science program that he has developed and launched at Saint Mary's College of California centers ethics and social justice while teaching students to be high-quality computing professionals.

Aubrey Lawson, Wiley

Aubrey Lawson is a CS Content Developer at zyBooks. She received her bachelor's and master's degrees in Computer Science from Clemson University, and her PhD research focuses on CS education.

Chris Mayfield, James Madison University

Chris Mayfield, PhD, is a Professor of Computer Science at James Madison University. His research focuses on CS education and professional development at the undergraduate and high school levels. He received a PhD in Computer Science from Purdue University and bachelor’s degrees in CS and German from the University of Utah.

Narges Norouzi, UC Berkeley

Narges Norouzi received her MS and PhD from the University of Toronto, focusing on applied deep learning. She has since been involved in working on applied machine learning projects with a focus on biology and education. Her CS education research focuses on using artificial intelligence in the classroom to close the equity gap and leading student-centered programs that promote equity and access.

Contributing Authors

Contributing authors, left to right: Yamuna Rajasekhar and Reed Kanemaru.

Yamuna Rajasekhar, Wiley

Yamuna Rajasekhar, PhD, is Director of Content, Authoring, and Research at Wiley. She works across disciplines on research strategy, authoring pedagogy and training, and content development for Computer Science and IT. She received her MS and PhD from University of North Carolina at Charlotte, focusing on Computer Engineering education. Prior to joining Wiley as a content author, Yamuna was an Assistant Professor of Computer Engineering at Miami University, where her research was focused on assistive technology with embedded systems and Computer Engineering education.

Reed Kanemaru, Wiley

Reed Kanemaru earned a BS in Computer Science from University of California, Riverside in 2020 and an MS in Computer Science from University of California, Riverside in 2021. Since graduating, he has worked as a Content/Software Developer at zyBooks.

Reviewers

Mel Akhimiemona, Community College of Baltimore County

Doina Bein, Cal State Fullerton

Phillip Bradford, University of Connecticut

James Braman, Community College of Baltimore County

Robert Burrows, College of DuPage

Deena Engel, New York University

Gabriel Ferrer, Hendrix College

Nazli Hardy, Millersville University

Matthew Hertz, University at Buffalo

Rania Hodhod, Columbus State University

Akira Kawaguchi, The City College of New York

Kevin Lin, University of Washington

Matin Pirouz, Fresno State

Muhammad Rahman, Clayton State University

Jerry Reed, Valencia College

Kathleen Tamerlano, Cuyahoga Community College

Linda Tansil

Academic Integrity

Academic integrity builds trust, understanding, equity, and genuine learning. While students may encounter significant challenges in their courses and their lives, doing their own work and maintaining a high degree of authenticity will result in meaningful outcomes that will extend far beyond their college career. Faculty, administrators, resource providers, and students should work together to maintain a fair and positive experience.

We realize that students benefit when academic integrity ground rules are established early in the course. To that end, OpenStax has created an interactive to aid with academic integrity discussions in your course.

attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license

Visit our academic integrity slider (https://www.openstax.org/r/academic-integrity-slider). Click and drag icons along the continuum to align these practices with your institution and course policies. You may then include the graphic on your syllabus, present it in your first course meeting, or create a handout for students.

At OpenStax we are also developing resources supporting authentic learning experiences and assessment. Please visit this book's page for updates. For an in-depth review of academic integrity strategies, we highly recommend visiting the International Center of Academic Integrity (ICAI) website at https://academicintegrity.org/ (https://academicintegrity.org/).

Community Hubs

OpenStax partners with the Institute for the Study of Knowledge Management in Education (ISKME) to offer Community Hubs on OER Commons—a platform for instructors to share community-created resources that support OpenStax books, free of charge. Through our Community Hubs, instructors can upload their own materials or download resources to use in their own courses, including additional ancillaries, teaching material, multimedia, and relevant course content. We encourage instructors to join the hubs for the subjects most relevant to your teaching and research as an opportunity both to enrich your courses and to engage with other faculty. To reach the Community Hubs, visit www.oercommons.org/hubs/openstax (https://oercommons.org/groups/openstax-introduction-to-python-programming/14678/?__hub_id=27).

Technology partners

As allies in making high-quality learning materials accessible, our technology partners offer optional low-cost tools that are integrated with OpenStax books. To access the technology options for your text, visit your book page on OpenStax.org.

1 Statements

Figure 1.1 credit: Larissa Chu, CC BY 4.0

Chapter Outline

Introduction

Computers and programs are everywhere in today's world. Programs affect many aspects of daily life and society as a whole. People depend on programs for communication, shopping, entertainment, health care, and countless other needs. Learning how to program computers opens the door to many careers and opportunities for building a better world.

Programs consist of statements to be run one after the other. A **statement** describes some action to be carried out.

The statement `print("Good morning")` instructs Python to output the message `"Good morning"` to the user. The statement `count = 0` instructs Python to assign the integer `0` to the variable `count`.

This chapter introduces statements for input and output, assigning variables, and basic arithmetic. Making mistakes is a normal part of programming, and the chapter includes advice on understanding error messages. The chapter ends with a short history of Python and discusses why Python has become so popular today.

1.1 Background

Learning Objectives

By the end of this section you should be able to

- Name two examples of computer programs in everyday life.
- Explain why Python is a good programming language to learn.

Computer programs

A **computer** is an electronic device that stores and processes information. Examples of computers include smartphones, tablets, laptops, desktops, and servers. Technically, a **program** is a sequence of instructions that a computer can run. Programs help people accomplish everyday tasks, create new technology, and have fun.

The goal of this book is to teach introductory programming and problem solving. Writing programs is a creative activity, inherently useful, and rewarding! No prior background in computer science is necessary to read this book. Many different types of programs exist, as shown in the illustration below. This book will focus on general purpose programs that typically run "behind the scenes."

CHECKPOINT

Online music streaming

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-1-background)

CONCEPTS IN PRACTICE

Computers and programs

1. How many types of programs were described in the animation?
   a. 3
   b. 4
   c. 5

2. What type of program will this book explain how to write?
   a. a tool that summarizes an individual's music preferences
   b. a mobile app for managing and sharing playlists of songs
   c. a website that shows the top artists for the past five years

3. Which of the following devices is an example of a computer?
   a. wired headphones that plug into a smartphone
   b. remote control that pauses or skips the current song
   c. wi-fi speaker that streams music from Amazon

4. Reading this book requires a strong background in mathematics.
   a. true
   b. false

EXPLORING FURTHER

Later chapters of this book show how to write analysis programs using real data. Example libraries that provide access to online streaming services include **Spotipy** (https://openstax.org/r/100spotipy), **Pytube** (https://openstax.org/r/100pytube), and **Pydora** (https://openstax.org/r/100pydora). Python-related tools often have the letters "py" in their name.

The Python language

This book introduces Python (https://openstax.org/r/100python), one of the top programming languages today. Leading tech giants like Google, Apple, NASA, Instagram, Pixar, and others use Python extensively.

One reason why Python is popular is because many libraries exist for doing real work. A **library** is a collection of code that can be used in other programs. Python comes with an extensive Standard Library (https://openstax.org/r/100pythlibrary) for solving everyday computing problems like extracting data from files and creating summary reports. In addition, the community develops many other libraries for Python. Ex: Pandas (https://openstax.org/r/100pandas) is a widely used library for data analysis.

Another reason why Python is popular is because the syntax is concise and straightforward. The **syntax** of a language defines how code must be structured. Syntax rules define the keywords, symbols, and formatting used in programs. Compared to other programming languages, Python is more concise and straightforward.

EXAMPLE 1.1

Hello world in Python and Java

By tradition, Hello World (https://openstax.org/r/100helloworld) is the first program to write when learning a new language. This program simply displays the message `"Hello, World!"` to the user. The hello world program is only one line in Python:

```
print("Hello, World!")
```

In contrast, the hello world program is five lines in Java (a different language).

```
public class Hello {
  public static void main(String[] args) {
   System.out.println("Hello, World!");
  }
}
```

However, conciseness is not the only consideration for which language is used. In different situations different languages may be more appropriate. Ex: Java is often used in Android development.

CHECKPOINT

Counting lines in a file

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-1-background)

CONCEPTS IN PRACTICE

Python vs Java syntax

5. In general, Python programs are _____ than Java programs.
   a. faster
   b. longer
   c. shorter

6. In the example programs above, what syntax required by Java is not required by Python?
   a. semicolons
   b. parentheses
   c. quote marks

TRY IT

Favorite song

The program below asks for your name and displays a friendly greeting. Run the program and see what happens. In the error message, EOF stands for End of File.

- Many of the programs in this chapter expect input from the user. Enter your name in the Input box below the code. Run the program again, and see what changes.
- Copy the following lines to the end of the program: `print("What is your favorite song?")` `song = input() print("Cool! I like", song, "too.")`
- The modified program reads two lines of input: `name` and `song`. Add your favorite song to the Input box below your name, and run the program again.

The next section of the book will explain how `print()` and `input()` work. Feel free to experiment with this code until you are ready to move on.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-1-background)

1.2 Input/output

Learning objectives

By the end of this section you should be able to

- Display output using the `print()` function.
- Obtain user input using the `input()` function.

Basic output

The **`print()`** function displays output to the user. **Output** is the information or result produced by a program. The `sep` and `end` options can be used to customize the output. Table 1.1 shows examples of `sep` and `end`.

Multiple values, separated by commas, can be printed in the same statement. By default, each value is separated by a space character in the output. The `sep` option can be used to change this behavior.

By default, the `print()` function adds a newline character at the end of the output. A **newline** character tells

the display to move to the next line. The end option can be used to continue printing on the same line.

Code	Output
`print("Today is Monday.")` `print("I like string beans.")`	`Today is Monday.` `I like string beans.`
`print("Today", "is", "Monday")` `print("Today", "is", "Monday", sep="...")`	`Today is Monday` `Today...is...Monday`
`print("Today is Monday, ", end="")` `print("I like string beans.")`	`Today is Monday, I like string beans.`
`print("Today", "is", "Monday", sep="? ", end="!!")` `print("I like string beans.")`	`Today? is? Monday!!I like string beans.`

Table 1.1 Example uses of `print()`.

CHECKPOINT

Displaying output to the user

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-2-inputoutput)

CONCEPTS IN PRACTICE

The print() function

1. Which line of code prints `Hello world!` as one line of output?
   a. `print(Hello world!)`
   b. `print("Hello", "world", "!")`
   c. `print("Hello world!")`

2. Which lines of code prints `Hello world!` as one line of output?
   a. `print("Hello")`
      `print(" world!")`
   b. `print("Hello")`
      `print(" world!", end="")`
   c. `print("Hello", end="")`
      `print(" world!")`

3. What output is produced by the following statement?

```
print("555", "0123", sep="-")
```

a. `555 0123`
b. `5550123-`
c. `555-0123`

DO SPACES REALLY MATTER?

Spaces and newline characters are not inherently important. However, learning to be precise is an essential skill for programming. Noticing little details, like how words are separated and how lines end, helps new programmers become better.

Basic input

Computer programs often receive input from the user. **Input** is what a user enters into a program. An input statement, `variable = input("prompt")`, has three parts:

1. A **variable** refers to a value stored in memory. In the statement above, *variable* can be replaced with any name the programmer chooses.
2. The **input()** function reads one line of input from the user. A function is a named, reusable block of code that performs a task when called. The input is stored in the computer's memory and can be accessed later using the variable.
3. A **prompt** is a short message that indicates the program is waiting for input. In the statement above, *"prompt"* can be omitted or replaced with any message.

CHECKPOINT

Obtaining input from the user

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-2-inputoutput)

CONCEPTS IN PRACTICE

The input() function

4. Which line of code correctly obtains and stores user input?
a. `input()`
b. `today_is = input`
c. `today_is = input()`

5. Someone named Sophia enters their name when prompted with

```
print("Please enter your name: ")
name = input()
```

What is displayed by `print("You entered:", name)`?
a. `You entered: name`
b. `You entered: Sophia`

c. `You entered:, Sophia`

6. What is the output if the user enters "six" as the input?

```
print("Please enter a number: ")
number = input()
print("Value =", number)
```

a. `Value = six`
b. `Value = 6`
c. `Value = number`

TRY IT

Frost poem

Write a program that uses multiple `print()` statements to output the following poem by Robert Frost (https://openstax.org/r/100robertfrost). Each `print()` statement should correspond to one line of output.

Tip: You don't need to write the entire program all at once. Try writing the first `print()` statement, and then click the Run button. Then write the next `print()` statement, and click the Run button again. Continue writing and testing the code incrementally until you finish the program.

```
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I--
I took the one less traveled by,
And that has made all the difference.
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-2-inputoutput)

TRY IT

Name and likes

Write a program that asks the following two questions (example input in bold):

Shakira
`What do you like?` **`singing`**

```
What is your name? Shakira
What do you like? singing
```

Output a blank line after reading the input. Then output the following message based on the input:

```
Shakira likes singing

    Shakira likes singing
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-2-inputoutput)

1.3 Variables

Learning objectives

By the end of this section you should be able to

- Assign variables and print variables.
- Explain rules for naming variables.

Assignment statement

Variables allow programs to refer to values using names rather than memory locations. Ex: age refers to a person's age, and `birth` refers to a person's date of birth.

A statement can set a variable to a value using the **assignment operator** (**=**). Note that this is different from the equal sign of mathematics. Ex: `age = 6` or `birth = "May 15"`. The left side of the assignment statement is a variable, and the right side is the value the variable is assigned.

CHECKPOINT

Assigning and using variables

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-3-variables)

CONCEPTS IN PRACTICE

Assigning and using variables

1. Which line of code correctly retrieves the value of the variable, `city`, after the following assignment?

```
city = "Chicago"
```

   a. `print("In which city do you live?")`
   b. `city = "London"`
   c. `print("The city where you live is", city)`

2. Which program stores and retrieves a variable correctly?
   a.
```
print("Total =", total)
total = 6
```
   b.
```
total = 6
print("Total =", total)
```
   c. `print("Total =", total)`

```
total = input()
```

3. Which is the assignment operator?
   a. `:`
   b. `==`
   c. `=`

4. Which is a valid assignment?
   a. `temperature = 98.5`
   b. `98.5 = temperature`
   c. `temperature - 23.2`

Variable naming rules

A variable name can consist of letters, digits, and underscores and be of any length. The name cannot start with a digit. Ex: `101class` is invalid. Also, letter case matters. Ex: `Total` is different from `total`. Python's style guide recommends writing variable names in **snake case**, which is all lowercase with underscores in between each word, such as `first_name` or `total_price`.

A name should be short and descriptive, so words are preferred over single characters in programs for readability. Ex: A variable named `count` indicates the variable's purpose better than a variable named `c`.

Python has reserved words, known as **keywords**, which have special functions and cannot be used as names for variables (or other objects).

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

Table 1.2 Keywords

CONCEPTS IN PRACTICE

Valid variable names

5. Which can be used as a variable name?
   a. median
   b. class

c. import

6. Why is the name, `2nd_input`, not a valid variable name?
   a. contains an underscore
   b. starts with a digit
   c. is a keyword

7. Which would be a good name for a variable storing a zip code?
   a. `z`
   b. `var_2`
   c. `zip_code`

8. Given the variable name, `DogBreed`, which improvement conforms to Python's style guide?
   a. `dog_breed`
   b. `dogBreed`
   c. `dog-breed`

TRY IT

Final score

Write a Python computer program that:

- Creates a variable, `team1`, assigned with the value `"Liverpool"`.
- Creates a variable, `team2`, assigned with the value `"Chelsea"`.
- Creates a variable `score1`, assigned with the value `4`.
- Creates a variable, `score2`, assigned with the value `3`.
- Prints `team1`, `"versus"`, and `team2` as a single line of output.
- Prints `"Final score: "`, `score1`, `"to"`, `score2` as a single line of output.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-3-variables)

1.4 String basics

Learning objectives

By the end of this section you should be able to

- Use the built-in `len()` function to get a string's length.
- Concatenate string literals and variables using the + operator.

Quote marks

A string is a sequence of characters enclosed by matching single (') or double (") quotes. Ex: `"Happy birthday!"` and `'21'` are both strings.

To include a single quote (') in a string, enclose the string with matching double quotes ("). Ex: `"Won't this work?"` To include a double quote ("), enclose the string with matching single quotes ('). Ex: `'They said "Try it!", so I did'`.

Valid string	Invalid string
`"17" or '17'`	`17`
`"seventeen" or 'seventeen'`	`seventeen`
`"Where?" or 'Where?'`	`"Where?'`
`"I hope you aren't sad."`	`'I hope you aren't sad.'`
`'The teacher said "Correct!" '`	`"The teacher said "Correct!" "`

Table 1.3 Rules for strings.

CONCEPTS IN PRACTICE

Valid and invalid strings

1. Which of the following is a string?
   a. `Hello!`
   b. `29`
   c. `"7 days"`

2. Which line of code assigns a string to the variable `email`?
   a. `"fred78@gmail.com"`
   b. `"email = fred78@gmail.com"`
   c. `email = "fred78@gmail.com"`

3. Which is a valid string?
   a. `I know you'll answer correctly!`
   b. `'I know you'll answer correctly!'`
   c. `"I know you'll answer correctly!"`

4. Which is a valid string?
   a. `You say "Please" to be polite`
   b. `"You say "Please" to be polite"`
   c. `'You say "Please" to be polite'`

len() function

A common operation on a string object is to get the string length, or the number of characters in the string. The **len()** function, when called on a string value, returns the string length.

CHECKPOINT

Using len() to get the length of a string

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/

1-4-string-basics)

CONCEPTS IN PRACTICE

Applying len() function to string values

5. What is the return value for `len("Hi Ali")`?
   a. `2`
   b. `5`
   c. `6`

6. What is the length of an empty string variable (`""`)?
   a. undefined
   b. 0
   c. 2

7. What is the output of the following code?

```
number = "12"
number_of_digits = len(number)
print("Number", number, "has", number_of_digits, "digits.")
```

   a. `Number 12 has 12 digits.`
   b. `Number 12 has 2 digits.`
   c. `Number 12 has number_of_digits digits.`

Concatenation

Concatenation is an operation that combines two or more strings sequentially with the concatenation operator (+). Ex: `"A"` + `"part"` produces the string `"Apart"`.

CHECKPOINT

Concatenating multiple strings

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-4-string-basics)

CONCEPTS IN PRACTICE

String concatenation

8. Which produces the string `"10"`?
   a. `1 + 0`
   b. `"1 + 0"`
   c. `"1" + "0"`

9. Which produces the string "Awake"?

a. `"wake" + "A"`
b. `"A + wake"`
c. `"A" + "wake"`

10. A user enters `"red"` after the following line of code.

```
color = input("What is your favorite color?")
```

Which produces the output `"Your favorite color is red!"`?

a. `print("Your favorite color is " + color + !)`
b. `print("Your favorite color is " + "color" + "!")`
c. `print("Your favorite color is " + color + "!")`

11. Which of the following assigns `"one-sided"` to the variable `holiday`?

a. `holiday = "one" + "sided"`
b. `holiday = one-sided`
c. `holiday = "one-" + "sided"`

TRY IT

Name length

Write a program that asks the user to input their first and last name separately. Use the following prompts (example input in bold):

```
What is your first name? Alan
What is your last name? Turing
```

The program should then output the length of each name. Based on the example input above, the output would be:

```
Your first name is 4 letters long
Your last name is 6 letters long
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-4-string-basics)

TRY IT

Punctuation

Write a Python computer program that:

- Assigns the string `"Freda"` to a variable, `name`.

- Assigns the string `"happy"` to a variable, `feel`.
- Prints the string `"Hi Freda!"` with a single `print()` function using the variable `name`.
- Prints the string `"I'm glad you feel happy."` with a single `print()` function using the variable `feel`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-4-string-basics)

1.5 Number basics

Learning objectives

By the end of this section you should be able to

- Use arithmetic operators to perform calculations.
- Explain the precedence of arithmetic operators.

Numeric data types

Python supports two basic number formats, integer and floating-point. An integer represents a whole number, and a floating-point format represents a decimal number. The format a language uses to represent data is called a **data type**. In addition to integer and floating-point types, programming languages typically have a string type for representing text.

CHECKPOINT

Integer and floating-point

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-5-number-basics)

CONCEPTS IN PRACTICE

Integers, floats, and strings

Assume that `x = 1`, `y = 2.0`, and `s = "32"`.

1. What is the output of the following code?

```
print(x, type(x))
```

   a. `1 'int'`.
   b. `1.0 <class 'float'>`.
   c. `1 <class 'int'>`.

2. What is the output of the following code?

```
print(y, type(y))
```

   a. `2.0 <class 'int'>`
   b. `2.0 <class 'float'>`
   c. `2 <class 'int'>`

3. What is the type of the following value?

```
"12.0"
```

a. `string`
b. `int`
c. `float`

Basic arithmetic

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, and division.

Four basic arithmetic operators exist in Python:

1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)

CHECKPOINT

Examples of arithmetic operators

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-5-number-basics)

CONCEPTS IN PRACTICE

Applying arithmetic operators

Assume that `x = 7`, `y = 20`, and `z = 2`.

4. Given the following lines of code, what is the output of the code?

```
c = 0
c = x - z
c = c + 1
print(c)
```

a. `1`
b. `5`
c. `6`

5. What is the value of a?

```
a = 3.5 - 1.5
```

a. `2`
b. `2.0`

c. `2.5`

6. What is the output of `print(x / z)`?
   a. `3`
   b. `3.0`
   c. `3.5`

7. What is the output of `print(y / z)`?
   a. `0`
   b. `10`
   c. `10.0`

8. What is the output of `print(z * 1.5)`?
   a. `2`
   b. `3`
   c. `3.0`

Operator precedence

When a calculation has multiple operators, each operator is evaluated in order of **precedence**. Ex: `1 + 2 * 3` is `7` because multiplication takes precedence over addition. However, `(1 + 2) * 3` is `9` because parentheses take precedence over multiplication.

Operator	Description	Example	Result
`()`	Parentheses	`(1 + 2) * 3`	`9`
`**`	Exponentiation	`2 ** 4`	`16`
`+, -`	Positive, negative	`-math.pi`	`-3.14159`
`*, /`	Multiplication, division	`2 * 3`	`6`
`+, -`	Addition, subtraction	`1 + 2`	`3`

Table 1.4 Operator precedence from highest to lowest.

CHECKPOINT

Order of operations in an arithmetic expression

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-5-number-basics)

CONCEPTS IN PRACTICE

Multiple arithmetic operators

9. What is the value of `4 * 3 ** 2 + 1`?

a. `37`
b. `40`
c. `145`

10. Which part of `(1 + 3) ** 2 / 4` evaluates first?

a. `4 ** 2`
b. `1 + 3`
c. `2 / 4`

11. What is the value of `-4 ** 2`?

a. `-16`
b. `16`

12. How many operators are in the following statement?

```
result = -2 ** 3
```

a. 1
b. 2
c. 3

TRY IT

Values and types

Write a Python computer program that:

1. Defines an integer variable named `'int_a'` and assigns `'int_a'` with the value `10`.
2. Defines a floating-point variable named `'float_a'` and assigns `'float_a'` with the value `10.0`.
3. Defines a string variable named `'string_a'` and assigns `'string_a'` with the string value `"10"`.
4. Prints the value of each of the three variables along with their type.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-5-number-basics)

TRY IT

Meters to feet

Write a Python computer program that:

1. Assigns the integer value `10` to a variable, `meters`.
2. Assigns the floating-point value `3.28` to a variable, `meter2feet`.
3. Calculates 10 meters in feet by multiplying `meter` by `meter2feet`. Store the result in a variable, `feet`.

4. Prints the content of variable `feet` in the output.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-5-number-basics)

1.6 Error messages

Learning objectives

By the end of this section you should be able to

- Identify the error type and line number in error messages.
- Correct syntax errors, name errors, and indentation errors.

How to read errors

A natural part of programming is making mistakes. Even experienced programmers make mistakes when writing code. Errors may result when mistakes are made when writing code. The computer requires very specific instructions telling the computer what to do. If the instructions are not clear, then the computer does not know what to do and gives back an error.

When an error occurs, Python displays a message with the following information:

1. The line number of the error.
2. The type of error (Ex: `SyntaxError`).
3. Additional details about the error.

Ex: Typing `print "Hello!"` without parentheses is a syntax error. In Python, parentheses are required to use `print`. When attempting to run `print "Hello!"`, Python displays the following error:

```
Traceback (most recent call last):
  File "/home/student/Desktop/example.py", line 1
    print "Hello"
                ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("Hello")?
```

The caret character (^) shows where Python found the error. Sometimes the error may be located one or two lines before where the caret symbol is shown because Python may not have discovered the error until then. **Traceback** is a Python report of the location and type of error. The word traceback suggests a programmer trace back in the code to find the error if the error is not seen right away.

Learning to read error messages carefully is an important skill. The amount of technical jargon can be overwhelming at first. But this information can be very helpful.

CHECKPOINT

Incorrect variable name

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-6-error-messages)

CONCEPTS IN PRACTICE

Parts of an error

Given the following error message:

```
Traceback (most recent call last):
  File "/home/student/Desktop/example.py", line 2
    print "test"
        ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("test")?
```

1. What is the filename of the program?
   a. Desktop
   b. example.py
   c. test

2. On which line was the error found?
   a. 1
   b. 2
   c. 3

3. What type of error was found?
   a. missing parentheses
   b. `SyntaxError`
   c. traceback

Common types of errors

Different types of errors may occur when running Python programs. When an error occurs, knowing the type of error gives insight about how to correct the error. The following table shows examples of mistakes that anyone could make when programming.

Mistake	Error message	Explanation
`print("Have a nice day!"`	`SyntaxError: unexpected EOF while parsing`	The closing parenthesis is missing. Python is surprised to reach the end of file (EOF) before this line is complete.
`word = input("Type a word: )`	`SyntaxError: EOL while scanning string literal`	The closing quote marks are missing. As a result, the string does not terminate before the end of line (EOL).
`print("You typed:", wird)`	`NameError: name 'wird' is not defined`	The spelling of `word` is incorrect. The programmer accidentally typed the wrong key.

Table 1.5 Simple mistakes.

Mistake	Error message	Explanation
`prints("You typed:", word)`	`NameError: name 'prints' is not defined`	The spelling of `print` is incorrect. The programmer accidentally typed an extra letter.
`  print("Hello")`	`IndentationError: unexpected indent`	The programmer accidentally typed a space at the start of the line.
`    print("Goodbye")`	`IndentationError: unexpected indent`	The programmer accidentally pressed the Tab key at the start of the line.

Table 1.5 Simple mistakes.

CONCEPTS IN PRACTICE

Types of errors

For each program below, what type of error will occur?

4.
```
print("Breakfast options:")
 print("A. Cereal")
 print("B. Eggs")
 print("C. Yogurt")
choice = input("What would you like? ")
```

a. IndentationError
b. NameError
c. SyntaxError

5.
```
birth = input("Enter your birth date: )
print("Happy birthday on ", birth)
```

a. IndentationError
b. NameError
c. SyntaxError

6.
```
print("Breakfast options:")
print(" A. Cereal")
print(" B. Eggs")
print(" C. Yogurt")
choice = intput("What would you like? ")
```

a. IndentationError

```
    b. NameError
    c. SyntaxError
```

TRY IT

Three errors

The following program has three errors.

- Run the program to find the first error, and correct the corresponding line of code.
- Then run the program again to find and correct the second error.
- Keep running and correcting the program until no errors are found.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-6-error-messages)

TRY IT

Wrong symbols

This code is based on an earlier example, but the code contains several mistakes.

- One line is missing required punctuation, and another line uses incorrect symbols.
- Run the program to find the first error, and correct the corresponding line of code.
- Keep running and correcting the program until no errors are found.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-6-error-messages)

1.7 Comments

Learning objectives

By the end of this section you should be able to

- Write concise, meaningful comments that explain intended functionality of the code.
- Write a docstring (more verbose comment) that describes the program functionality.

The hash character

Comments are short phrases that explain what the code is doing. Ex: Lines 1, 8, and 10 in the following program contain comments. Each comment begins with a hash character (*#*). All text from the hash character to the end of the line is ignored when running the program. In contrast, hash characters inside of strings are treated as regular text. Ex: The string `"Item #1: "` does not contain a comment.

When writing comments:

- The *#* character should be followed by a single space. Ex: *`# End of menu`* is easier to read than *`#End of menu`*.
- Comments should explain the purpose of the code, not just repeat the code itself. Ex: *`# Get the user's preferences`* is more descriptive than *`# Input item1 and item2`*.

EXAMPLE 1.2

Program with three comments

```
1  # Display the menu options
2  print("Lunch Menu")
3  print("----------")
4  print("Burrito")
5  print("Enchilada")
6  print("Taco")
7  print("Salad")
8  print() # End of menu
9
10 # Get the user's preferences
11 item1 = input("Item #1: ")
12 item2 = input("Item #2: ")
```

CONCEPTS IN PRACTICE

Simple comments

1. The main purpose of writing comments is to ____.
   a. avoid writing syntax errors
   b. explain what the code does
   c. make the code run faster

2. Which symbol is used for comments in Python?
   a. #
   b. /*
   c. //

3. Which comment is formatted correctly?
   a. 0 spaces:
      `#Get the user input`
   b. 1 space:
      `# Get the user input`
   c. 2 spaces:
      `#  Get the user input`

Code quality

The example program above had two parts: (1) display the menu options, and (2) get the user's preferences. Together, the blank lines and comments show the overall structure of the program.

Programmers spend more time reading code than writing code. Therefore, making code easier for others to read and understand is important. Two ways to improve code quality include:

- Separate each part (lines that have a similar purpose) with a blank line.

- Write a comment before each part. Not every line needs a comment.

CHECKPOINT

Comments in a program

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-7-comments)

CONCEPTS IN PRACTICE

Code quality

4. Which comment is most useful for the following code?

```
print("You said:", adj1 + " " + noun1)
```

a. `# Append adj1 and noun1`
b. `# Print out a bunch of stuff`
c. `# Show the resulting phrase`

5. Where should a blank line be inserted?

```
1 name = input("Whose birthday is today? ")
2 print("Happy birthday to", name)
3 print("Everyone cheer for", name)
```

a. After line 1
b. After line 2
c. After line 3

6. To temporarily prevent a line from being run, one might . . .
a. introduce a syntax error in the line.
b. remove the line from the program.
c. insert a # at the beginning of the line.

Documentation

Python programs may optionally begin with a string known as a docstring. A **docstring** is documentation written for others who will use the program but not necessarily read the source code. Most of the official documentation at docs.python.org (https://openstax.org/r/100docstrings) is generated from docstrings.

Documentation can be long, so docstrings are generally written as multi-line strings (`"""`). Common elements of a docstring include a one-line summary, a blank line, and a longer description.

CHECKPOINT

Vacations docstring

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-7-comments)

CONCEPTS IN PRACTICE

Documentation

7. The main purpose of writing docstrings is to . . .
   a. summarize the program's purpose or usage.
   b. explain how each part of the code works.
   c. maintain a list of ideas for new features.

8. Which of the following is NOT a docstring?
   a. `"""Vacations Madlib."""`
   b. `"""Vacations Madlib. This program asks the user for two adjectives and two nouns, which are then used to print a funny story about a vacation. """`
   c.
```
# Vacations Madlib.
#
# This program asks the user for two adjectives
# and two nouns, which are then used to print
# a funny story about a vacation.
```

9. Which docstring is most useful for this program?
   a. `"""Vacations Madlib."""`
   b. `"""Vacations Madlib. This program asks the user for two adjectives and two nouns, which are then used to print a funny story about a vacation. """`
   c. `"""Vacations Madlib. This program asks the user for two adjectives and two nouns, which are then used to print a funny story about a vacation. The code uses four variables to store the user input: two for the adjectives, and two for the nouns. The output is displayed on seven lines, beginning with a blank line after the input. """`

TRY IT

Whose birthday

Add two comments to the following program: one for the input, and one for the output. Separate the input and output with a blank line. Then, compare your comments with the sample solution, and ask yourself the following questions:

- Are your comments longer or shorter? Why?
- Is the formatting of your comments correct?

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-7-comments)

TRY IT

Gravity calculation

Write a docstring for the following program. The first line of the docstring should explain, in one short

sentence, what the program is. The second line of the docstring should be blank. The third and subsequent lines should include a longer explanation of what the program does. At the end of the docstring, add a line that says `"Author: "` followed by your name.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-7-comments)

1.8 Why Python?

Learning objectives

By the end of this section you should be able to

- Name two historical facts about how Python was first created.
- Describe two ways Python is considered a popular language.

Historical background

Python has an interesting history. In 1982, Guido van Rossum (https://openstax.org/r/100vanRossum), the creator of Python, started working at CWI (https://openstax.org/r/100CWI), a Dutch national research institute. He joined a team that was designing a new programming language, named ABC, for teaching and prototyping. ABC's simplicity was ideal for beginners, but the language lacked features required to write advanced programs.

Several years later, van Rossum joined a different team at CWI working on an operating system. The team needed an easier way to write programs for monitoring computers and analyzing data. Languages common in the 1980's were (and still are) difficult to use for these kinds of programs. van Rossum envisioned a new language that would have a simple syntax, like ABC, but also provide advanced features that professionals would need.

At first, van Rossum started working on this new language as a hobby during his free time. He named the language Python because he was a fan of the British comedy group Monty Python (https://openstax.org/r/100MontyPython). Over the next year, he and his colleagues successfully used Python many times for real work. van Rossum eventually decided to share Python with the broader programming community online. He freely shared Python's entire source code so that anyone could write and run Python programs.

Python's first release, known as Version 0.9.0, appeared in 1991, about six years after C++ and four years before Java. van Rossum's decisions to make the language simple yet advanced, suitable for everyday tasks, and freely available online contributed to Python's long-term success.

CHECKPOINT

Key decisions

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-8-why-python)

CONCEPTS IN PRACTICE

Python history

1. The Python programming language was named after a _____.

a. British comedy group
b. Dutch programmer
c. non-venomous snake

2. Which programming language came first?
   a. Java
   b. Python

3. Which sentence best describes the beginning of Python?
   a. CWI hired Guido van Rossum to design a new programming language to compete with C++.
   b. Python started out as a hobby and became open source after several years of development.
   c. van Rossum posted Python's source code online after working on the language for one year.

EXPLORING FURTHER

For more details about Python's history, see "A brief history of Python (https://openstax.org/r/100history)" by Vasilisa Sheromova, and "History and Philosophy of Python (https://openstax.org/r/100sheromova)" by Bernd Klein.

Popularity of Python

Over the years, Python has become a nonprofit organization with a thriving community. Millions of programmers around the world use Python for all kinds of interesting projects. Hundreds of thousands of Python libraries have been released as open source software. The Python community is very active and supportive online, answering questions and sharing code.

One way to see Python's popularity is the TIOBE index (https://openstax.org/r/100TIOBE). TIOBE is a Dutch company that provides products and services for measuring software code quality. Since 2001, TIOBE has tracked the popularity of programming languages and posted the results online. Figure 1.2 shows the TIOBE index over time for five of the most popular languages.

The TIOBE index is based on the number of search engine results for each language. The percentage refers to how many results belong to that language. Python has been among the top 10 languages every year since 2004. In October 2021, Python became the #1 language on the TIOBE index. No other language but C and Java had been #1 for the previous 20 years.

Another way to see Python's popularity is to analyze how frequently Python is discussed online. Stack Overflow (https://openstax.org/r/100overflow) is a question-and-answer website for programmers. Figure 1.3 shows the number of questions asked each month that were tagged with Python, JavaScript, and so forth. In recent years, Python has become the most asked about language in programming forums.

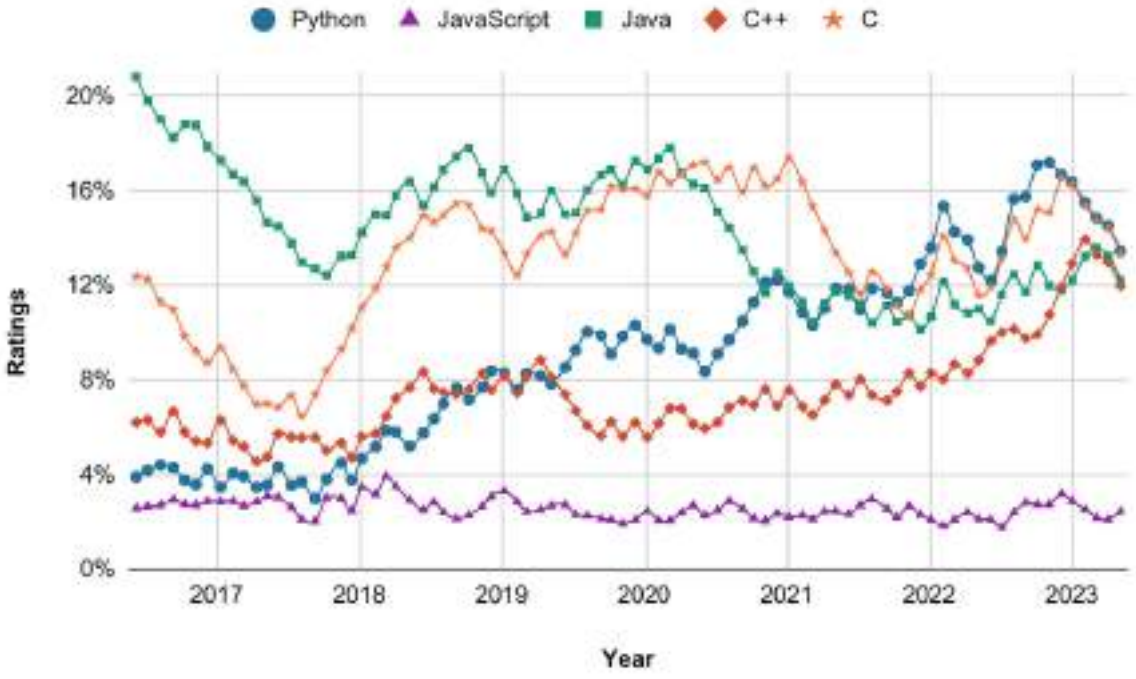


Figure 1.2 **TIOBE programming community index**. Source: www.tiobe.com (https://www.tiobe.com/tiobe-index/)

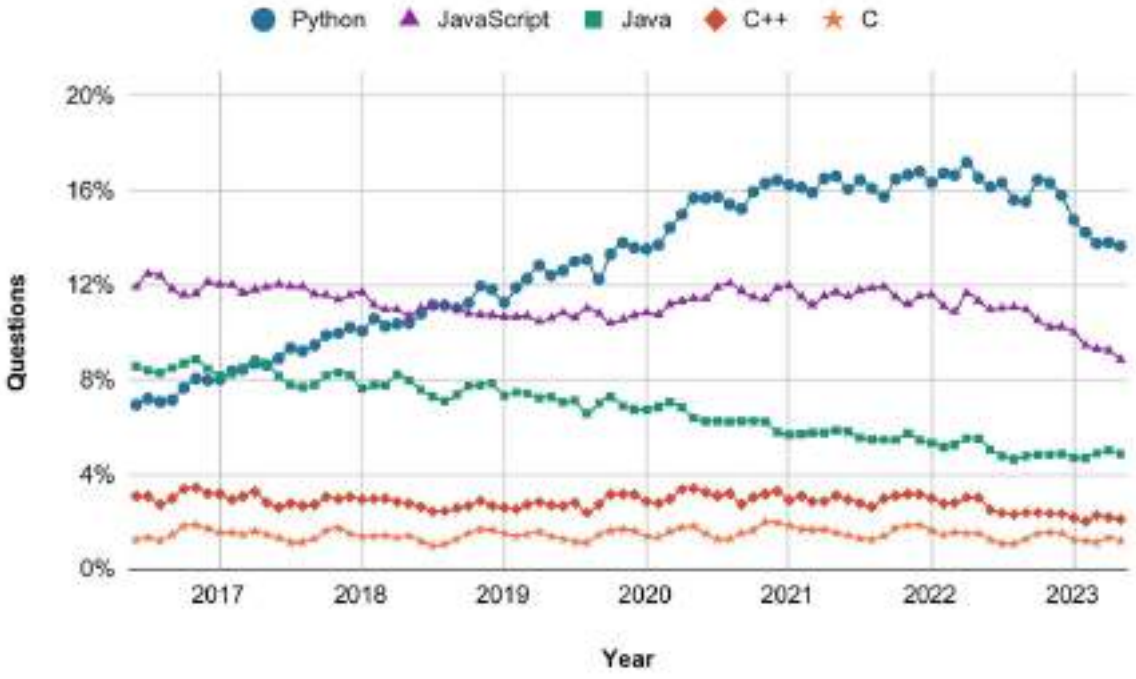


Figure 1.3 **Stack Overflow questions per month**. Source: data.stackexchange.com (https://data.stackexchange.com/)

CONCEPTS IN PRACTICE

Python popularity

4. According to the TIOBE Index, which two languages were most popular from 2001 to 2021?

a. C and C++
b. Java and C
c. Python and JavaScript

5. In what year did Python become the most asked about language on Stack Overflow?
   a. 2018
   b. 2019
   c. 2020

6. How long has TIOBE been tracking programming language popularity?
   a. since 1991
   b. since 2001
   c. since 2015

1.9 Chapter summary

This chapter introduced the basics of programming in Python, including:

- `print()` and `input()`.
- Variables and assignment.
- Strings, integers, and floats.
- Arithmetic, concatenation.
- Common error messages.
- Comments and docstrings.

At this point, you should be able to write programs that ask for input, perform simple calculations, and output the results. The programming practice below ties together most topics presented in the chapter.

Function	Description
`print(values)`	Outputs one or more values, each separated by a space, to the user.
`input(prompt)`	If present, `prompt` is output to the user. The function then reads a line of input from the user.
`len(string)`	Returns the length (the number of characters) of a string.
`type(value)`	Returns the type (or class) of a value. Ex: `type(123)` is `<class 'int'>`.
Operator	**Description**
= (Assignment)	Assigns (or updates) the value of a variable. In Python, variables begin to exist when assigned for the first time.

Table 1.6 Chapter 1 reference.

Function	Description
+ (Concatenation)	Appends the contents of two strings, resulting in a new string.
+ (Addition)	Adds the values of two numbers.
- (Subtraction)	Subtracts the value of one number from another.
* (Multiplication)	Multiplies the values of two numbers.
/ (Division)	Divides the value of one number by another.
** (Exponentiation)	Raises a number to a power. Ex: `3**2` is three squared.
Syntax	**Description**
# (Comment)	All text is ignored from the # symbol to the end of the line.
`'` or `"` (String)	Strings may be written using either kind of quote. Ex: `'A'` and `"A"` represent the same string. By convention, this book uses double quotes (`"`) for most strings.
`"""` (Docstring)	Used for documentation, often in multi-line strings, to summarize a program's purpose or usage.

Table 1.6 Chapter 1 reference.

TRY IT

Fun facts

Write a program that assigns a variable named `number` to any integer of your choice. Ex: `number = 74`. Then, use this variable to calculate and output the following results:

```
74 squared is 5476
74 cubed is 405224
One tenth of 74 is 7.4
```

```
    74 plus 123 is 197
    74 minus 456 is -382
```

Run the program multiple times, using a different integer each time. Your output should be mathematically correct for any integer that you choose.

The point of this exercise is to perform basic arithmetic within a print statement. Do not use any other variables besides `number`. Your program should have only one assignment statement (at the beginning).

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-9-chapter-summary)

TRY IT

Mad lib

A mad lib is a word game in which one person asks others for words to substitute into a pre-written story. The story is then read aloud with the goal of making people laugh.

This exercise is based the Vacations Mad Lib (https://openstax.org/r/100madlibs) available on the Printables (https://openstax.org/r/100printable) section of MadLibs.com. Write a program that asks the user to input two adjectives and two nouns (user input in bold):

```
tranquil
    Adjective: scandalous
    Noun: pancake
    Noun: field

    Adjective: tranquil
    Adjective: scandalous
    Noun: pancake
    Noun: field
```

Use `input()` to display each prompt exactly as shown. The user's input should be on the same line as the prompt. Each colon must be followed by exactly one space. After reading the input, the program should output the following three lines:

```
tranquil place with your scandalous family.
    Usually you go to some place that is near a/an pancake or up on a/an field.

    A vacation is when you take a trip to some tranquil place with your scandalous
family.
    Usually you go to some place that is near a/an pancake or up on a/an field.
```

Notice that the first line should be completely blank. Replace the bold words (from the above example) with the actual words input by the user.

Your final program should have four input statements, three print statements, and at least two comments.

For completeness, write an appropriate docstring at the top of the program.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/1-9-chapter-summary)

2 Expressions

Figure 2.1 credit: modification of work "Grace Hopper at Univac I console", courtesy of the Computer History Museum

Chapter Outline

Introduction

A computer program is a sequence of statements that run one after the other. In Python, many statements consist of one or more expressions. An **expression** represents a single value to be computed. Ex: The expression `3*x - 5` evaluates to 7 when x is 4. Learning to recognize expressions opens the door for programming all kinds of interesting calculations.

Expressions are often a combination of literals, variables, and operators. In the previous example, 3 and 5 are literals, x is a variable, and * and - are operators. Expressions can be arbitrarily long, consisting of many calculations. Expressions can also be as short as one value. Ex: In the assignment statement `x = 5`, the literal 5 is an expression.

The Statements chapter introduced simple expressions like `1 * 2` and `"Hi " + "there"`. This chapter explores other kinds of expressions for working with numbers and strings. The first section shows a great way to experiment with expressions using a Python shell. Later sections present more details about integers and floating-point numbers, explain how to import and use the `math` module, and show how to make long lines of code easier to read.

2.1 The Python shell

Learning objectives

By the end of this section you should be able to

- Use a Python shell to run statements and expressions interactively.
- Explain the function of the up and down arrow keyboard shortcuts.

The interpreter

Python is a high-level language, meaning that the source code is intended for humans to understand. Computers, on the other hand, understand only low-level machine code made up of 1's and 0's. Programs written in high-level languages must be translated into machine code to run. This translation process can happen all at once, or a little at a time, depending on the language.

Python is an interpreted language: the source code is translated one line at a time while the program is running. The Python **interpreter** translates source code into machine code and runs the resulting program. If and when an error occurs, the interpreter stops translating the source code and displays an error message.

Most development environments include a Python shell for experimenting with code interactively. A **shell**, also called a console or terminal, is a program that allows direct interaction with an interpreter. The interpreter usually runs an entire program all at once. But the interpreter can run one line of code at a time within a Python shell.

CHECKPOINT

Running a Python shell

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-1-the-python-shell)

CONCEPTS IN PRACTICE

Using a Python shell

1. Python is a _____ language.
   a. high-level
   b. low-level

2. Which of the following is the most basic line of code the interpreter can run?
   a. `print(1 + 1)`
   b. `1 + 1`
   c. `1`

3. What result does the shell display after running the line `name = input()`?
   a. the name that was input
   b. nothing (except for >>>)

The arrow keys

A Python shell is convenient for exploring and troubleshooting code. The user can try something, look at the

results, and then try something else. When an error occurs, an error message is displayed, but the program keeps running. That way, the user can edit the previous line and correct the error interactively.

The acronym REPL (pronounced "rep ul") is often used when referring to a shell. **REPL** stands for "read-eval-print loop," which describes the repetitive nature of a shell:

1. **R**ead/input some code
2. **E**valuate/run the code
3. **P**rint any results
4. **L**oop back to step 1

Most shells maintain a history of every line of code the user types. Pressing the up or down arrow key on the keyboard displays the history. The **up arrow** displays the previous line; the **down arrow** displays the next line. That way, the user can repeat a line without having to type the line again.

CHECKPOINT

Correcting a typo

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-1-the-python-shell)

CONCEPTS IN PRACTICE

Using the arrow keys

4. Which arrow keys were needed to edit the typo?
   a. only the up arrow key
   b. the up and down arrows
   c. the up and left arrows

5. What keys would the user press to go back two lines?
   a. press the up arrow twice
   b. press the down arrow twice
   c. press the left arrow twice

TRY IT

Exploring the shell

Running code interactively is a great way to learn how Python works. Open a Python shell on your computer, or use the one at python.org/shell (https://openstax.org/r/100pythonshell). Then enter any Python code, one line at a time, to see the result. Here are a few expressions to try:

- `x = 5`
- `3*x - 5`
- `3 * (x-5)`
- `x`
- `type(1)`
- `type('1')`

- `str(1)`
- `int('1')`
- `abs(-5)`
- `abs(5)`
- `len("Yo")`
- `len("HoHo")`
- `round(9.49)`
- `round(9.50)`

Note: These functions (`type`, `str`, `int`, `len`, and `round`) will be explored in more detail later in the chapter. You can read more about the built-in functions (https://openstax.org/r/100builtin) in the Python documentation.

TRY IT

Correcting mistakes

Open a Python shell on your computer, or use the one at python.org/shell (https://openstax.org/r/100pythonshell). Run the following two statements in the shell:

- `x = 123`
- `y = 456`

Making mistakes is common while typing in a shell. The following lines include typos and other errors. For each line: (1) run the line in a shell to see the result, (2) press the up arrow to repeat the line, and (3) edit the line to get the correct result.

- `print("Easy as", X)`
- `print("y divided by 2 is", y / 0)`
- `name = intput("What is your name? ")`
- `print(name, "is", int(name), "letters long.")`
- `print("That's all folks!)`

The expected output, after correcting typos, should look like:

- `Easy as 123`
- `y divided by 2 is 228.0`
- `(no error/output)`
- `Stacie is 6 letters long.`
- `That's all folks!`

2.2 Type conversion

Learning objectives

By the end of this section you should be able to

- Explain how the interpreter uses implicit type conversion.
- Use explicit type conversion with `int()`, `float()`, and `str()`.

Implicit type conversion

Common operations update a variable such that the variable's data type needs to be changed. Ex: A GPS first assigns `distance` with `250`, an integer. After a wrong turn, the GPS assigns `distance` with `252.5`, a float. The Python interpreter uses **implicit type conversion** to automatically convert one data type to another. Once `distance` is assigned with `252.5`, the interpreter will convert `distance` from an integer to a float without the programmer needing to specify the conversion.

CHECKPOINT

Example: Book ratings

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-2-type-conversion)

CONCEPTS IN PRACTICE

Implicit type conversion in practice

Consider the example above.

1. What is `book_rating`'s data type on line 7?
   a. float
   b. integer

2. What would `book_rating`'s data type be if `update = 1.0` instead of `0.5`?
   a. float
   b. integer

3. What is the data type of x after the following code executes?

```
x = 42.0
x = x * 1
```

   a. float
   b. integer

Explicit type conversion

A programmer often needs to change data types to perform an operation. Ex: A program should read in two values using `input()` and sum the values. Remember `input()` reads in values as strings. A programmer can use **explicit type conversion** to convert one data type to another.

- **int()** converts a data type to an integer. Any fractional part is removed. Ex: `int(5.9)` produces `5`.
- **float()** converts a data type to a float. Ex: `float(2)` produces `2.0`.
- **str()** converts a data type to a string. Ex: `str(3.14)` produces `"3.14"`.

CHECKPOINT

Example: Ordering pizza

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-2-type-conversion)

CONCEPTS IN PRACTICE

Example: Ordering pizza

Consider the example above.

4. Which function converts the input number of slices to a data type that can be used in the calculation?
   a. `float()`
   b. `input()`
   c. `int()`

5. How could line 3 be changed to improve the program overall?
   a. Use `float()` instead of `int()`.
   b. Add `1` to the result of `int()`.
   c. Add `str()` around `int()`.

CONCEPTS IN PRACTICE

Using int(), float(), and str()

Given `x = 4.5` and `y = int(x)`, what is the value of each expression?

6. y
   a. `4`
   b. `5`

7. `str(x)`
   a. `4.5`
   b. `"4.5"`

8. `float(y)`
   a. `4.0`
   b. `4.5`

TRY IT

Grade average

The following program computes the average of three predefined exam grades and prints the average twice. Improve the program to read the three grades from input and print the average first as a float, and

then as an integer, using explicit type conversion. Ignore any differences that occur due to rounding.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-2-type-conversion)

TRY IT

Cups of water

The following program should read in the ounces of water the user drank today and compute the number of cups drank and the number of cups left to drink based on a daily goal. Assume a cup contains 8 ounces. Fix the code to calculate `cups_drank` and `cups_left` and match the following:

- `ounces` is an integer representing the ounces the user drank.
- `cups_drank` is a float representing the number of cups of water drank.
- `cups_left` is an integer representing the number of cups of water left to drink (rounded down) out of the daily goal of 8 cups.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-2-type-conversion)

TRY IT

Product as float

The following program reads two integers in as strings. Calculate the product of the two integers, and print the result as a float.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-2-type-conversion)

2.3 Mixed data types

Learning objectives

By the end of this section you should be able to

- Identify the data types produced by operations with integers, floats, and strings.
- Use operators and type conversions to combine integers, floats, and strings.

Combining integers and floats

Programmers often need to combine numbers of different data types. Ex: A program computes the total for an online shopping order:

```
quantity = int(input())
price = float(input())
total = quantity * price
print(total)
```

`quantity` is an integer, and `price` is a float. So what is the data type of `total`? For input `3` and `5.0`, `total` is

a float, and the program prints `15.0`.

Combining an integer and a float produces a float. A float is by default printed with at least one figure after the decimal point and has as many figures as needed to represent the value. Note: Division using the / operator always produces a float.

CHECKPOINT

Operations combining integers and floats

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-3-mixed-data-types)

CONCEPTS IN PRACTICE

Operations combining integers and floats

1. `8 * 0.25`
   a. `2`
   b. `2.0`

2. `2 * 9`
   a. `18`
   b. `18.0`

3. `20 / 2`
   a. `10`
   b. `10.0`

4. `7 / 2`
   a. `3.0`
   b. `3.5`

5. `12.0 / 4`
   a. `3`
   b. `3.0`

6. `8 - 1.0`
   a. `7.0`
   b. `7`

7. `5 - 0.25`
   a. `4.5`
   b. `4.75`

Combining numeric types and strings

Easy type conversion in Python can lead a programmer to assume that any data type can be combined with another. Ex: Noor's program reads in a number from input and uses the number in a calculation. This results in

an error in the program because the `input()` function by default stores the number as a string. Strings and numeric data types are incompatible for addition, subtraction, and division. One of the operands needs to be explicitly converted depending on the goal of arithmetic or string concatenation.

The * operator also serves as the repetition operator, which accepts a string operand and an integer operand and repeats the string. Ex: `"banjo" * 3` produces `"banjobanjobanjo"`.

CHECKPOINT

Adding a string and an integer

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-3-mixed-data-types)

CONCEPTS IN PRACTICE

Operations combining numeric types and strings

8. `int('34') + 56`
   a. `'3456'`
   b. `90`
   c. `'90'`

9. `str(12) + ' red roses'`
   a. `'12 red roses'`
   b. `'12'`
   c. `Error`

10. `'50' * 3`
    a. `'150'`
    b. `150`
    c. `'505050'`

11. `str(5.2) + 7`
    a. `12.2`
    b. `'12.2'`
    c. `Error`

12. `80.0 + int('100')`
    a. `180`
    b. `180.0`
    c. `'180'`

13. `str(3.14) + '159'`
    a. `162.14`
    b. `'3.14159'`
    c. `Error`

14. `2.0 * 'this'`

a. `'this'`
b. `'thisthis'`
c. `Error`

TRY IT

After the point

Write a program that reads in a string of digits that represents the digits after the decimal point of a number, num. Concatenate the input string together with `'.'` and `num`, and print the result. Ex: If input is `345`, the program will print `2.345`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-3-mixed-data-types)

TRY IT

Print n times

Write a program that reads in two strings, `str1` and `str2`, and an integer, `count`. Concatenate the two strings with a space in between and a newline (`"\n"`) at the end. Print the resulting string `count` times.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-3-mixed-data-types)

2.4 Floating-point errors

Learning objectives

By the end of this section you should be able to

- Explain numerical inaccuracies related to floating-point representation.
- Use the `round()` function to mitigate floating-point errors in output.

Floating-point errors

Computers store information using 0's and 1's. All information must be converted to a string of 0's and 1's. Ex: 5 is converted to 101. Since only two values, 0 or 1, are allowed the format is called binary.

Floating-point values are stored as binary by Python. The conversion of a floating point number to the underlying binary results in specific types of floating-point errors.

A **round-off error** occurs when floating-point values are stored erroneously as an approximation. The difference between an approximation of a value used in computation and the correct (true) value is called a round-off error.

Ex: Storing the float $(0.1)_{10}$ results in binary values that actually produce $(0.1000000000000000055511151231257827021181583404541015625)_{10}$ when converted back, which is not

exactly equal to $(0.1)_{10}$.

```
# Print floats with 30 decimal places
print(f'{0.1:.30f}') # prints 0.1
print(f'{0.2:.30f}') # prints 0.2
print(f'{0.4:.30f}') # prints 0.4
```

```
0.100000000000000005551115123126
0.200000000000000011102230246252
0.400000000000000022204460492503
```

Table 2.1 Round-off error. (The example above shows a formatted string or f-string, which are introduced in the Objects chapter.)

An **overflow error** occurs when a value is too large to be stored. The maximum and minimum floating-point values that can be represented are 1.8×10^{308} and -1.8×10^{308}, respectively. Attempting to store a floating-point value outside the range $\left(-1.8 \times 10^{308}, 1.8 \times 10^{308}\right)$ leads to an overflow error.

Below, 3.0^{256} and 3.0^{512} can be represented, but 3.0^{1024} is too large and causes an overflow error.

```
print('3.0 to the power of 256 =',
3.0**256)
print('3.0 to the power of 512 = ',
3.0**512)
print('3.0 to the power of 1024 = ',
3.0**1024)
```

```
3.0 to the power of 256 =
1.3900845237714473e+122
3.0 to the power of 512 =
1.9323349832288915e+244
3.0 to the power of 1024 =
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    print('3.0 to the power of 1024 = ',
3.0**1024)
OverflowError: (34, 'Numerical result out
of range')
```

Table 2.2 Overflow error.

CONCEPTS IN PRACTICE

Floating-point errors

For each situation, which error occurs?

1. The statement `result = 2.0 * (10.0 ** 500)` assigns the variable result with too large of a value.
   a. round-off
   b. overflow

2. `0.12345678901234567890123456789 * 0.1` produces `0.012345678901234568430878013601`.
   a. round-off
   b. overflow

Floating point round() function

Python's **round()** function is used to round a floating-point number to a given number of decimal places. The function requires two arguments. The first argument is the number to be rounded. The second argument decides the number of decimal places to which the number is rounded. If the second argument is not provided, the number will be rounded to the closest integer. The `round()` function can be used to mitigate floating-point errors.

Ex:

- `round(2.451, 2) = 2.45`
- `round(2.451) = 2`

CHECKPOINT

Examples of round() function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-4-floating-point-errors)

CONCEPTS IN PRACTICE

Examples of round() function

3. What is the output of `round(6.6)`?
   a. `6`
   b. `6.6`
   c. `7`

4. What is the output of `round(3.5, 2)`?
   a. `3`
   b. `3.5`
   c. `3.50`

5. What is the output of `round(12)`?
   a. `12.0`
   b. `12`
   c. `12.00`

6. What is the output of `round(0.1, 1)`?
   a. `0.1`
   b. `0.10`
   c. `0.1000000000000000055511151231257827021181583404541015625`

TRY IT

Inaccurate tips

The following code calculates the tip amount, given a bill amount and the tip ratio. Experiment with the

following bill amounts and tip ratios and see if any inaccuracies may result in calculating the tip amount.

- bill amount: 22.70 and 33.33
- tip ratio: 0.15, 0.18, and 0.20

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-4-floating-point-errors)

TRY IT

Area of a triangle

Complete the following steps to calculate a triangle's area, and print the result of each step. The area of a triangle is $\frac{bh}{2}$, where b is the base and h is the height.

1. Calculate the area of a triangle with base = 7 and height = 3.5.
2. Round the triangle's area to one decimal place.
3. Round the triangle's area to the nearest integer value.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-4-floating-point-errors)

2.5 Dividing integers

Learning objectives

By the end of this section you should be able to

- Evaluate expressions that involve floor division and modulo.
- Use the modulo operator to convert between units of measure.

Division and modulo

Python provides two ways to divide numbers:

- **True division** (`/`) converts numbers to floats before dividing. Ex: `7 / 4` becomes `7.0 / 4.0`, resulting in `1.75`.
- **Floor division** (`//`) computes the quotient, or the number of times divided. Ex: `7 // 4` is `1` because 4 goes into 7 one time, remainder 3. The **modulo operator** (`%`) computes the remainder. Ex: `7 % 4` is `3`.

Note: The `%` operator is traditionally pronounced "mod" (short for "modulo"). Ex: When reading `7 % 4` out loud, a programmer would say "seven mod four."

CHECKPOINT

Quotient and remainder

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-5-dividing-integers)

CONCEPTS IN PRACTICE

Division and modulo

What is the value of each expression?

1. `13 / 5`
   a. `2`
   b. `2.6`
   c. `3`

2. `13 % 5`
   a. `2`
   b. `2.6`
   c. `3`

3. `1 // 4`
   a. `0`
   b. `0.25`
   c. `1`

4. `2 % 0`
   a. `0`
   b. `2`
   c. `Error`

Unit conversions

Division is useful for converting one unit of measure to another. To convert centimeters to meters, a variable is divided by 100. Ex: 300 centimeters divided by 100 is 3 meters.

Amounts often do not divide evenly as integers. 193 centimeters is 1.93 meters, or 1 meter and 93 centimeters. A program can use floor division and modulo to separate the units:

- The quotient, 1 meter, is `193 // 100`.
- The remainder, 93 centimeters, is `193 % 100`.

Programs often use floor division and modulo together. If one line of code floor divides by `m`, the next line will likely modulo by `m`. The unit `m` by which an amount is divided is called the **modulus**. Ex: When converting centimeters to meters, the modulus is 100.

CHECKPOINT

Money and time

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-5-dividing-integers)

CONCEPTS IN PRACTICE

Unit conversions

5. What is the modulus for converting minutes to hours?
   a. 40
   b. 60
   c. 280

6. A program has the line `pounds = ounces // 16`. What is likely the next line of code?
   a. `ounces = ounces % 16`
   b. `pounds = ounces % 16`
   c. `ounces = ounces - pounds * 16`

TRY IT

Arrival time

Having a mobile device can be a lifesaver on long road trips. Programs like Google Maps find the shortest route and estimate the time of arrival. The time of arrival is based on the current time plus how long the trip will take.

Write a program that (1) inputs the current time and estimated length of a trip, (2) calculates the time of arrival, and (3) outputs the results in hours and minutes. Your program should use the following prompts (user input in bold):

```
13
    Current minute (0-59)? 25
    Trip time (in minutes)? 340

    Current hour (0-23)? 13
    Current minute (0-59)? 25
    Trip time (in minutes)? 340
```

In this example, the current time is 13:25 (1:25pm). The trip time is 340 minutes (5 hours and 40 minutes). 340 minutes after 13:25 is 19:05 (7:05pm). Your program should output the result in this format:

```
    Arrival hour is 19
    Arrival minute is 5
```

The arrival hour must be between 0 and 23. Ex: Adding 120 minutes to 23:00 should be 1:00, not 25:00. The arrival minute must be between 0 and 59. Ex: Adding 20 minutes to 8:55 should be 9:15, not 8:75.

Hint: Multiply the current hour by 60 to convert hours to minutes. Then, calculate the arrival time, in total minutes, as an integer.

Your code must not use Python keywords from later chapters, such as `if` or `while`. The solution requires

only addition, multiplication, division, and modulo.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-5-dividing-integers)

TRY IT

Change machine

Self-checkout aisles are becoming increasingly popular at grocery stores. Customers scan their own items, and a computer determines the total purchase amount. Customers who pay in cash insert dollar bills, and a machine automatically dispenses change in coins.

That's where this program comes into the story. Your task is to calculate how many of each coin to dispense. Your program should use the following prompts (user input in bold):

```
18.76
    Cash payment? 20

    Total amount? 18.76
    Cash payment? 20
```

You may assume that the cash paid will always be a whole number (representing dollar bills) that is greater than or equal to the total amount. The program should calculate and output the amount of change due and how many dollars, quarters, dimes, nickels, and pennies should be dispensed:

```
    Change Due $1.24

     Dollars: 1
    Quarters: 0
       Dimes: 2
     Nickels: 0
     Pennies: 4
```

Hint: Calculate the total change, in cents, as an integer. Use the `round()` function to avoid floating-point errors.

Your code must not use Python keywords from later chapters, such as `if` or `while`. The solution requires only subtraction, multiplication, division, and modulo.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-5-dividing-integers)

2.6 The math module

Learning objectives

By the end of this section you should be able to

- Distinguish between built-in functions and math functions.
- Use functions and constants defined in the math module.

Importing modules

Python comes with an extensive standard library (https://openstax.org/r/100pythlibrary) of modules. A **module** is previously written code that can be imported in a program. The **import statement** defines a variable for accessing code in a module. Import statements often appear at the beginning of a program.

The standard library also defines built-in functions such as `print()`, `input()`, and `float()`. A **built-in function** is always available and does not need to be imported. The complete list of built-in functions (https://openstax.org/r/100builtin) is available in Python's official documentation.

A commonly used module in the standard library is the math module (https://openstax.org/r/100mathmodule). This module defines functions such as `sqrt()` (square root). To call `sqrt()`, a program must `import math` and use the resulting `math` variable followed by a dot. Ex: `math.sqrt(25)` evaluates to `5.0`.

The following program imports and uses the math module, and uses built-in functions for input and output.

EXAMPLE 2.1

Calculating the distance between two points

```
import math

x1 = float(input("Enter x1: "))
y1 = float(input("Enter y1: "))
x2 = float(input("Enter x2: "))
y2 = float(input("Enter y2: "))

distance = math.sqrt((x2-x1)**2 + (y2-y1)**2)
print("The distance is", distance)
```

CHECKPOINT

Importing math in a Python shell

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-6-the-math-module)

CONCEPTS IN PRACTICE

Built-in functions and math module

1. In the above example, when evaluating math, why did the interpreter raise a `NameError`?
   a. The math module was not available.

b. The variable pie was spelled incorrectly.
c. The math module was not imported.

2. Which of these functions is builtin and does not need to be imported?
a. `log()`
b. `round()`
c. `sqrt()`

3. Which expression results in an error?
a. `math.abs(1)`
b. `math.log(1)`
c. `math.sqrt(1)`

Mathematical functions

Commonly used math functions and constants are shown below. The complete math module listing (https://openstax.org/r/100mathmodule) is available in Python's official documentation.

Constant	Value	Description
`math.e`	$e = 2.71828\ldots$	Euler's number: the base of the natural logarithm.
`math.pi`	$\pi = 3.14159\ldots$	The ratio of the circumference to the diameter of a circle.
`math.tau`	$\tau = 6.28318\ldots$	The ratio of the circumference to the radius of a circle. Tau is equal to 2π.

Table 2.3 Example constants in the math module.

Function	Description	Examples
Number-theoretic		
`math.ceil(x)`	The ceiling of x: the smallest integer greater than or equal to x.	`math.ceil(7.4)` → `8` `math.ceil(-7.4)` → `-7`
`math.floor(x)`	The floor of x: the largest integer less than or equal to x.	`math.floor(7.4)` → `7` `math.floor(-7.4)` → `-8`
Power and logarithmic		
`math.log(x)`	The natural logarithm of x (to base e).	`math.log(math.e)` → `1.0` `math.log(0)` → `ValueError: math domain error`

Table 2.4 Example functions in the math module.

Function	Description	Examples
`math.log(x, base)`	The logarithm of x to the given base.	`math.log(8, 2)` → `3.0` `math.log(10000, 10)` → `4.0`
`math.pow(x, y)`	x raised to the power y. Unlike the ** operator, `math.pow()` converts x and y to type float.	`math.pow(3, 0)` → `1.0` `math.pow(3, 3)` → `27.0`

`math.sqrt(x)`	The square root of x.	`math.sqrt(9)` → `3.0`
		`math.sqrt(-9)` →
		`ValueError: math domain error`

Trigonometric		
`math.cos(x)`	The cosine of x radians.	`math.cos(0)` → `1.0` `math.cos(math.pi)` → `-1.0`
`math.sin(x)`	The sine of x radians.	`math.sin(0)` → `0.0` `math.sin(math.pi/2)` → `1.0`

`math.tan(x)`	The tangent of x radians.	`math.tan(0)` → `0.0`
		`math.tan(math.pi/4)` →
		`0.999`
		(Round-off error; the result should be 1.0.)

Table 2.4 Example functions in the math module.

CONCEPTS IN PRACTICE

Using math functions and constants

4. What is the value of `math.tau/2`?
   a. approximately `2.718`
   b. approximately `3.142`
   c. approximately `6.283`

5. What is the value of `math.sqrt(100)`?
   a. the float `10.0`
   b. the integer `10`
   c. `ValueError: math domain error`

6. What is πr^2 in Python syntax?
   a. `pi * r**2`

b. `math.pi * r**2`
c. `math.pi * r*2`

7. Which expression returns the integer 27?
a. `3 ** 3`
b. `3.0 ** 3`
c. `math.pow(3, 3)`

TRY IT

Quadratic formula

In algebra, a quadratic equation is written as $ax^2 + bx + c = 0$. The coefficients *a*, *b*, and *c* are known values. The variable *x* represents an unknown value. Ex: $2x^2 + 3x - 5 = 0$ has the coefficients $a = 2$, $b = 3$, and $c = -5$. The quadratic formula provides a quick and easy way to solve a quadratic equation for *x*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The plus-minus symbol indicates the equation has two solutions. However, Python does not have a plus-minus operator. To use this formula in Python, the formula must be separated:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Write the code for the quadratic formula in the program below. Test your program using the following values for *a*, *b*, and *c*:

Provided input			Expected output	
a	b	c	x1	x2
1	0	-4	2.0	-2.0
1	2	-3	1.0	-3.0
2	1	-1	0.5	-1.0

Table 2.5

Provided input			Expected output
0	1	1	division by zero
1	0	1	math domain error

Table 2.5

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-6-the-math-module)

TRY IT

Cylinder formulas

In geometry, the surface area and volume of a right circular cylinder can be computed as follows:

$A = 2\pi rh + 2\pi r^2$

$V = \pi r^2 h$

Write the code for these two formulas in the program below. Hint: Your solution should use both `math.pi` and `math.tau`. Test your program using the following values for r and h:

Provided input		Expected output	
r	h	area	volume
0	0	0.0	0.0
1	1	12.57	3.14
1	2	18.85	6.28
2.5	4.8	114.67	94.25
3.1	7.0	196.73	211.33

Table 2.6

If you get an error, try to look up what that error means.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-6-the-math-module)

2.7 Formatting code

Learning objectives

By the end of this section you should be able to

- Identify good spacing for expressions and statements.
- Write multi-line statements using implicit line joining.

Recommended spacing

Most spaces in Python code are ignored when running programs; however, spaces at the start of a line are very important. The following two programs are equivalent:

- Good spacing:

```
name = input("Enter someone's name: ")
place = input("Enter a famous place: ")
print(name, "should visit", place + "!")
```

- Poor spacing:

```
name=input ("Enter someone's name: " )
place =input("Enter a famous place: ")
print( name,"should visit" , place+ "!")
```

One might argue that missing or extra spaces do not matter. After all, the two programs above run exactly the same way. However, the "poor spacing" version is more difficult to read. Code like `name=input` and `place+` might lead to confusion.

Good programmers write code that is as easy to read as possible. That way, other programmers are more likely to understand the code. To encourage consistency, the Python community has a set of guidelines about where to put spaces and blank lines, what to name variables, how to break up long lines, and other important topics.

PYTHON STYLE GUIDE

PEP 8 (https://openstax.org/r/100PEP8) is the official style guide for Python. **PEP** stands for Python Enhancement Proposal. Members of the Python community write PEPs to document best practices and propose new features. The table below is based on guidelines from PEP 8 under the heading Whitespace in Expressions and Statements (https://openstax.org/r/100whitespace).

Guideline	Example	Common Mistakes
Parentheses: no space before or after.	`print("Go team!")`	`print ("Go team!")` `print( "Go team!" )`
Commas: no space before, one space after.	`print("Hello", name)`	`print("Hello" , name)` `print("Hello",name)`

Assignment: one space before and after the =.	`name = input("Your name? ")`	`name=input("Your name? ")`
		`name= input("Your name? ")`
		`name =input("Your name? ")`

Concatenation: one space before and after the +.	`print("Hi", name + "!")`	`print("Hi", name+"!")`
		`print("Hi", name+ "!")`
		`print("Hi", name +"!")`

Arithmetic: use space to show lower precedence.	`x**2 + 5*x - 8`	`x ** 2 + 5 * x - 8`
		`x ** 2+5 * x-8`
		`x**2+5*x-8`

Table 2.7 Guidelines for spaces.

CONCEPTS IN PRACTICE

Recommended spacing

1. Which statement is formatted properly?
   a. `name = input("What is your name? ")`
   b. `name = input ("What is your name? ")`
   c. `name = input( "What is your name? " )`

2. Which statement is formatted properly?
   a. `name=name+"!"`
   b. `name = name+"!"`
   c. `name = name + "!"`

3. Which statement is formatted properly?
   a. `print("Hello",name)`
   b. `print("Hello", name)`
   c. `print("Hello " , name)`

4. Which expression is formatted properly?

a. `b**2 - 4*a*c`
b. `b ** 2 - 4 * a * c`
c. `b**2 - 4*a * c`

Automatic concatenation

Long strings make Python programs difficult to read. Ex: This program prints the first sentence of the US Declaration of Independence (https://openstax.org/r/100declaration):

print("The unanimous Declaration of the thirteen united States of America, When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.")

PEP 8 recommends that each line of code be less than 80 characters long. That way, programmers won't need to scroll horizontally to read the code. The above program can be rewritten by breaking up the original string:

```
print("The unanimous Declaration of the thirteen united States of "
      "America, When in the Course of human events, it becomes "
      "necessary for one people to dissolve the political bands "
      "which have connected them with another, and to assume among "
      "the powers of the earth, the separate and equal station to "
      "which the Laws of Nature and of Nature's God entitle them, a "
      "decent respect to the opinions of mankind requires that they "
      "should declare the causes which impel them to the separation.")
```

For convenience, Python automatically concatenates multiple strings. The + operator is not required in this situation.

CHECKPOINT

String concatenation

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-7-formatting-code)

CONCEPTS IN PRACTICE

String literal concatenation

5. Which line prints the word `"grandmother"`?
   a. `print(grandmother)`
   b. `print("grand" "mother")`
   c. `print("grand", "mother")`

6. What string is equivalent to `"Today is" "a holiday"`?

a. `'Today isa holiday'`
b. `'Today is a holiday'`
c. `'Today is" "a holiday'`

7. If name is `"Ada"`, what does `print("Hello," name)` output?
a. `Hello,Ada`
b. `Hello, Ada`
c. `SyntaxError`

Multi-line statements

Most statements in a Python program need only one line of code. But occasionally longer statements need to span multiple lines. Python provides two ways to write multi-line statements:

- Explicit line joining, using \ characters:

```
decl = "The unanimous Declaration of the thirteen united States of " \
       "America, When in the Course of human events, it becomes " \
       "necessary for one people to dissolve the political bands..."
```

- Implicit line joining, using parentheses:

```
decl = ("The unanimous Declaration of the thirteen united States of "
       "America, When in the Course of human events, it becomes "
       "necessary for one people to dissolve the political bands...")
```

Implicit line joining is more common, since many statements and expressions use parentheses anyway. PEP 8 recommends avoiding the use of explicit line joining whenever possible.

CONCEPTS IN PRACTICE

Multi-line statements

8. Which character is used for explicit line joining?
a. `/`
b. `\`
c. `|`

9. What is the best way to print a very long string?
a. Break up the string into multiple smaller strings.
```
print("..." # first part of string
"..." # next part of string
"...")
```
b. Print the string using multiple print statements.
```
print("...") # first part of string
print("...") # next part of string
print("...")
```

c. Assign the string to a variable and print the variable.

```
text = "..." # the entire string
print(text)
```

10. Which example consists of two statements?

a.
```
print("Happy "
"New Year")
```

b.
```
saying = ("Happy "
"New Year")
```

c.
```
saying= "Happy "
"New Year"
```

TRY IT

Spaced out

The following code works correctly but is formatted poorly. In particular, the code does not include spaces recommended by PEP 8. Furthermore, two of the lines are about 90 characters long. Reformat the code to follow the guidelines in this section. Be careful not to change the behavior of the code itself.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-7-formatting-code)

TRY IT

Five quotes

Write a program that prints the following five quotes (source: BrainyQuote (https://openstax.org/r/100brainyquote)) from Guido van Rossum, the creator of Python. Your program should have exactly five print statements, one for each quote:

```
    1. "If you're talking about Java in particular, Python is about the best fit
you can get amongst all the other languages. Yet the funny thing is, from a
language point of view, JavaScript has a lot in common with Python, but it is sort
of a restricted subset."
    2. "The second stream of material that is going to come out of this project is
a programming environment and a set of programming tools where we really want to
focus again on the needs of the newbie. This environment is going to have to be
extremely user-friendly."
    3. "I have this hope that there is a better way. Higher-level tools that
actually let you see the structure of the software more clearly will be of
tremendous value."
    4. "Now, it's my belief that Python is a lot easier than to teach to students
programming and teach them C or C++ or Java at the same time because all the
details of the languages are so much harder. Other scripting languages really don't
work very well there either."
```

```
    5. "I would guess that the decision to create a small special purpose language
or use an existing general purpose language is one of the toughest decisions that
anyone facing the need for a new language must make."
```

Notice that all of these lines are longer than 80 characters, and some contain single quote marks. Format the code using multi-line statements and escape sequences as necessary.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-7-formatting-code)

2.8 Python careers

Learning objectives

By the end of this section you should be able to

- Summarize how Python is used in fields other than CS and IT.
- Describe two different kinds of applications made with Python.

Fields and applications

Learning Python opens the door to many programming-related careers. Example job titles include software engineer, data scientist, web developer, and systems analyst. These jobs often require a degree in computer science (CS), information technology (IT), or a related field. However, programming is not limited to these fields and careers.

Many professionals use Python to support the work they do. Python programs can automate tasks and solve problems quickly. Table 2.8 shows a few examples of Python outside of computing fields. Knowing how to program is a useful skill that can enhance any career.

Python is a versatile language that supports many kinds of applications. Table 2.9 shows a few examples of programs that can be written in Python. Given Python's usefulness and popularity, Python is a great language to learn. A supportive community of professionals and enthusiasts is ready to help.

Field	Example use of Python
Business	An accountant writes a Python program to generate custom sales reports.
Education	A teacher writes a Python program to organize homework submissions.
Fine arts	An artist writes a Python program to operate an interactive art display.
Humanities	A linguist writes a Python program to analyze changes in slang word usage.
Science	A biologist writes a Python program to analyze DNA sequences for cancer.

Table 2.8 Python outside of CS and IT.

Application	Example use of Python
Artificial intelligence	An engineer develops models to support image and voice recognition. Ex: TensorFlow library (https://openstax.org/r/100tensorflow).
Data visualization	A statistician creates charts to make sense of large amounts of data. Ex: Matplotlib library (https://openstax.org/r/100matplotlib).
General purpose	Most programs in this book are general. Ex: Read inputs, perform calculations, print results.
Scientific computing	Python is very useful for conducting experiments and analyzing data. Ex: The SciPy project (https://openstax.org/r/100scipyproject).
Web development	Python can run interactive websites. Ex: Instagram is built with Django (https://openstax.org/r/100django), a Python framework.

Table 2.9 Applications built with Python.

CONCEPTS IN PRACTICE

Fields and applications

1. Which of the following fields use Python to support their work?
   a. geography
   b. health care
   c. political science
   d. all of the above

2. Which of the following Python libraries creates charts and plots?
   a. Matplotlib
   b. SciPy
   c. TensorFlow

3. Which of the following applications were built with Python?
   a. Facebook
   b. Instagram
   c. WhatsApp

EXPLORING FURTHER

For more examples of applications that can be built with Python, see "Top 12 Fascinating Python Applications in Real-World" (https://openstax.org/r/100top12apps) by Rohit Sharma. For more information about Python related careers, see "What Does a Python Developer Do?" (https://openstax.org/r/100careers) in BrainStation's career guide.

2.9 Chapter summary

Highlights from this chapter include:

- Expressions and statements can be run interactively using a shell.
- Input strings can be converted to other types. Ex: `int(input())`.
- Strings can be concatenated with other types. Ex: `"$" + str(cost)`.
- Floats are subject to round-off and overflow errors.
- Integers can be divided exactly using `//` and `%`.
- Modules like `math` provide many useful functions.
- Formatting long lines helps improve readability.

At this point, you should be able to write programs that ask for input of mixed types, perform mathematical calculations, and output results with better formatting. The programming practice below ties together most topics presented in the chapter.

Function	Description
`abs(x)`	Returns the absolute value of x.
`int(x)`	Converts x (a string or float) to an integer.
`float(x)`	Converts x (a string or integer) to a float.
`str(x)`	Converts x (a float or integer) to a string.
`round(x, ndigits)`	Rounds x to `ndigits` places after the decimal point. If `ndigits` is omitted, returns the nearest integer to x.
Operator	**Description**
`s * n` (Repetition)	Creates a string with n copies of `s`. Ex: `"Ha" * 3` is `"HaHaHa"`.
`x / y` (Real division)	Divides x by y and returns the entire result as a float. Ex: `7 / 4` is `1.75`.
`x // y` (Floor division)	Divides x by y and returns the quotient as an integer. Ex: `7 // 4` is `1`.
`x % y` (Modulo)	Divides x by y and returns the remainder as an integer. Ex: `7 % 4` is `3`.

Table 2.10 Chapter 2 reference.

TRY IT

Baking bread

The holidays are approaching, and you need to buy ingredients for baking many loaves of bread. According to a recipe by King Arthur Flour (https://openstax.org/r/100kingarthurflr), you will need the following ingredients for each loaf:

- 1 1/2 teaspoons instant yeast
- 1 1/2 teaspoons salt
- 1 1/2 teaspoons sugar
- 2 1/2 cups all-purpose flour
- 2 cups sourdough starter
- 1/2 cup lukewarm water

Write a program that inputs the following variables: `bread_weight` (float), `serving_size` (float), and `num_guests` (int). The output will look like the following:

Note: The measures the program comes up with are exact, but to bake, the baker would have to use some approximation. Ex: 9.765625 cups all-purpose flour really means 9 and 3/4 cups.

```
For 25 people, you will need 3.90625 loaves of bread:

  5.859375 teaspoons instant yeast
  5.859375 teaspoons salt
  5.859375 teaspoons sugar
  9.765625 cups all-purpose flour
  7.8125 cups sourdough starter
  1.953125 cups lukewarm water
```

In the above output, `bread_weight` is 16.0 ounces, `serving_size` is 2.5 ounces, and `num_guests` is 25 people. Use these three variables to calculate the number of loaves needed.

Make sure your output matches the above example exactly. Notice that each line of the ingredients begins with two spaces.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-9-chapter-summary)

TRY IT

Tip calculator

Google has a variety of search tricks (https://openstax.org/r/100googletricks) that present users with instant results. If you search on Google for tip calculator (https://openstax.org/r/100tipcalculatr), an interactive tool is included at the top of the results. The goal of this exercise is to implement a similar tip calculator.

Begin by prompting the user to input the following values (user input in bold):

```
43.21
    Percentage to tip: 18
    Number of people: 2

    Enter bill amount: 43.21
    Percentage to tip: 18
    Number of people: 2
```

Then calculate the tip amount and total amount for the bill, based on the user input. Output the results using this format:

```
    Tip amount: $7.78
    Total amount: $50.99

    Tip per person: $3.89
    Total per person: $25.49
```

Your program should output all dollar amounts rounded to two decimal places. The output should be exactly six lines, as shown above. Notice the blank line before each section of the output. Notice also the space before but not after the dollar sign.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/2-9-chapter-summary)

3 Objects

Figure 3.1 credit: modification of work "Port of Melbourne", by Chris Phutully/Flickr, CC BY 2.0

Chapter Outline

Introduction

An **object** is a single unit of data in a Python program. So far, this book has introduced three types of objects: strings, integers, and floats. This chapter takes a closer look at how strings are represented and how integers and floats can be formatted. To better understand what an object actually is, the relationship between variables and objects is emphasized. The chapter also introduces two types of containers: lists and tuples. A **container** is an object that can hold an arbitrary number of other objects. At the end of this chapter, you will be able to solve more complex problems using fewer variables.

3.1 Strings revisited

Learning objectives

By the end of this section you should be able to

- Extract a specific character from a string using an index.
- Use escape sequences to represent special characters.

Indexes

A string is a sequence of zero or more characters. Each character has an index that refers to the character's position. Indexes are numbered from left to right, starting at 0. Indexes are also numbered from right to left,

starting at -1.

CHECKPOINT

String indexes

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-1-strings-revisited)

CONCEPTS IN PRACTICE

String indexes

1. What is the index of the second character in a string?
   a. 1
   b. 2
   c. -2

2. If `s = "Python!"`, what is the value of `s[1] + s[-1]`?
   a. `"P!"`
   b. `"y!"`
   c. `"yn"`

3. If `s = "Python!"`, what type of object is `s[0]`?
   a. character
   b. integer
   c. string

Unicode

Python uses **Unicode**, the international standard for representing text on computers. Unicode defines a unique number, called a **code point**, for each possible character. Ex: `"P"` has the code point 80, and `"!"` has the code point 33.

The built-in `ord()` function converts a character to a code point. Ex: `ord("P")` returns the integer 80. Similarly, the built-in `chr()` function converts a code point to a character. Ex: `chr(33)` returns the string `"!"`.

Unicode is an extension of **ASCII**, the American Standard Code for Information Interchange. Originally, ASCII defined only 128 code points, enough to support the English language. Unicode defines over one million code

points and supports most of the world's written languages.

32 (space)	64 @	96 `
33 !	65 A	97 a
34 "	66 B	98 b
35 #	67 C	99 c
36 $	68 D	100 d
37 %	69 E	101 e
38 &	70 F	102 f
39 '	71 G	103 g
40 (	72 H	104 h
41)	73 I	105 i
42 *	74 J	106 j
43 +	75 K	107 k
44 ,	76 L	108 l
45 -	77 M	109 m
46 .	78 N	110 n
47 /	79 O	111 o
48 0	80 P	112 p
49 1	81 Q	113 q
50 2	82 R	114 r
51 3	83 S	115 s
52 4	84 T	116 t
53 5	85 U	117 u
54 6	86 V	118 v
55 7	87 W	119 w
56 8	88 X	120 x
57 9	89 Y	121 y
58 :	90 Z	122 z
59 ;	91 [	123 {
60 <	92 \	124 |
61 =	93]	125 }
62 >	94 ^	126 ~
63 ?	95 _	127 (delete)

Table 3.1 Character values. This table shows code points 32 to 127 as defined by ASCII and Unicode. Code points 0 to 31 are non-printable characters that were used for telecommunications.

CONCEPTS IN PRACTICE

ord() and chr()

4. What is the code point for the letter A?

a. 1

b. 65

c. 97

5. What value does `ord("0")` return?
   a. `0`
   b. `48`
   c. `Error`

6. What does `chr(126)` return?
   a. `~`
   b. `"~"`
   c. `Error`

Special characters

An **escape sequence** uses a backslash (\) to represent a special character within a string.

Escape sequence	Meaning	Example	Screen output
`\n`	A newline character that indicates the end of a line of text.	`print("Escape\nsequence!")`	`Escape` `sequence!`
`\t`	A tab character; useful for indenting paragraphs or aligning text on multiple lines.	`print("Escape\tsequence!")`	`Escape  sequence!`
`\'`	A single quote; an alternative to enclosing the string in double quotes.	`print('I\'ll try my best!')`	`I'll try my best`
`\"`	A double quote; an alternative to enclosing the string in single quotes.	`print("I heard you said \"Yes\"")`	`I heard you said "Yes"`
`\\`	A backslash character.	`print("This prints a \\")`	`This prints a \`

Table 3.2 Common escape sequences.

CONCEPTS IN PRACTICE

Tabs and newlines

7. Which of the following is an escape sequence?
   a. `t`
   b. `/t`
   c. `\t`

8. Which statement prints a backslash (\) to the screen?
   a. `print(\\)`
   b. `print(\"\")`
   c. `print("\\")`

9. Which statement prints `Enter` and `here` on separate lines?
   a. `print("Enter here")`
   b. `print("Enter" + \n + "here")`
   c. `print("Enter" + "\n" + "here")`

TRY IT

Hopper quote

Grace Hopper (https://openstax.org/r/100gracehopper) (1906–1992) was a famous computer scientist (and rear admiral in the US Navy!) who came up with the idea of machine-independent programming languages. She envisioned a programming language based on English and made many contributions that paved the way for modern programming languages, including Python.

Write a program that prints the following text, including the quotation marks. Your program may not use single quotes (') anywhere in the code. The last line must be indented with a tab character.

```
"To me programming is more than an important practical art.
It is also a gigantic undertaking in the foundations of knowledge."
        -- Grace Hopper
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-1-strings-revisited)

TRY IT

Shift cipher

During the Roman Empire, Julius Caesar (100–44 BCE) used a simple technique to encrypt private messages. Each letter of the message was replaced with the third next letter of the alphabet. Ex: If the message was CAT, the C became F, the A became D, and the T became W, resulting in the message FDW. This technique is known as a shift cipher because each letter is shifted by some amount. In Caesar's case,

the amount was three, but other amounts (besides 0) would work too.

Write a program that prompts the user to input the following two values (example input in bold):

```
Enter a 3-letter word: CAT
Shift by how many letters? 3
```

The program should then shift each letter of the word by the desired amount. Based on the example above, the output would be:

```
The secret message is: FDW
```

Hint: Use the `ord()` function to convert each letter to an integer, add the shift amount to each integer, use the `chr()` function to convert each integer to a character, and concatenate the resulting characters.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-1-strings-revisited)

3.2 Formatted strings

Learning objectives

By the end of this section you should be able to

- Use f-strings to simplify output with multiple values.
- Format numbers with leading zeros and fixed precision.

F-strings

A **formatted string literal** (or **f-string**) is a string literal that is prefixed with `"f"` or `"F"`. A **replacement field** is an expression in curly braces ({}) inside an f-string. Ex: The string `f"Good morning, {first} {last}!"` has two replacement fields: one for a first name, and one for a last name. F-strings provide a convenient way to combine multiple values into one string.

CHECKPOINT

Printing an f-string

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-2-formatted-strings)

CONCEPTS IN PRACTICE

Basic f-strings

1. What is the output of the following code?

```
animal = "dog"
```

```
says = "bark"
print(f"My {animal} says {says} {says} {says}")
```

a. My dog bark bark bark bark
b. My dog says bark bark bark
c. Error

2. What is the output of the following code?

```
temp = "hot"
food = "potato"
print("{temp} {food}")
```

a. {temp} {food}
b. hot potato
c. Error

3. How can the following code be rewritten using an f-string?

```
print(x, "+", y, "=", x + y)
```

a. print(f"x + y = {x+y}")
b. print(f"{x + y} = {x + y}")
c. print(f"{x} + {y} = {x+y}")

Formatting numbers

Programs often need to display numbers in a specific format. Ex: When displaying the time, minutes are formatted as two-digit integers. If the hour is 9 and the minute is 5, then the time is "9:05" (not "9:5").

In an f-string, a replacement field may include a format specifier introduced by a colon. A **format specifier** defines how a value should be formatted for display. Ex: In the string `f"{hour}:{minute:02d}"`, the format specifier for `minute` is `02d`.

Format	Description	Example	Result
`d`	Decimal integer (default integer format).	`f"{12345678:d}"`	`'12345678'`
`,d`	Decimal integer, with comma separators.	`f"{12345678:,d}"`	`'12,345,678'`
`10d`	Decimal integer, at least 10 characters wide.	`f"{12345678:10d}"`	`'  12345678'`
`010d`	Decimal integer, at least 10 characters wide, with leading zeros.	`f"{12345678:010d}"`	`'0012345678'`

Table 3.3 Example format specifiers. The table shows common ways that numbers can be formatted. Many more formatting options are available and described in Python's Format Specification Mini-Language (https://docs.python.org/3/library/string.html#formatspec).

Format	Description	Example	Result
`f`	Fixed-point (default is 6 decimal places).	`f"{math.pi:f}"`	`'3.141593'`
`.4f`	Fixed-point, rounded to 4 decimal places.	`f"{math.pi:.4f}"`	`'3.1416'`
`8.4f`	Fixed-point, rounded to 4 decimal places, at least 8 characters wide.	`f"{math.pi:8.4f}"`	`'  3.1416'`
`08.4f`	Fixed-point, rounded to 4 decimal places, at least 8 characters wide, with leading zeros.	`f"{math.pi:08.4f}"`	`'003.1416'`

Table 3.3 Example format specifiers. The table shows common ways that numbers can be formatted. Many more formatting options are available and described in Python's Format Specification Mini-Language (https://docs.python.org/3/library/string.html#formatspec).

CONCEPTS IN PRACTICE

Formatting numbers

4. What f-string formats a date as `MM/DD/YYYY`?
   a. `f"{month:02d}/{day:02d}/{year}"`
   b. `f"{month:2d}/{day:2d}/{year}"`
   c. `f"{month}/{day}/{year}"`

5. What statement displays the variable `money` rounded to two decimal places?
   a. `print(f"{money:2d}")`
   b. `print(f"{money:2f}")`
   c. `print(f"{money:.2f}")`

6. What format specifier displays a floating-point number with comma separators and rounded to two decimal places?
   a. `.2,f`
   b. `,.2f`
   c. `,d.2f`

TRY IT

Mad lib (f-string)

A mad lib (https://openstax.org/r/100fstring) is a funny story that uses words provided by a user. The following mad lib is based on four words (user input in bold):

```
Enter a name: Buster
Enter a noun: dog
```

```
Enter an adjective: super
Verb ending in -ing: swimming

Buster, the super dog, likes to go swimming.
```

Most of the code for this mad lib is already written. Complete the code below by writing the f-string.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-2-formatted-strings)

TRY IT

Wage calculator

You just landed a part-time job and would like to calculate how much money you will earn. Write a program that inputs the time you start working, the time you stop working, and your hourly pay rate (example input in bold):

```
Starting hour: 9
Starting minute: 30
Stopping hour: 11
Stopping minute: 0
Hourly rate: 15
```

Based on the user input, your program should calculate and display the following results:

```
Worked 9:30 to 11:00
Total hours: 1.5
Payment: $22.50
```

For this exercise, you need to write code that (1) calculates the total payment and (2) formats the three output lines. Use f-strings and format specifiers to display two-digit minutes, one decimal place for hours, and two decimal places for payment. The input code has been provided as a starting point.

Assume the use of a 24-hour clock. Ex: 16:15 is used instead of 4:15pm.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-2-formatted-strings)

3.3 Variables revisited

Learning objectives

By the end of this section you should be able to

- Distinguish between variables, objects, and references.
- Draw memory diagrams with integers, floats, and strings.

References to objects

In Python, every variable refers to an object. The assignment statement `message = "Hello"` makes the variable `message` refer to the object `"Hello"`. Multiple variables may refer to the same object. Ex: `greeting = message` makes `greeting` refer to the same object as `message`. A **memory diagram** shows the relationship between variables and objects.

CHECKPOINT

Example memory diagram

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-3-variables-revisited)

CONCEPTS IN PRACTICE

Variables and objects

1. How many assignment statements are in the above animation?
   a. 2
   b. 3
   c. 4

2. Which of the following best describes the objects assigned?
   a. two float objects
   b. two int objects, one float object
   c. one int object, one float object

3. What symbol is used to show a variable's current value?
   a. an arrow
   b. a small black box
   c. a rounded box

EXPLORING FURTHER

Python Tutor (https://openstax.org/r/100pythontutor) is a free online tool for visualizing code execution. A user can enter any Python code, click Visualize Execution, and then click the Next button to run the code one line at a time. Here is the rating and score example (https://openstax.org/r/100ratingscore) from the animation above.

Python Tutor is also useful for drawing memory diagrams similar to the ones in this book. Before clicking Visualize Execution, change the middle option from "inline primitives, don't nest objects [default]" to "render all objects on the heap (Python/Java)" as shown in the following screenshot:

Figure 3.2

Properties of objects

Every object has an identity, a type, and a value:

- An object's **identity** is a unique integer associated with the object. Generally, this integer refers to the memory location where the object is stored. Once created, an object's identity never changes. The built-in `id()` function returns the object's identity.
- An object's **type** determines the possible values and operations of an object. Ex: Integers and floats can be "divided" using the `/` operator, but strings cannot. The built-in `type()` function returns the object's type.
- An object's **value** represents the current state of the object. Many objects, such as numbers and strings, cannot be modified once created. Some objects, such as lists (introduced later), are designed to be modified.

CHECKPOINT

Identity, type, and value

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-3-variables-revisited)

CONCEPTS IN PRACTICE

id() and type()

4. Which value might be returned by `id(rating)`?
   a. `9793344`
   b. `<class 'float'>`
   c. `10.0`

5. Which value might be returned by `type(rating)`?
   a. `10.0`
   b. `"float"`
   c. `<class 'float'>`

6. What expression returns the value of an object?
   a. `value(rating)`
   b. `rating`
   c. `"rating"`

EXPLORING FURTHER

As shown in a memory diagram, variables and objects are two separate ideas. Calling a function like `id()` or `type()` returns information about an object, not a variable. In fact, a variable doesn't have an identity or a type, as shown in this example:

```
>>> rating = 10  # Integer object somewhere in memory.
>>> type(rating)
<class 'int'>
>>> id(rating)
9793344
>>> rating = "ten"  # String object somewhere else in memory.
>>> type(rating)
<class 'str'>
>>> id(rating)
140690967388272
```

One might incorrectly think that the rating variable's type or identity changes. However, the only thing that changes is which object the rating variable refers to.

TRY IT

Three variables

1. Draw a memory diagram for the following code:

```
a = 1
b = 2
c = b
b = a
a = c
```

2. Run the code on Python Tutor (https://openstax.org/r/100pythruncode) to check your answer.
3. Based on your diagram, answer these questions:
   - What is the final value of a, b, and c?
   - How many integer objects are created?

TRY IT

Different types

1. Draw a memory diagram for the following code:

```
name = "Chocolate"
length = len(name)
```

```
price = 1.99
lower = min(length, price)
product = name
name = name * 2
```

2. Run the code on Python Tutor (https://openstax.org/r/100pythruncode) to check your answer.
3. Based on your diagram, answer these questions:
   - What is the type and value of each object?
   - Which object does each variable reference?

3.4 List basics

Learning objectives

By the end of this section you should be able to

- Use indexes to access individual elements in a list.
- Use indexes to modify individual elements in a list.
- Use `len()` function to find the length of a list.
- Demonstrate that lists can be changed after creation.

Lists

A list object can be used to bundle elements together in Python. A list is defined by using square brackets `[]` with comma separated values within the square brackets. Ex: `list_1 = [1, 2, 4]`.

Empty lists can be defined in two ways:

- `list_1 = []`
- `list_1 = list()`

Lists can be made of elements of any type. Lists can contain integers, strings, floats, or any other type. Lists can also contain a combination of types. Ex: `[2, "Hello", 2.5]` is a valid list.

Python lists allow programmers to change the contents of the list in various ways.

CHECKPOINT

Lists

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-4-list-basics)

CONCEPTS IN PRACTICE

Lists

1. Which is the correct way to make a list of the numbers 3, 4, and 5?
   a. `new_list == [3, 4, 5]`
   b. `new_list = [3, 4, 5]`
   c. `new_list = (3, 4, 5)`

2. Which of the following is not a valid way to specify a list in Python?
   a. `my_list = [2, 2 3, 23]`
   b. `my_list = ["Jimmy", 2, "times"]`
   c. `my_list = ["C", "C++", "Python", "Java", "Rust", "Scala"]`

Using indexes

Individual list elements can be accessed directly using an index. Indexes begin at 0 and end at one less than the length of the sequence. Ex: For a sequence of 50 elements, the first position is 0, and the last position is 49.

The index number is put in square brackets `[]` and attached to the end of the name of the list to access the required element. Ex: `new_list[3]` accesses the 4th element in `new_list`. An expression that evaluates to an integer number can also be used as an index. Similar to strings, negative indexing can also be used to address individual elements. Ex: Index -1 refers to the last element and -2 the second-to-last element.

The `len()` function, when called on a list, returns the length of the list.

EXAMPLE 3.1

List indexes and len() function

The following code demonstrates the use of list indexes and the `len()` function. Line 6 shows the use of the `len()` function to get the length of the list. Line 10 shows how to access an element using an index. Line 14 shows how to modify a list element using an index.

```
1  # Setup a list of numbers
2      num_list = [2, 3, 5, 9, 11]
3      print(num_list)
4
5      # Print the length of the list
6      print("Length: ", len(num_list))
7
8      # Print the 4th element in the list
9      # The number 3 is used to refer to the 4th element
10     print("4th element:", num_list[3])
11
12     # The desired value of the 4th element is actually 7
13     # Update the value of the 4th element to 7
14     num_list[3] = 7
15
16     # The list of the first 5 prime numbers
17     print(num_list)
```

The above code's output is:

```
[2, 3, 5, 9, 11]
Length: 5
4th element: 9
```

```
[2, 3, 5, 7, 11]
```

CONCEPTS IN PRACTICE

List indexes and the len() function

3. Which is the correct way to access the 17th element in a list called `cardList`?
   a. `card_list[17]`
   b. `16)card_list(16)`
   c. `card_list[16]`

4. What is the correct way to modify the element `"Tom"` to `"Tim"` in the following list?

```
name_list = ["Carla", "Monique", "Westin", "Tom"]
```

   a. `name_list[-1] = "Tim"`
   b. `name_list[4] = "Tim"`
   c. `name_list[end] = "Tim"`

TRY IT

List basics

Write a program to complete the following:

1. Create a list with the following elements: 2, 23, 39, 6, -5.
2. Change the third element of the list (index 2) to 35.
3. Print the resulting list and the list's length.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-4-list-basics)

3.5 Tuple basics

Learning objectives

By the end of this section you should be able to

- Describe the features and benefits of a tuple.
- Develop a program that creates and uses a tuple successfully.
- Identify and discuss the mutability of a tuple.

Creating tuples and accessing elements

A **tuple** is a sequence of comma separated values that can contain elements of different types. A tuple must be created with commas between values, and conventionally the sequence is surrounded by parentheses.

Each element is accessed by index, starting with the first element at index 0.

`tuple_1 = (2, 3, 4)`
`print(f'tuple_1: {tuple_1}')`
`print(tuple_1[1])`
`print()`
`data_13 = ('Aimee Perry', 96, [94, 100, 97, 93])`
`print(f'data_13: {data_13}')`
`print(data_13[2])`
	`tuple_1: (2, 3, 4)`
	`3`
	
	`data_13: ('Aimee Perry', 96, [94, 100, 97, 93])`
	`[94, 100, 97, 93]`

Table 3.4 Example tuples.

CONCEPTS IN PRACTICE

Creating tuples and accessing elements

1. Consider the example above. Which accesses `tuple_1`'s first element?
   a. `tuple_1[0]`
   b. `tuple_1[1]`
   c. `tuple_1[2]`

2. Which creates a valid tuple?
   a. `tuple_2 = (42, 26, 13)`
   b. `tuple_3 = (42, 26, 13.5)`
   c. both

3. Which creates a tuple with three elements?
   a. `my_tuple = 'a', 'b', 'c'`
   b. `my_tuple = ['a', 'b', 'c']`
   c. `my_tuple = ('a', 'b', 'c')`

Tuple properties

How do tuples compare to lists? Tuples are ordered and allow duplicates, like lists, but have different mutability. An **immutable** object cannot be modified after creation. A **mutable** object can be modified after creation. Tuples are immutable, whereas lists are mutable.

CHECKPOINT

Mutability of a tuple vs. a list

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-5-tuple-basics)

CONCEPTS IN PRACTICE

Using tuples

4. Consider the final program in the example above. What is the value of `my_tuple`?
   a. `(0.693, 0.414, 3.142)`
   b. `(0.693, 1.414, 3.142)`
   c. `Error`

5. Why does the following code produce an error?

```
tuple_1 = ('alpha', 'bravo', 'charlie')
tuple_1.append('charlie')
```

   a. Append isn't allowed.
   b. Duplicates aren't allowed.
   c. Strings aren't allowed

6. A programmer wants to create a sequence of constants for easy reference that can't be changed anywhere in the program. Which sequence would be the most appropriate?
   a. list
   b. tuple

MUTABILITY AND PERFORMANCE

Tuples are immutable and have a fixed size, so tuples use less memory. Overall, tuples are faster to create and access, resulting in better performance that can be noticeable with large amounts of data.

TRY IT

Creating a tuple from a list

Suppose a programmer wants to create a tuple from a list to prevent future changes. The `tuple()` function creates a tuple from an object like a list. Ex: `my_tuple = tuple(my_list)` creates a tuple from the list `my_list`. Update the program below to create a tuple `final_grades` from the list grades.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-5-tuple-basics)

TRY IT

Creating a tuple with user input

Write a program that reads in two strings and two integers from input and creates a tuple, `my_data`, with the four values.

Given input:

```
x
y
15
20
```

The output is:

```
my_data: ('x', 'y', 15, 20)
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/3-5-tuple-basics)

3.6 Chapter summary

Highlights from this chapter include:

- A **string** is a sequence of values that represents Unicode code points.
- An **index** refers to the position of a value in a sequence (string, list, tuple).
- Positive indexes range from 0 to length–1. Negative indexes range from –1 to –length.
- F-strings are a convenient way to print multiple outputs and format integers and floats.
- Variables refer to objects. Memory diagrams are useful for drawing variables and objects.
- A list object can be used to refer to multiple objects, by index, using the same variable.
- A tuple is similar to a list, but uses parentheses and cannot be changed once created.

Code	Description
`ord(c)`	Gets an integer representing the Unicode code point of a character.
`chr(i)`	Converts a Unicode code point (integer) into a One-character string.
`'\n'`	Escape sequence for the newline character.
`'\t'`	Escape sequence for the tab character.
`f"{number:.02d}"`	Creates a string by formatting an integer to be at least two digits.

Table 3.5 Chapter 3 reference.

Code	Description
`f"{number:.2f}"`	Creates a string by formatting a float to have two decimal places.
`id(object)`	Gets the identity (memory location) of an object.
`type(object)`	Gets the type (class name) of an object.
`my_list = ["a", "b", "c"]`	Creates a list of three strings.
`my_tuple = ("a", "b", "c")`	Creates a tuple of three strings.
`my_list[0]`	Gets the first element (`"a"`) of `my_tuple`.
`my_tuple[-1]`	Gets the last element (`"c"`) of `my_tuple`.

Table 3.5 Chapter 3 reference.

4 Decisions

Figure 4.1 credit: modification of work "Fork In The Road", by Ian Sane/Flickr, CC BY 2.0

Chapter Outline

Introduction

The Python interpreter follows a single path of execution when executing a program. What if a programmer wants to define multiple possible paths? Ex: Instead of always taking the left path, a program uses the path width to decide which path to take. If the left path is wider, take the right path. Else, take the left path.

A **branch** is a group of statements that execute based on a condition. The Expressions chapter introduced expressions. This chapter explores how expressions can be used as conditions to make decisions in programs.

4.1 Boolean values

Learning objectives

By the end of this section you should be able to

- Explain a Boolean value.
- Use bool variables to store Boolean values.
- Demonstrate converting integers, floats, and strings to Booleans.
- Demonstrate converting Booleans to integers, floats, and strings.
- Use comparison operators to compare integers, floats, and strings.

bool data type

People often ask binary questions such as yes/no or true/false questions. Ex: Do you like pineapple on pizza? Ex: True or false: I like pineapple on pizza. The response is a Boolean value, meaning the value is either true or false. The **bool** data type, standing for Boolean, represents a binary value of either true or false. `true` and `false` are keywords, and capitalization is required.

CHECKPOINT

Example: Crosswalk sign

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-1-boolean-values)

CONCEPTS IN PRACTICE

Using Boolean variables

Consider the following code:

```
is_fruit = "True"
is_vegetable = 0
is_dessert = False
```

1. What is the data type of `is_fruit`?
   a. Boolean
   b. integer
   c. string

2. What is the data type of `is_vegetable`?
   a. Boolean
   b. integer
   c. string

3. What is the data type of `is_dessert`?
   a. Boolean
   b. integer
   c. string

4. How many values can a Boolean variable represent?
   a. 2
   b. 4
   c. 8

5. Which is a valid value for a Boolean variable?
   a. `true`
   b. `True`
   c. `1`

6. Suppose the following is added to the code above:

```
is_dessert = 0
print(type(is_dessert))
```

What is the output?

a. `<class 'bool'>`
b. `<class 'int'>`
c. `Error`

Type conversion with bool()

Deciding whether a value is true or false is helpful when writing programs/statements based on decisions. Converting data types to Booleans can seem unintuitive at first. Ex: `Is "ice cream" True`? But the conversion is actually simple.

bool() converts a value to a Boolean value, `True` or `False`.

- `True`: any non-zero number, any non-empty string
- `False`: 0, empty string

CHECKPOINT

Converting integers, floats, and strings using bool()

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-1-boolean-values)

CONCEPTS IN PRACTICE

Converting numeric types and strings to Booleans

7. `bool(0.000)`
   a. `True`
   b. `False`

8. `bool(-1)`
   a. `True`
   b. `False`

9. `bool("")`
   a. `True`
   b. `False`

10. `bool("0")`
    a. `True`
    b. `False`

11. Given input `False`, what is `bool(input())`?

a. `True`
b. `False`

CONCEPTS IN PRACTICE

Converting Booleans to numeric types and strings

Given `is_on = True`, what is the value of each expression?

12. `float(is_on)`
a. `0.0`
b. `1.0`

13. `str(is_on)`
a. `"is_on"`
b. `"True"`

14. `int(is_on)`
a. `0`
b. `1`

Comparison operators

Programmers often have to answer questions like "Is the current user the admin?" A programmer may want to compare a string variable, user, to the string, "admin". **Comparison operators** are used to compare values, and the result is either true or false. Ex: `is_admin = (user == "admin")`. `user` is compared with `"admin"` using the == operator, which tests for equality. The Boolean variable, `is_admin`, is assigned with the Boolean result.

The 6 comparison operators:

- equal to: **==**
- not equal to: **!=**
- greater than: **>**
- less than: **<**
- greater than or equal to: **>=**
- less than or equal to: **<=**

CHECKPOINT

Example: Rolling a d20 in a tabletop game

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-1-boolean-values)

CONCEPTS IN PRACTICE

Comparing values

For each new variable, what is the value of `compare_result`?

15.
```
x = 14
compare_result = (x <= 13)
```

a. True
b. False

16.
```
w = 0
compare_result = (w != 0.4)
```

a. True
b. False

17.
```
v = 4
compare_result = (v < 4.0)
```

a. True
b. False

18.
```
y = 2
compare_result = (y > "ab")
```

a. True
b. False
c. Error

19.
```
z = "cilantro"
compare_result = (z == "coriander")
```

a. True
b. False

20.
```
a = "dog"
compare_result = (a < "cat")
```

a. True
b. False

= VS ==

A common mistake is using = for comparison instead of ==. Ex: `is_zero = num=0` will always assign `is_zero` and `num` with `0`, regardless of num's original value. The = operator performs assignment and will modify the variable. The == operator performs comparison, does not modify the variable, and produces `True` or `False`.

EXPLORING FURTHER

- Unicode Basic Latin Chart (https://openstax.org/r/100unicodelatin)

TRY IT

Friday Boolean

"It's Friday, I'm in love" —from "Friday I'm in Love," a song released by the Cure in 1992.

Write a program that reads in the day of the week. Assign the Boolean variable, `in_love`, with the result of whether the day is Friday or not.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-1-boolean-values)

TRY IT

Even numbers

Write a program that reads in an integer and prints whether the integer is even or not. Remember, a number is even if the number is divisible by 2. To test this use `number % 2 == 0`. Ex: If the input is `6`, the output is `"6 is even: True"`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-1-boolean-values)

4.2 If-else statements

Learning objectives

By the end of this section you should be able to

- Identify which operations are performed when a program with `if` and `if-else` statements is run.
- Identify the components of an `if` and `if-else` statement and the necessary formatting.
- Create an `if-else` statement to perform an operation when a condition is true and another operation otherwise.

if statement

If the weather is rainy, grab an umbrella! People make decisions based on conditions like if the weather is rainy, and programs perform operations based on conditions like a variable's value. Ex: A program adds two

numbers. If the result is negative, the program prints an error.

A **condition** is an expression that evaluates to true or false. An **if statement** is a decision-making structure that contains a condition and a body of statements. If the condition is true, the body is executed. If the condition is false, the body is not executed.

The `if` statement's body must be grouped together and have one level of indentation. The PEP 8 style guide recommends four spaces per indentation level. The Python interpreter will produce an error if the body is empty.

CHECKPOINT

Example: Quantity check

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-2-if-else-statements)

USING BOOLEAN VARIABLES

A Boolean variable already has a value of `True` or `False` and can be used directly in a condition rather than using the equality operator. Ex: `if is_raining == True:` can be simplified to `if is_raining:`.

CONCEPTS IN PRACTICE

Using if statements

1. Given the following, which part is the condition?

```
if age < 12:
  print("Discount for children available")
```

   a. `age`
   b. `age < 12`
   c. `print("Discount for children available")`

2. Given the following, which lines execute if the condition is true?

```
1 print("Have a great day.")
2 if is_raining:
3   print("Don't forget an umbrella!")
4 print("See you soon.")
```

   a. 1, 2, 3
   b. 1, 2, 4
   c. 1, 2, 3, 4

3. Given the following (same as above), which lines execute if the condition is False?

```
1 print("Have a great day.")
```

```
2 if is_raining:
3   print("Don't forget an umbrella!")
4 print("See you soon.")
```

a. 1, 2, 3
b. 1, 2, 4
c. 1, 2, 3, 4

4. Given `num = -10`, what is the final value of `num`?

```
if num < 0:
  num = 25
if num < 100:
  num = num + 50
```

a. `-10`
b. `40`
c. `75`

5. Given input `10`, what is the final value of `positive_num`?

```
positive_num = int(input("Enter a positive number:"))
if positive_num < 0:
  print("Negative input set to 0")
positive_num = 0
```

a. `10`
b. `0`
c. `Error`

if-else statement

An `if` statement defines actions to be performed when a condition is true. What if an action needs to be performed only when the condition is false? Ex: If the restaurant is less than a mile away, we'll walk. Else, we'll drive.

An **else statement** is used with an `if` statement and contains a body of statements that is executed when the `if` statement's condition is false. When an `if-else` statement is executed, one and only one of the branches is taken. That is, the body of the `if` or the body of the `else` is executed. Note: The `else` statement is at the same level of indentation as the `if` statement, and the body is indented.

`if-else` statement template:

```
# Statements before

    if condition:
      # Body
    else:
      # Body

```

```
8 |     # Statements after
```

CHECKPOINT

Example: Trivia question

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-2-if-else-statements)

CONCEPTS IN PRACTICE

Exploring if-else statements

6. Given the following code, the `else` branch is taken for which range of x?

```
if x >= 15:
  # Do something
else:
  # Do something else
```

a. `x >= 15`
b. `x <= 15`
c. `x < 15`

7. Given `x = 40`, what is the final value of y?

```
if x > 30:
  y = x - 10
else:
  y = x + 10
```

a. `30`
b. `40`
c. `50`

8. Given `y = 50`, which is *not* a possible final value of y?

```
if x < 50:
  y = y / 2
else:
  y = y * 2
y = y + 5
```

a. `30`
b. `55`
c. `105`

TRY IT

Improved division

The following program divides two integers. Division by 0 produces an error. Modify the program to read in a new denominator (with no prompt) if the denominator is 0.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-2-if-else-statements)

TRY IT

Converting temperature units

The following program reads in a temperature as a float and the unit as a string: "f" for Fahrenheit or "c" for Celsius.

Calculate `new_temp`, the result of converting temp from Fahrenheit to Celsius or Celsius to Fahrenheit based on unit. Calculate `new_unit`: "c" if unit is "f" and "f" if unit is "c".

Conversion formulas:

- Degrees Celsius = (degrees Fahrenheit - 32) * 5/9
- Degrees Fahrenheit = (degrees Celsius * 5/9) + 32

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-2-if-else-statements)

4.3 Boolean operations

Learning objectives

By the end of this section you should be able to

- Explain the purpose of logical operators.
- Describe the truth tables for `and`, `or`, and `not`.
- Create expressions with logical operators.
- Interpret `if-else` statements with conditions using logical operators.

Logical operator: and

Decisions are often based on multiple conditions. Ex: A program printing if a business is open may check that `hour >= 9` and `hour < 17`. A **logical operator** takes condition operand(s) and produces `True` or `False`.

Python has three logical operators: *and*, *or*, and *not*. The **and** operator takes two condition operands and returns `True` if both conditions are true.

p	q	p and q
True	True	True
True	False	False
False	True	False
False	False	False

Table 4.1 Truth table: p and q.

CHECKPOINT

Example: Museum entry

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-3-boolean-operations)

CONCEPTS IN PRACTICE

Using the and operator

1. Consider the example above. Jaden tries to enter when the capacity is 2500 and there are 2 hours before close. Can Jaden enter?
   a. yes
   b. no

2. Consider the example above. Darcy tries to enter when the capacity is 3000. For what values of `hrs_to_close` will Darcy to be able to enter?
   a. `hrs_to close > 1.0`
   b. no such value

3. Given `is_admin = False` and `is_online = True`, what is the value of `is_admin` and `is_online`?
   a. `True`
   b. `False`

4. Given `x = 8` and `y = 21`, what is the final value of z?

```
if (x < 10) and (y > 20):
  z = 5
else:
  z = 0
```

   a. `0`
   b. `5`

Logical operator: or

Sometimes a decision only requires one condition to be true. Ex: If a student is in the band or choir, they will perform in the spring concert. The **or** operator takes two condition operands and returns `True` if either condition is `true`.

p	q	p or q
True	True	True
True	False	True
False	True	True
False	False	False

Table 4.2 Truth table: p or q.

CHECKPOINT

Example: Streaming prompt

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-3-boolean-operations)

CONCEPTS IN PRACTICE

Using the or operator

5. Given `days = 21` and `is_damaged` is `False`, is the refund processed?

```
if (days < 30) or is_damaged:
  # Process refund
```

a. yes
b. no

6. For what values of age is there no discount?

```
if (age < 12) or (age > 65):
  # Apply student/senior discount
```

a. `age >= 12`
b. `age <= 65`
c. `(age >= 12)` and `(age <= 65)`

7. Given `a = 9` and `b = 10`, does the test pass?

```
if (a%2 == 0 and b%2 == 1) or (a%2 == 1 and b%2 == 0):
```

```
  # Test passed
else:
  # Test failed
```

a. yes
b. no

Logical operator: not

If the computer is not on, press the power button. The **not** operator takes one condition operand and returns `True` when the operand is false and returns `False` when the operand is true.

not is a useful operator that can make a condition more readable and can be used to toggle a Boolean's value. Ex: `is_on = not is_on`.

p	not p
`True`	`False`
`False`	`True`

Table 4.3 Truth table: not p.

CHECKPOINT

Example: Diving warning

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-3-boolean-operations)

CONCEPTS IN PRACTICE

Using the not operator

8. Given `x = 13`, what is the value of `not(x < 10)`?

a. `True`
b. `False`

9. Given `x = 18`, is x in the correct range?

```
if not(x > 15 and x < 20):
  # x in correct range
```

a. yes
b. no

10. Given `is_turn = False` and `timer = 65`, what is the final value of `is_turn`?

```
if timer > 60:
  is_turn = not is_turn
```

a. True
b. False

TRY IT

Speed limits

Write a program that reads in a car's speed as an integer and checks if the car's speed is within the freeway limits. A car's speed must be at least 45 mph but no greater than 70 mph on the freeway.

If the speed is within the limits, print "Good driving". Else, print "Follow the speed limits".

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-3-boolean-operations)

4.4 Operator precedence

Learning objectives

By the end of this section you should be able to

- Describe how precedence impacts order of operations.
- Describe how associativity impacts order of operations.
- Explain the purpose of using parentheses in expressions with multiple operators.

Precedence

When an expression has multiple operators, which operator is evaluated first? Precedence rules provide the priority level of operators. Operators with the highest precedence execute first. Ex: `1 + 2 * 3` is `7` because multiplication takes precedence over addition. However, `(1 + 2) * 3` is `9` because parentheses take precedence over multiplication.

Operator	Meaning
()	Parentheses
**	Exponentiation (right associative)
*, /, //, %	Multiplication, division, floor division, modulo
+, -	Addition, subtraction
<, <=, >, >=, ==, !=	Comparison operators

Table 4.4 Operator precedence from highest to lowest.

Operator	Meaning
not	Logical `not` operator
and	Logical `and` operator
or	Logical `or` operator

Table 4.4 Operator precedence from highest to lowest.

CHECKPOINT

Operator precedence

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-4-operator-precedence)

CONCEPTS IN PRACTICE

Precedence rules

Which part of each expression is evaluated first?

1. `x ** 2 + 6 / 3`
   a. `6 / 3`
   b. `x ** 2`
   c. `2 + 6`

2. `not 3 * 5 > 10`
   a. `3 * 5`
   b. `not 3`
   c. `5 > 10`

3. `z == 5 and x / 8 < 100`
   a. `5 and x`
   b. `x / 8`
   c. `8 < 100`

Associativity

What if operators beside each other have the same level of precedence? Associativity determines the order of operations when precedence is the same. Ex: `8 / 4 * 3` is evaluated as `(8/4) * 3` rather than `8 / (4*3)` because multiplication and division are left associative. Most operators are left associative and are evaluated from left to right. Exponentiation is the main exception (noted above) and is right associative: that is, evaluated from right to left. Ex: `2 ** 3 ** 4` is evaluated as `2 ** (3**4)`.

When comparison operators are chained, the expression is converted into the equivalent combination of

comparisons and evaluated from left to right. Ex. `10 < x <= 20` is evaluated as `10 < x` and `x <= 20`.

CHECKPOINT

Operation precedence

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-4-operator-precedence)

CONCEPTS IN PRACTICE

Associativity

How is each expression evaluated?

4. `10 + 3 * 2 / 4`
   a. `10 + (3 * (2 / 4))`
   b. `10 + ((3 * 2) / 4)`
   c. `(10 + 3) * (2 / 4)`

5. `2 * 2 ** 2 ** 3`
   a. `2 * ((2 ** 2) ** 3)`
   b. `2 * (2 ** (2 ** 3))`
   c. `((2*2) ** 2) ** 3`

6. `100 < x > 150`
   a. `100 < x and x < 150`
   b. `100 < x or x > 150`
   c. `100 < x and x > 150`

Enforcing order and clarity with parentheses

Operator precedence rules can be hard to remember. Parentheses not only assert a different order of operations but also reduce confusion.

CHECKPOINT

Using parentheses

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-4-operator-precedence)

CONCEPTS IN PRACTICE

Using parentheses

7. Consider the example above. Why was the evaluation order different from what the programmer wanted?
   a. Equality has precedence over `and`.

b. All operators are evaluated right to left.
c. Order is random when parentheses aren't used.

8. Given `x = 8` and `y = 9`, what is the result of the following?
`x + 3 * y - 5`
a. `30`
b. `44`
c. `94`

9. Given `x = 8` and `y = 9`, what is the result of the following?
`(x+3) * (y-5)`
a. `30`
b. `44`
c. `94`

PEP 8 RECOMMENDATIONS: SPACING AROUND OPERATORS

The PEP 8 style guide recommends consistent spacing around operators to avoid extraneous and confusing whitespace.

- Avoid multiple spaces and an unequal amount of whitespace around operators with two operands.
  Avoid: `x= y * 44`
  Better: `x = y * 44`
- Avoid spaces immediately inside parentheses.
  Avoid: `x = ( 4 * y )`
  Better: `x = (4 * y)`
- Surround the following operators with one space: assignment, augment assignment, comparison, Boolean.
  Avoid: `x= y<44`
  Better: `x = y < 44`
- Consider adding whitespace around operators with lower priority.
  Avoid: `x = 5 * z+20`
  Better: `x = 5*z + 20`

4.5 Chained decisions

Learning objectives

By the end of this section you should be able to

- Identify the branches taken in an `if-elif` and `if-elif-else` statement.
- Create a chained decision statement to evaluate multiple conditions.

elif

Sometimes, a complicated decision is based on more than a single condition. Ex: A travel planning site reviews the layovers on an itinerary. If a layover is greater than 24 hours, the site should suggest accommodations. *Else if* the layover is less than one hour, the site should alert for a possible missed connection.

Two separate `if` statements do not guarantee that only one branch is taken and might result in both branches being taken. Ex: The program below attempts to add a curve based on the input test score. If the input is `60`, both `if` statements are incorrectly executed, and the resulting score is 75.

```
score = int(input())
if score < 70:
  score += 10
# Wrong:
if 70 <= score < 85:
  score += 5
```

Chaining decision statements with `elif` allows the programmer to check for multiple conditions. An **elif** (short for else if) statement checks a condition when the prior decision statement's condition is false. An `elif` statement is part of a chain and must follow an `if` (or `elif`) statement.

`if-elif` statement template:

```
# Statements before

if condition:
  # Body
elif condition:
  # Body

# Statements after
```

CHECKPOINT

Example: Livestream features

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-5-chained-decisions)

CONCEPTS IN PRACTICE

Using elif

1. Fill in the blank to execute Body 2 when `condition_1` is false and `condition_2` is true.

```
if condition_1:
  # Body 1
__ condition_2:
  # Body 2
```

a. `if`
b. `elif`

c. `else`

2. Given `x = 42` and `y = 0`, what is the final value of y?

```
if x > 44:
  y += 2
elif x < 50:
  y += 5
```

a. `2`
b. `5`
c. `7`

3. Which conditions complete the code such that if x is less than 0, `Body 1` executes, else if x equals 0, `Body 2` executes.

```
if _________:
  # Body 1
elif _________:
  # Body 2
```

a. `x < 0`
   `x == 0`
b. `x == 0`
   `x < 0`
c. `x <= 0`
   [no condition]

4. Which of the following is a valid chained decision statement?

a.
```
if condition_1:
  # Body 1
elif condition_2:
  # Body 2
```

b.
```
if condition_1:
  # Body 1
  elif condition_2:
    # Body 2
```

c.
```
elif condition_1:
  # Body 1
if condition_2:
  # Body 2
```

5. Given `attendees = 350`, what is the final value of `rooms`?

```
rooms = 1
if attendees >= 100:
  rooms += 3
if attendees <= 200:
  rooms += 7
```

```
elif attendees <= 400:
  rooms += 14
```

a. 4
b. 15
c. 18

if-elif-else statements

Elifs can be chained with an `if-else` statement to create a more complex decision statement. Ex: A program shows possible chess moves depending on the piece type. If the piece is a pawn, show moving forward one (or two) places. Else if the piece is a bishop, show diagonal moves. Else if . . . (finish for the rest of the pieces).

CHECKPOINT

Example: Possible chess moves

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-5-chained-decisions)

CONCEPTS IN PRACTICE

Using elif within if-elif-else statements

6. Given `hour = 12`, what is printed?

```
if hour < 8:
  print("Too early")
elif hour < 12:
  print("Good morning")
elif hour < 13:
  print("Lunchtime")
elif hour < 17:
  print("Good afternoon")
else:
  print("Too late")
```

a. Good morning
b. Lunchtime
c. Good afternoon
d. Too late

7. Where can an `elif` statement be added?

```
_1_
if condition:
  # Body
```

```
  _2_
elif condition:
  # Body
_3_
else:
  # Body
_4_
```

a. 1
b. 2
c. 3
d. 4

8. Given `x = -1` and `y = -2`, what is the final value of y?

```
if x < 0 and y < 0:
  y = 10
elif x < 0 and y > 0:
  y = 20
else:
  y = 30
```

a. `10`
b. `20`
c. `30`

9. How could the following statements be rewritten as a chained statement?

```
if price < 9.99:
  order = 50
if 9.99 <= price < 19.99:
  order = 30
if price >= 19.99:
  order = 10
```

a.
```
if price < 9.99:
  order = 50
else:
  order = 30
order = 10
```
b.
```
if price < 9.99:
  order = 50
elif price < 19.99:
  order = 30
elif price == 19.99:
  order = 10
```
c.
```
if price < 9.99:
  order = 50
```

```
    elif price < 19.99:
      order = 30
    else:
      order = 10
```

TRY IT

Crochet hook size conversion

Write a program that reads in a crochet hook's US size and computes the metric diameter in millimeters. (A subset of sizes is used.) If the input does not match B-G, the diameter should be assigned with -1.0. Ex: If the input is D, the output is `"3.25 mm"`.

Size conversions for US size: mm

- B : 2.25
- C : 2.75
- D : 3.25
- E : 3.5
- F : 3.75
- G : 4.0

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-5-chained-decisions)

TRY IT

Color wavelengths

Write a program that reads in an integer representing a visible light wavelength in nanometers. Print the corresponding color using the following inclusive ranges:

- Violet: 380–449
- Blue: 450–484
- Cyan: 485–499
- Green: 500–564
- Yellow: 565–589
- Orange: 590–624
- Red: 625–750

Assume the input is within the visible light spectrum, 380-750 inclusive.

Given input:

```
550
```

The output is:

```
Green
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-5-chained-decisions)

4.6 Nested decisions

Learning objectives

By the end of this section you should be able to

- Describe the execution paths of programs with nested `if-else` statements.
- Implement a program with nested `if-else` statements.

Nested decision statements

Suppose a programmer is writing a program that reads in a game ID and player count and prints whether the user has the right number of players for the game.

The programmer may start with:

```
if game == 1 and players < 2:
  print("Not enough players")
if game == 1 and players > 4:
  print("Too many players")
if game == 1 and (2 <= players <= 4):
  print("Ready to start")
if game == 2 and players < 3:
  print("Not enough players")
if game == 2 and players > 6:
  print("Too many players")
if game == 2 and (3 <= players <= 6):
  print("Ready to start")
```

The programmer realizes the code is redundant. What if the programmer could decide the game ID first and then make a decision about players? Nesting allows a decision statement to be inside another decision statement, and is indicated by an indentation level.

An improved program:

```
if game == 1:
  if players < 2:
    print("Not enough players")
  elif players > 4:
    print("Too many players")
  else:
    print("Ready to start")
if game == 2:
```

```
  if players < 3:
    print("Not enough players")
  elif players > 6:
    print("Too many players")
  else:
    print("Ready to start")
# Test game IDs 3-end
```

CHECKPOINT

Example: Poisonous plant identification

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-6-nested-decisions)

CONCEPTS IN PRACTICE

Using nested if-else statements

1. Consider the example above. Given `leaf_count = 9` and `leaf_shape = "teardrop"`, what is the output?
   a. `Might be poison ivy`
   b. `Might be poison oak`
   c. `Might be poison sumac`

2. Given `num_dancers = 49`, what is printed?

```
if num_dancers < 0:
  print("Error: num_dancers is negative")
else:
  if num_dancers % 2 == 1:
    print("Error: num_dancers is odd")
  print(num_dancers, "dancers")
```

   a. `Error: num_dancers is odd`
   b. `49 dancers`
   c. `Error: num_dancers is odd`
      `49 dancers`

3. Given `x = 256`, `y = 513`, and `max = 512`, which of the following will execute?

```
if x == y:
  # Body 1
elif x < y:
  # Body 2
  if y >= max:
    # Body 3
  else:
```

```
    # Body 4
else:
  # Body 5
```

a. Body 2
b. Body 2, Body 3
c. Body 2, Body 5

4. Given x =118, y = 300, and max = 512, which of the following will execute?

```
if x == y:
  # Body 1
elif x < y:
  # Body 2
  if y >= max:
    # Body 3
else:
  # Body 4
else:
  # Body 5
```

a. Body 2
b. Body 3
c. Error

TRY IT

Meal orders

Write a program that reads in a string, "lunch" or "dinner", representing the menu choice, and an integer, 1, 2, or 3, representing the user's meal choice. The program then prints the user's meal choice.

Lunch Meal Options

- 1: Caesar salad
- 2: Spicy chicken wrap
- 3: Butternut squash soup

Dinner Meal Options

- 1: Baked salmon
- 2: Turkey burger
- 3: Mushroom risotto

Ex: If the input is:
lunch
3

The output is:
Your order: Butternut squash soup

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-6-nested-decisions)

4.7 Conditional expressions

Learning objectives

By the end of this section you should be able to

- Identify the components of a conditional expression.
- Create a conditional expression.

Conditional expressions

A **conditional expression** (also known as a "ternary operator") is a simplified, single-line version of an `if-else` statement.

Conditional expression template:

```
expression_if_true if condition else expression_if_false
```

A conditional expression is evaluated by first checking the condition. If `condition` is true, `expression_if_true` is evaluated, and the result is the resulting value of the conditional expression. Else, `expression_if_false` is evaluated, and the result is the resulting value of the conditional expression.

A variable can be assigned with a conditional expression. Ex: Finding the max of two numbers can be calculated with `max_num = y if x < y else x`

Note: Conditional expressions have the lowest precedence of all Python operations.

CHECKPOINT

Example: Version check

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-7-conditional-expressions)

CONCEPTS IN PRACTICE

Using conditional expressions

1. What is the conditional expression version of the following `if-else` statement?

```
if x%2 == 0:
  response = 'even'
else:
  response = 'odd'
```

a. `response = if x%2 == 0 "even" else "odd"`
b. `response = "odd" if x%2 == 0 else "even"`

c. `response = "even" if x%2 == 0 else "odd"`

2. Given `x = 100` and `offset = 10`, what is the value of `result`?

```
result = x + offset if x < 100 else x - offset
```

a. `90`
b. `100`
c. `110`

3. Which part of the conditional expression is incorrect?

```
min_num = x if x < y else min_num = y
```

a. `min_num = x`
b. `x < y`
c. `min_num = y`

4. Which of the following is an improved version of the following `if-else` statement?

```
if x < 50:
  result = True
else:
  result = False
```

a. `result = True if x < 50 else False`
b. `result = x < 50`

5. What are the possible values of `total`?

```
total = fee + 10 if hours > 12 else 2
```

a. `10, 2`
b. `fee + 10, 2`
c. `fee + 10, fee + 2`

TRY IT

Ping values

Write a program that reads in an integer, `ping`, and prints `ping_report`, a string indicating whether the ping is low to average or too high. `ping` values under 150 have a `ping_report` of `"low to average"`. `ping` values of 150 and higher have a `ping_report` of `"too high"`. Use a conditional expression to assign `ping_report`.

Ex: If the input is `30`, the output is `"Ping is low to average"`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/4-7-conditional-expressions)

4.8 Chapter summary

Highlights from this chapter include:

- Booleans represent a value of `True` or `False`.
- Comparison operators compare values and produce `True` or `False`.
- Logical operators take condition operand(s) and produce `True` or `False`.
- Operators are evaluated in order according to precedence and associativity.
- Conditions are expressions that evaluate to `True` or `False`.
- Decision statements allow different paths of execution (branches) through code based on conditions.
- Decision statements can be nested inside other decision statements.
- Conditional expressions are single-line versions of `if-else` statements.

At this point, you should be able to write programs that evaluate conditions and execute code statements accordingly with the correct order of operations. The programming practice below ties together most topics presented in the chapter.

Function	Description
`bool(x)`	Converts x to a Boolean value, either `True` or `False`.
Operator	**Description**
`x == y` (Equality)	Compares the values of x and y and returns `True` if the values are equal and `False` otherwise. Ex: `10 == 10` is `True`.
`x != y` (Inequality)	Compares the values of x and y and returns `True` if the values are inequal and `False` otherwise. Ex: `7 != 4` is `True`.
`x > y` (Greater than)	Compares the values of x and y and returns `True` if the x is greater than y and `False` otherwise. Ex: `9 > 3` is `True`.
`x < y` (Less than)	Compares the values of x and y and returns `True` if the x is less than y and `False` otherwise. Ex: `9 < 8` is `False`.
`x >= y` (Greater than or equal)	Compares the values of x and y and returns `True` if the x is greater than or equal to y and `False` otherwise. Ex: `2 >= 2` is `True`.
`x <= y` (Less than or equal)	Compares the values of x and y and returns `True` if the x is less than or equal to y and `False` otherwise. Ex: `8 <= 7` is `False`.

Table 4.5 Chapter 4 reference.

Function	Description
`x and y` (Logical)	Evaluates the Boolean values of x and y and returns `True` if both are true. Ex: `True and False` is `False`.
`x or y` (Logical)	Evaluates the Boolean values of x and y and returns `True` if either is true. Ex: `True or False` is `True`.
`not x` (Logical)	Evaluates the Boolean value of x and returns `True` if the value is false and `False` if the value is true. Ex: `not True` is `False`.
Decision statement	**Description**

`if` statement	`# Statements before`
	
	`if condition:`
	`  # Body`
	
	`# Statements after`

`else` statement	`# Statements before`
	
	`if condition:`
	`  # Body`
	`else:`
	`  # Body`
	
	`# Statements after`

Table 4.5 Chapter 4 reference.

Function	Description

`elif` statement	`# Statements before`
	
	`if condition:`
	`  # Body`
	`elif condition:`
	`  # Body`
	`else:`
	`  # Body`
	
	`# Statements after`

Nested `if` statement	`# Statements before`
	
	`if condition:`
	`  if condition:`
	`  # Body`
	`  else:`
	`  # Body`
	`else:`
	`  if condition:`
	`  # Body`
	`  else:`
	`  # Body`
	
	`# Statements after`

Conditional expression	`expression_if_true if condition else expression_if_false`

Table 4.5 Chapter 4 reference.

5 Loops

Figure 5.1 credit: modification of work "Quantum Computing", by Kevin Dooley/Flickr, CC BY 2.0

Chapter Outline

Introduction

A **loop** is a code block that runs a set of statements while a given condition is true. A loop is often used for performing a repeating task. Ex: The software on a phone repeatedly checks to see if the phone is idle. Once the time set by a user is reached, the phone is locked. Loops can also be used for iterating over lists like student names in a roster, and printing the names one at a time.

In this chapter, two types of loops, `for` loop and `while` loop, are introduced. This chapter also introduces `break` and `continue` statements for controlling a loop's execution.

5.1 While loop

Learning objectives

By the end of this section you should be able to

- Explain the loop construct in Python.
- Use a `while` loop to implement repeating tasks.

While loop

A **while loop** is a code construct that runs a set of statements, known as the loop body, while a given condition, known as the loop expression, is true. At each iteration, once the loop statement is executed, the

loop expression is evaluated again.

- If true, the loop body will execute at least one more time (also called looping or iterating one more time).
- If false, the loop's execution will terminate and the next statement after the loop body will execute.

CHECKPOINT

While loop

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-1-while-loop)

CONCEPTS IN PRACTICE

While loop example

Fibonacci is a series of numbers in which each number is the sum of the two preceding numbers. The Fibonacci sequence starts with two ones: 1, 1, 2, 3, ... Consider the following code that prints all Fibonacci numbers less than 20, and answer the following questions.

```
    # Initializing the first two Fibonacci numbers
    f = 1
    g = 1
    print (f, end = ' ')

    # Running the loop while the last Fibonacci number is less than 20
    while g < 20:
      print(g, end = ' ')
      # Calculating the next Fibonacci number and updating the last two sequence
numbers
      temp = f
      f = g
      g = temp + g
```

1. How many times does the loop execute?
   a. 5
   b. 6
   c. 7

2. What is the variable g's value when the while loop condition evaluates to `False`?
   a. `13`
   b. `20`
   c. `21`

3. What are the printed values in the output?
   a. `1 1 2 3 5 8 13`
   b. `1 2 3 5 8 13`
   c. `1 1 2 3 5 8 13 21`

Counting with a while loop

A `while` loop can be used to count up or down. A counter variable can be used in the loop expression to determine the number of iterations executed. Ex: A programmer may want to print all even numbers between 1 and 20. The task can be done by using a counter initialized with 1. In each iteration, the counter's value is increased by one, and a condition can check whether the counter's value is an even number or not. The change in the counter's value in each iteration is called the **step size**. The step size can be any positive or negative value. If the step size is a positive number, the counter counts in ascending order, and if the step size is a negative number, the counter counts in descending order.

EXAMPLE 5.1

A program printing all odd numbers between 1 and 10

```
# Initialization
counter = 1

# While loop condition
while counter <= 10:
  if counter % 2 == 1:
    print(counter)
  # Counting up and increasing counter's value by 1 in each iteration
  counter += 1
```

CHECKPOINT

Counting with while loop

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-1-while-loop)

CONCEPTS IN PRACTICE

while loop counting examples

Given the code, answer the following questions.

```
n = 4
while n > 0:
  print(n)
  n = n - 1

print("value of n after the loop is", n)
```

4. How many times does the loop execute?
   a. 3
   b. 4
   c. 5

5. Which line is printed as the last line of output?
   a. `value of n after the loop is -1.`
   b. `value of n after the loop is 0.`
   c. `value of n after the loop is 1.`

6. What happens if the code is changed as follows?

```
n = 4
while n > 0:
  print(n)
  # Modified line
  n = n + 1

print("value of n after the loop is", n)
```

   a. The code will not run.
   b. The code will run for one additional iteration.
   c. The code will never terminate.

TRY IT

Reading inputs in a while loop

Write a program that takes user inputs in a `while` loop until the user enters `"begin"`. Test the code with the given input values to check that the loop does not terminate until the input is `"begin"`. Once the input `"begin"` is received, print `"The while loop condition has been met."`.

Enter different input words to see when the `while` loop condition is met.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-1-while-loop)

TRY IT

Sum of odd numbers

Write a program that reads two integer values, `n1` and `n2`. Use a `while` loop to calculate the sum of odd numbers between `n1` and `n2` (inclusive of `n1` and `n2`). Remember, a number is odd if `number % 2 != 0`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-1-while-loop)

5.2 For loop

Learning objectives

By the end of this section you should be able to

- Explain the `for` loop construct.
- Use a `for` loop to implement repeating tasks.

For loop

In Python, a **container** can be a range of numbers, a string of characters, or a list of values. To access objects within a container, an iterative loop can be designed to retrieve objects one at a time. A **for loop** iterates over all elements in a container. Ex: Iterating over a class roster and printing students' names.

CHECKPOINT

For loop example for iterating over a container object

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-2-for-loop)

CONCEPTS IN PRACTICE

For loop over a string container

A string variable can be considered a container of multiple characters, and hence can be iterated on. Given the following code, answer the questions.

```
str_var = "A string"

count = 0
for c in str_var:
  count += 1

print(count)
```

1. What is the program's output?
   a. `7`
   b. `8`
   c. `9`

2. What's the code's output if the line `count += 1` is replaced with `count *= 2`?
   a. `0`
   b. `16`
   c. 2^8

3. What is printed if the code is changed as follows?

```
str_var = "A string"

count = 0
for c in str_var:
  count += 1
  # New line
  print(c, end = '*')

print(count)
```

a. A string*
b. A*s*t*r*i*n*g*
c. A* *s*t*r*i*n*g*

Range( ) function in for loop

A `for` loop can be used for iteration and counting. The `range( )` function is a common approach for implementing counting in a `for` loop. A **range()** function generates a sequence of integers between the two numbers given a step size. This integer sequence is inclusive of the start and exclusive of the end of the sequence. The `range( )` function can take up to three input values. Examples are provided in the table below.

Range function	Description	Example	Output
`range(end)`	• Generates a sequence beginning at `0` until end. • Step size: `1`	`range(4)`	`0, 1, 2, 3`
`range(start, end)`	• Generates a sequence beginning at `start` until end. • Step size: `1`	`range(0, 3)`	`0, 1, 2`
		`range(2, 6)`	`2, 3, 4, 5`
		`range(-13, -9)`	`-13, -12, -11, -10`
`range(start, end, step)`	• Generates a sequence beginning at `start` until end. • Step size: `step`	`range(0, 4, 1)`	`0, 1, 2, 3`
		`range(1, 7, 2)`	`1, 3, 5`

Table 5.1 Using the range() function.

Range function	Description	Example	Output
		`range(3, -2, -1)`	`3, 2, 1, 0, -1`
		`range(10, 0, -4)`	`10, 6, 2`

Table 5.1 Using the range() function.

EXAMPLE 5.2

Two programs printing all integer multiples of 5 less than 50 (Notice the compactness of the `for` construction compared to the `while`)

```
# For loop condition using
# range() function to print
# all multiples of 5 less than 50
for i in range(0, 50, 5):
  print(i)
```

```
# While loop implementation of printing
# multiples of 5 less than 50

# Initialization
i = 0
# Limiting the range to be less than 50
while i < 50:
    print(i)
    i+=5
```

Table 5.2

CONCEPTS IN PRACTICE

For loop using a range() function

4. What are the arguments to the `range()` function for the increasing sequence of every 3rd integer from 10 to 22 (inclusive of both ends)?

a. `range(10, 23, 3)`
b. `range(10, 22, 3)`
c. `range(22, 10, -3)`

5. What are the arguments to the `range()` function for the decreasing sequence of every integer from 5 to 1 (inclusive of both ends)?

a. `range(5, 1, 1)`
b. `range(5, 1, -1)`
c. `range(5, 0, -1)`

6. What is the sequence generated from `range(-1, -2, -1)`?
   a. `1`
   b. `-1, -2`
   c. `-2`

7. What is the output of the `range(1, 2, -1)`?
   a. `1`
   b. `1, 2`
   c. empty sequence

8. What is the output of `range(5, 2)`?
   a. `0, 2, 4`
   b. `2, 3, 4`
   c. empty sequence

TRY IT

Counting spaces

Write a program using a `for` loop that takes in a string as input and counts the number of spaces in the provided string. The program must print the number of spaces counted. Ex: If the input is `"Hi everyone"`, the program outputs `1`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-2-for-loop)

TRY IT

Sequences

Write a program that reads two integer values, n1 and n2, with `n1 < n2`, and performs the following tasks:

1. Prints all even numbers between the two provided numbers (inclusive of both), in ascending order.
2. Prints all odd numbers between the two provided numbers (exclusive of both), in descending order.

```
Input: 2 8



prints
2 4 6 8
7 5 3
```

Note: the program should return an error message if the second number is smaller than the first.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-2-for-loop)

5.3 Nested loops

Learning objectives

By the end of this section you should be able to

- Implement nested `while` loops.
- Implement nested `for` loops.

Nested loops

A **nested loop** has one or more loops within the body of another loop. The two loops are referred to as **outer loop** and **inner loop**. The outer loop controls the number of the inner loop's full execution. More than one inner loop can exist in a nested loop.

CHECKPOINT

Nested while loops

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-3-nested-loops)

EXAMPLE 5.3

Available appointments

Consider a doctor's office schedule. Each appointment is 30 minutes long. A program to print available appointments can use a nested `for` loop where the outer loop iterates over the hours, and the inner loop iterates over the minutes. This example prints time in hours and minutes in the range between 8:00am and 10:00am. In this example, the outer loop iterates over the time's hour portion between 8 and 9, and the inner loop iterates over the time's minute portion between 0 and 59.

```
hour = 8
minute = 0
while hour <= 9:
  while minute <= 59:
    print(hour, ":", minute)
    minute += 30
  hour += 1
  minute = 0
```

The above code's output is:

```
8 : 0
8 : 30
9 : 0
9 : 30
```

CONCEPTS IN PRACTICE

Nested while loop question set

1. Given the following code, how many times does the print statement execute?

```
i = 1
while i <= 5:
  j = 1
  while i + j <= 5:
    print(i, j)
    j += 1
  i += 1
```

a. 5
b. 10
c. 25

2. What is the output of the following code?

```
i = 1

while i <= 2:
  j = 1
  while j <= 3:
    print('*', end = '')
    j += 1
  print()
  i += 1
```

a. ******
b. ***
   ***
c. **
   **
   **

3. Which program prints the following output?

```
1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16
```

a.
```
i = 1
while i <= 4:
  while j <= 4:
    print(i * j, end = ' ')
    j += 1
  print()
```

```
      i += 1
b.  i = 1
    while i <= 4:
      j = 1
      while j <= 4:
        print(i * j, end = ' ')
        j += 1
      print()
      i += 1
c.  i = 1
    while i <= 4:
      j = 1
      while j <= 4:
        print(i * j, end = ' ')
        j += 1
      i += 1
```

Nested for loops

A nested `for` loop can be implemented and used in the same way as a nested `while` loop. A `for` loop is a preferable option in cases where a loop is used for counting purposes using a `range()` function, or when iterating over a container object, including nested situations. Ex: Iterating over multiple course rosters. The outer loop iterates over different courses, and the inner loop iterates over the names in each course roster.

CHECKPOINT

Nested for loops

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-3-nested-loops)

CONCEPTS IN PRACTICE

Nested loop practices

4. Given the following code, how many times does the outer loop execute?

```
for i in range(3):
  for j in range(4):
    print(i, ' ', j)
```

a. 3
b. 4
c. 12

5. Given the following code, how many times does the inner loop execute?

```
for i in range(3):
```

```
  for j in range(4):
    print(i, ' ', j)
```

a. 4
b. 12
c. 20

6. Which program prints the following output?

```
0 1 2 3
0 2 4 6
0 3 6 9
```

a.
```
for i in range(4):
  for j in range(4):
    print(i * j, end = ' ')
  print()
```
b.
```
for i in range(1, 4):
  for j in range(4):
    print(i * j)
```
c.
```
for i in range(1, 4):
  for j in range(4):
    print(i * j, end = ' ')
  print()
```

MIXED LOOPS

The two `for` and `while` loop constructs can also be mixed in a nested loop construct. Ex: Printing even numbers less than a given number in a list. The outer loop can be implemented using a `for` loop iterating over the provided list, and the inner loop iterates over all even numbers less than a given number from the list using a `while` loop.

```
numbers = [12, 5, 3]

i = 0
for n in numbers:
  while i < n:
    print (i, end = " ")
    i += 2
  i = 0
  print()
```

TRY IT

Printing a triangle of numbers

Write a program that prints the following output:

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

`Finish!`

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-3-nested-loops)

TRY IT

Printing two triangles

Write a program that prints the following output using nested `while` and `for` loops:

```
****
***
**
*
++++
+++
++
+
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-3-nested-loops)

5.4 Break and continue

Learning objectives

By the end of this section you should be able to

- Analyze a loop's execution with `break` and `continue` statements.
- Use `break` and `continue` control statements in `while` and `for` loops.

Break

A **break** statement is used within a `for` or a `while` loop to allow the program execution to exit the loop once a given condition is triggered. A `break` statement can be used to improve runtime efficiency when further loop execution is not required.

Ex: A loop that looks for the character `"a"` in a given string called `user_string`. The loop below is a regular `for` loop for going through all the characters of `user_string`. If the character is found, the break statement takes execution out of the `for` loop. Since the task has been accomplished, the rest of the `for` loop execution is bypassed.

```
user_string = "This is a string."
for i in range(len(user_string)):
  if user_string[i] == 'a':
    print("Found at index:", i)
    break
```

CHECKPOINT

Break statement in a while loop

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-4-break-and-continue)

INFINITE LOOP

A `break` statement is an essential part of a loop that does not have a termination condition. A loop without a termination condition is known as an **infinite loop**. Ex: An infinite loop that counts up starting from 1 and prints the counter's value while the counter's value is less than 10. A `break` condition is triggered when the counter's value is equal to 10, and hence the program execution exits.

```
counter = 1
while True:
  if counter >= 10:
    break
  print(counter)
  counter += 1
```

CONCEPTS IN PRACTICE

Using a break statement

1. What is the following code's output?

```
string_val = "Hello World"
for c in string_val:
  if c == " ":
    break
  print(c)
```

a. Hello
b. Hello World
c. H
   e
   l
   l
   o

2. Given the following code, how many times does the print statement get executed?

```
i = 1
while True:
  if i%3 == 0 and i%5 == 0:
    print(i)
    break
  i += 1
```

a. 0
b. 1
c. 15

3. What is the final value of i?

```
i = 1
count = 0
while True:
  if i%2 == 0 or i%3 == 0:
    count += 1
  if count >= 5:
    print(i)
    break
  i += 1
```

a. 5
b. 8
c. 30

Continue

A **continue** statement allows for skipping the execution of the remainder of the loop without exiting the loop entirely. A `continue` statement can be used in a `for` or a `while` loop. After the `continue` statement's execution, the loop expression will be evaluated again and the loop will continue from the loop's expression. A `continue` statement facilitates the loop's control and readability.

CHECKPOINT

Continue statement in a while loop

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-4-break-and-continue)

CONCEPTS IN PRACTICE

Using a continue statement

4. Given the following code, how many times does the print statement get executed?

```
i = 10
while i >= 0:
  i -= 1
  if i%3 != 0:
    continue
  print(i)
```

a. 3
b. 4
c. 11

5. What is the following code's output?

```
for c in "hi Ali":
  if c == " ":
    continue
  print(c)
```

a.
```
h
i

A
l
i
```
b.
```
h
i
A
l
i
```
c.
```
hi
Ali
```

TRY IT

Using break control statement in a loop

Write a program that reads a string value and prints `"Found"` if the string contains a space character. Else, prints `"Not found"`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-4-break-and-continue)

TRY IT

Using a continue statement in a loop

Complete the following code so that the program calculates the sum of all the numbers in list `my_list` that are greater than or equal to 5.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-4-break-and-continue)

5.5 Loop else

Learning objectives

By the end of this section you should be able to

- Use loop `else` statement to identify when the loop execution is interrupted using a `break` statement.
- Implement a loop `else` statement with a `for` or a `while` loop.

Loop else

A **loop else** statement runs after the loop's execution is completed without being interrupted by a break statement. A loop `else` is used to identify if the loop is terminated normally or the execution is interrupted by a `break` statement.

Ex: A `for` loop that iterates over a list of numbers to find if the value `10` is in the list. In each iteration, if `10` is observed, the statement `"Found 10!"` is printed and the execution can terminate using a break statement. If `10` is not in the list, the loop terminates when all integers in the list are evaluated, and hence the `else` statement will run and print `"10 is not in the list."` Alternatively, a Boolean variable can be used to track whether number `10` is found after loop's execution terminates.

EXAMPLE 5.4

Finding the number 10 in a list

In the code examples below, the code on the left prints `"Found 10!"` if the variable `i`'s value is `10`. If the value `10` is not in the list, the code prints `"10 is not in the list."`. The code on the right uses the seen Boolean variable to track if the value `10` is in the list. After loop's execution, if `seen`'s value is still false,

the code prints "10 is not in the list.".

`numbers = [2, 5, 7, 11, 12]`
`for i in numbers:`
`  if i == 10:`
`  print("Found 10!")`
`  break`
`else:`
`  print("10 is not in the list.")`
	`numbers = [2, 5, 7, 11, 12]`
	`seen = False`
	`for i in numbers:`
	`  if i == 10:`
	`  print("Found 10!")`
	`  seen = True`
	`if seen == False:`
	`  print("10 is not in the list.")`

Table 5.3

CHECKPOINT

Loop else template

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-5-loop-else)

CONCEPTS IN PRACTICE

Loop else practices

1. What is the output?

```
n = 16
exp = 0
i = n
while i > 1:
  if n%2 == 0:
    i = i/2
    exp += 1
  else:
    break
else:
  print(n,"is 2 to the", exp)
```

   a. `16 is 2 to the 3`
   b. `16 is 2 to the 4`
   c. no output

2. What is the output?

```
n = 7
exp = 0
i = n
while i > 1:
  if n%2 == 0:
    i = i//2
    exp += 1
  else:
    break
else:
  print(n,"is 2 to the", exp)
```

a. no output
b. `7 is 2 to the 3`
c. `7 is 2 to the 2`

3. What is the output?

```
numbers = [1, 2, 2, 6]
for i in numbers:
  if i >= 5:
    print("Not all numbers are less than 5.")
    break
  else:
    print(i)
    continue
else:
  print("all numbers are less than 5.")
```

a.
```
1
2
2
6
Not all numbers are less than 5.
```
b.
```
1
2
2
Not all numbers are less than 5.
```
c.
```
1
2
2
6
all numbers are less than 5.
```

TRY IT

Sum of values less than 10

Write a program that, given a list, calculates the sum of all integer values less than 10. If a value greater than or equal to 10 is in the list, no output should be printed. Test the code for different values in the list.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-5-loop-else)

5.6 Chapter summary

Highlights from this chapter include:

- A `while` loop runs a set of statements, known as the loop body, while a given condition, known as the loop expression, is true.
- A `for` loop can be used to iterate over elements of a container object.
- A `range()` function generates a sequence of integers between the two numbers given a step size.
- A nested loop has one or more loops within the body of another loop.
- A `break` statement is used within a `for` or a `while` loop to allow the program execution to exit the loop once a given condition is triggered.
- A `continue` statement allows for skipping the execution of the remainder of the loop without exiting the loop entirely.
- A loop `else` statement runs after the loop's execution is completed without being interrupted by a `break` statement.

At this point, you should be able to write programs with loop constructs. The programming practice below ties together most topics presented in the chapter.

Function	Description
`range(end)`	Generates a sequence beginning at 0 until end with step size of 1.
`range(start, end)`	Generates a sequence beginning at `start` until end with step size of 1.
`range(start, end, s)`	Generates a sequence beginning at `start` until end with the step size of `s`.
Loop constructs	**Description**

Table 5.4 Chapter 5 reference.

Function	Description

`while` loop	`# initialization`
	`while expression:`
	`  # loop body`
	
	`# statements after the loop`

`for` loop	`# initialization`
	`for loop_variable in container:`
	`  # loop body`
	
	`# statements after the loop`

Nested `while` loop	`while outer_loop_expression:`
	`  # outer loop body (1)`
	`  while inner_loop_expression:`
	`  # inner loop body`
	`  # outer loop body (2)`
	
	`# statements after the loop`

`break` statement	`# initialization`
	`while loop_expression:`
	`  # loop body`
	`  if break_condition:`
	`  break`
	`  # remaining body of loop`
	
	`# statements after the loop`

Table 5.4 Chapter 5 reference.

Function	Description

`continue` statement	`# initialization`
	`while loop_expression:`
	`  # loop body`
	`  if continue_condition:`
	`  continue`
	`  # remaining body of loop`
	
	`# statements after the loop`

Loop `else` statement	`# initialization`
	`for loop_expression:`
	`  # loop body`
	`  if break_condition:`
	`  break`
	`  # remaining body of loop`
	`else:`
	`  # loop else statement`
	
	`# statements after the loop`

Table 5.4 Chapter 5 reference.

TRY IT

Prime numbers

Write a program that takes in a positive integer number (`N`) and prints out the first `N` prime numbers on separate lines.

Note: A prime number is a number that is not divisible by any positive number larger than 1. To check whether a `number` is prime, the condition of `number % i != 0` can be checked for `i` greater than `1` and less than `number`.

Ex: if `N = 6`, the output is:

```
2
3
5
7
```

```
11
13
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/5-6-chapter-summary)

6 Functions

Figure 6.1 credit: modification of work "IMG_3037", by Jay Roc/Flickr, Public Domain

Chapter Outline

Introduction

Functions are the next step toward creating optimized code as a software developer. If the same block of code is reused repeatedly, a function allows the programmer to write the block of code once, name the block, and use the code as many times as needed by calling the block by name. Functions can read in values and return values to perform tasks, including complex calculations.

Like branching statements discussed in the Decisions chapter, functions allow different paths of execution through a program, and this chapter discusses control flow and the scope of variables in more detail.

6.1 Defining functions

Learning objectives

By the end of this section you should be able to

- Identify function calls in a program.
- Define a parameterless function that outputs strings.
- Describe benefits of using functions.

Calling a function

Throughout the book, functions have been called to perform tasks. Ex: `print()` prints values, and `sqrt()` calculates the square root. A **function** is a named, reusable block of code that performs a task when called.

CHECKPOINT

Example: Simple math program

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-1-defining-functions)

CONCEPTS IN PRACTICE

Identifying function calls

1. Which line has a function call?

```
1 input_num = 14
2 offset_num = input_num - 10
3 print(offset_num)
```

   a. line 1
   b. line 2
   c. line 3

2. How many times is print() called?

```
print("Please log in")
username = input("Username:")
password = input("Password:")
print("Login successful")
print("Welcome,", username)
```

   a. 1
   b. 3
   c. 5

3. How many function calls are there?

```
# Use float() to convert input for area calculation
width = float(input("Enter width:"))
height = float(input("Enter height:"))
print("Area is", width*height)
```

   a. 3
   b. 5
   c. 6

Defining a function

A function is defined using the **def** keyword. The first line contains `def` followed by the function name (in snake case), parentheses (with any parameters—discussed later), and a colon. The indented body begins with a documentation string describing the function's task and contains the function statements. A function must be defined before the function is called.

CHECKPOINT

Example: Welcome message function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-1-defining-functions)

CONCEPTS IN PRACTICE

Defining functions

4. What's wrong with the first line of the function definition?

```
def water_plant:
```

a. A docstring should go after the colon.
b. Parentheses should go after `water_plant`.
c. `def` should be `define` before `water_plant`.

5. What is the output?

```
def print_phone_num():
  print("Phone: (", 864, ")", 555, "-", 0199)

print("User info:")
print_phone_num()
```

a. `Phone: ( 864 ) 555 - 1000`
   `User info:`
   `Phone: ( 864 ) 555 - 1000`
b. `User info:`
c. `User info:`
   `Phone: ( 864 ) 555 - 1000`

6. Which statement calls a function named `print_pc_specs()`?

a. `print_pc_specs`
b. `print_pc_specs()`
c. `print_pc_specs():`

7. Which is an appropriate name for a function that calculates a user's taxes?

a. `calc_tax`
b. `calculate user tax`
c. `c_t`

Benefits of functions

A function promotes modularity by putting code statements related to a single task in a separate group. The body of a function can be executed repeatedly with multiple function calls, so a function promotes reusability. Modular, reusable code is easier to modify and is shareable among programmers to avoid reinventing the wheel.

CHECKPOINT

Improving a program with a function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-1-defining-functions)

CONCEPTS IN PRACTICE

Improving programs with functions

Consider the code above.

8. If the points were changed from floats to integers, how many statements would need to be changed in the original and revised programs respectively?
   a. 3, 1
   b. 4, 4
   c. 12, 4

9. How many times can `calc_distance()` be called?
   a. 1
   b. 3
   c. many

TRY IT

Cinema concession stand

Write a function, `concessions()`, that prints the food and drink options at a cinema.

Given:

```
concessions()
```

The output is:

```
Food/Drink Options:
Popcorn: $8-10
Candy: $3-5
Soft drink: $5-7
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-1-defining-functions)

TRY IT

Terms and conditions prompt

Write a function, `terms()`, that asks the user to accept the terms and conditions, reads in `Y/N`, and outputs a response. In the main program, read in the number of users and call `terms()` for each user.

Given inputs `1` and `"Y"`, the output is:

```
Do you accept the terms and conditions?
Y
Thank you for accepting.
```

Given inputs `2`, `"N"`, and `"Y"`, the output is:

```
Do you accept the terms and conditions?
N
Have a good day.
Do you accept the terms and conditions?
Y
Thank you for accepting.
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-1-defining-functions)

6.2 Control flow

Learning objectives

By the end of this section you should be able to

- Identify the control flow of a program.
- Describe how control flow moves between statements and function calls.

Control flow and functions

Control flow is the sequence of program execution. A program's control flow begins at the main program but rarely follows a strict sequence. Ex: Control flow skips over lines when a conditional statement isn't executed.

When execution reaches a function call, control flow moves to where the function is defined and executes the function statements. Then, control flow moves back to where the function was called and continues the sequence.

CHECKPOINT

Calling a brunch menu function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-2-control-flow)

CONCEPTS IN PRACTICE

Following the control flow

1. Which line is executed first?

```
1 def park_greet():
2   """Output greeting."""
3   print("Welcome. Open sunrise to sunset.")
4
5 car_count = 1
6 park_greet()
7 if car_count > 50:
8   # Direct to extra parking lot
```

   a. 1
   b. 3
   c. 5

2. Control flow moves to line 9, and `park_greet()` is called. Which line does control flow move to next?

```
 1 def extra_lot():
 2   # Function definition
 3
 4 def park_greet():
 5   """Output greeting."""
 6   print("Welcome. Open sunrise to sunset.")
 7
 8 car_count = 1
 9 park_greet()
10 if car_count > 50:
11   extra_lot()
```

   a. 1
   b. 4
   c. 10

3. Control flow moves to line 12, and `extra_lot()` is called. Which line does control flow move to after line 3 is executed?

```
1 def extra_lot():
2   """Output extra parking lot info."""
```

```
  print("Take the second right to park.")

def park_greet():
  """Output greeting."""
  print("Welcome. Open sunrise to sunset.")

car_count = 1
park_greet()
if car_count > 50:
  extra_lot()
```

a. 5
b. 8
c. 12

4. What is the output?

```
def park_greet():
  """Output greeting."""
  print("Welcome to the park")
print("Open sunrise to sunset")

park_greet()
```

a. `Welcome to the park`
b. `Welcome to the park`
   `Open sunrise to sunset`
c. `Open sunrise to sunset`
   `Welcome to the park`

Functions calling functions

Functions frequently call other functions to keep the modularity of each function performing one task. Ex: A function that calculates an order total may call a function that calculates sales tax. When a function called from another function finishes execution, control flow returns to the calling function.

CHECKPOINT

Example: Book club email messages

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-2-control-flow)

CONCEPTS IN PRACTICE

Functions calling functions

Consider the book club example above.

5. How many function calls occur during the execution of the program?
   a. 2
   b. 3
   c. 6

6. When line 3 is reached and executed, which line does control flow return to?
   a. 1
   b. 11
   c. 16

TRY IT

Updated terms and conditions prompt

Write an updated function, `terms()`, that asks the user to accept the terms and conditions, reads in `Y/N`, and outputs a response by calling `accepted()` or `rejected()`. `accepted()` prints `"Thank you for accepting the terms."` and `rejected()` prints `"You have rejected the terms. Thank you."`.

Given inputs `2`, `"Y"` and `"N"`, the output is:

```
Do you accept the terms and conditions?
Y
Thank you for accepting the terms.
```

Given a function call to `terms()` and input `"N"`, the output is:

```
Do you accept the terms and conditions?
N
You have rejected the terms. Thank you.
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-2-control-flow)

TRY IT

Laundromat information

Write a program that uses three functions to print information about a laundromat, Liam's Laundry:

- `laundromat_info()`: Prints the name, Liam's Laundry, and hours of operation, 7a - 11p, and calls `washers_open()` and `dryers_open()`
- `washers_open()`: Reads an integer, assigns `washer_count` with the value, and prints `washer_count`
- `dryers_open()`: Reads an integer, assigns `dryer_count` with the value, and prints `dryer_count`

The main program should just call `laundromat_info()`.

Given inputs `50` and `40`, the output is:

```
Liam's Laundry
7a - 11p
 Open washers: 50
 Open dryers: 40
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-2-control-flow)

6.3 Variable scope

Learning objectives

By the end of this section you should be able to

- Identify the scope of a program's variables.
- Discuss the impact of a variable's scope.

Global scope

A variable's **scope** is the part of a program where the variable can be accessed. A variable created outside of a function has **global scope** and can be accessed anywhere in the program. A Python program begins in global scope, and the global scope lasts for the entire program execution.

CHECKPOINT

Global variables in a program with a function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-3-variable-scope)

CONCEPTS IN PRACTICE

Global variables

1. Which variables are global?

```
num = float(input())
num_sq = num * num
print(num, "squared is", num_sq)
```

   a. `num` only
   b. `num_sq` only
   c. `num` and `num_sq`

2. Which variables have global scope?

```
def print_square():
```

```
    num_sq = num * num
    print(num, "squared is", num_sq)

num = float(input())
print_square()
```

a. num only
b. num_sq only
c. num and num_sq

3. Which functions can access num?

```
def print_double():
    num_d = num * 2
    print(num, "doubled is", num_d)

def print_square():
    num_sq = num * num
    print(num, "squared is", num_sq)

num = float(input())
print_double()
print_square()
```

a. print_double()
b. print_square()
c. print_double() and print_square()

Local scope

A variable created within a function has **local scope** and only exists within the function. A local variable cannot be accessed outside of the function in which the variable was created. After a function finishes executing, the function's local variables no longer exist.

CHECKPOINT

Global and local variables in a program with a function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-3-variable-scope)

CONCEPTS IN PRACTICE

Local variables

4. Which variables are local?

```
def print_time():
  out_str = "Time is " + str(hour) + ":" + str(min)
  print(out_str)

hour = int(input())
min = int(input())
print_time()
```

a. hour and min
b. out_str
c. hour, min, and out_str

5. Which variables are local?

```
def print_greeting():
  print(out_str)

hour = int(input())
min = int(input())
if hour < 12:
  out_str = "Good morning"
else:
  out_str = "Good day"
print_greeting()
```

a. hour and min
b. out_str
c. none

6. Which functions directly access out_str?

```
def print_greeting():
  print("Good day,")
  print_time()

def print_time():
  out_str = "Time is " + str(hour) + ":" + str(min)
  print(out_str)

hour = int(input())
min = int(input())
print_greeting()
```

a. print_greeting()
b. print_time()
c. print_greeting() and print_time()

Using local and global variables together

Python allows global and local variables to have the same name, which can lead to unexpected program behavior. A function treats a variable edited within the function as a local variable unless told otherwise. To edit a global variable inside a function, the variable must be declared with the `global` keyword.

CHECKPOINT

Editing global variables in a program with a function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-3-variable-scope)

CONCEPTS IN PRACTICE

Using both local and global variables

Consider the following variations on the example program with the input 9.

7. What is the output?

```
def update_hour():
  tmp = hour
  if is_dst:
    tmp += 1
  else:
    tmp -= 1

is_dst = True
hour = int(input("Enter hour: "))
update_hour()
print("New hour:", hour)
```

a. New hour: 9
b. New hour: 10
c. Error

8. What is the output?

```
def update_hour():
  new_hour = hour
  if is_dst:
    new_hour += 1
  else:
    new_hour -= 1

is_dst = True
hour = int(input("Enter hour: "))
update_hour()
print("New hour:", new_hour)
```

a. `New hour: 9`
b. `New hour: 10`
c. `Error`

9. What is the output?

```
Enter hour: "))
update_hour()
print("New hour:", new_hour)
def update_hour():
  global new_hour
  new_hour = hour
  if is_dst:
    new_hour += 1
  else:
    new_hour -= 1

is_dst = True
hour = int(input("Enter hour: "))
update_hour()
print("New hour:", new_hour)
```

a. `New hour: 9`
b. `New hour: 10`
c. `Error`

BENEFITS OF LIMITING SCOPE

A programmer might ask, "Why not just make all variables global variables to avoid access errors?" Making every variable global can make a program messy. Ex: A programmer debugging a large program discovers a variable has the wrong value. If the whole program can modify the variable, then the bug could be anywhere in the large program. Limiting a variable's scope to only what's necessary and restricting global variable use make a program easier to debug, maintain, and update.

TRY IT

Battle royale game launch

Write a program that reads in a selected game mode and calls one of two functions to launch the game. If the input is `"br"`, call `battle_royale()`. Otherwise, call `practice()`.

`battle_royale()`:

- Reads in the number of players.
- Computes the number of teammates still needed. A full team is 3 players.
- Calls the function `find_teammates()` with the calculated number.
- Prints `"Match starting . . ."`.

`practice()`:

- Reads in a string representing the desired map.
- Prints `"Launching practice on [desired map]"`.

Note: `find_teammates()` is provided and does not need to be edited.

Given input:

```
br
1
```

The output is:

```
Finding 2 players...
Match starting...
```

Given input:

```
p
Queen's Canyon
```

The output is:

```
Launching practice on Queen's Canyon
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-3-variable-scope)

6.4 Parameters

Learning objectives

By the end of this section you should be able to

- Identify a function's arguments and parameters.
- Describe how mutability affects how a function can modify arguments.

Arguments and parameters

What if a programmer wants to write a function that prints the contents of a list? Good practice is to pass values directly to a function rather than relying on global variables. A function **argument** is a value passed as input during a function call. A function **parameter** is a variable representing the input in the function definition. Note: The terms "argument" and "parameter" are sometimes used interchangeably in conversation and documentation.

CHECKPOINT

Global and local variables in a program with a function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-4-parameters)

CONCEPTS IN PRACTICE

Arguments and parameters

Consider the following:

```
1  def print_welcome(name):
2        print(f"Welcome {name}!")
3
4     username = int(input("Enter new username: "))
5     print_welcome(username)
```

1. Which is an argument?
   a. `name`
   b. `username`
   c. `name` and `username`

2. Which is a parameter?
   a. `name`
   b. `username`
   c. `name` and `username`

3. What is the scope of `name`?
   a. `print_welcome()` only
   b. whole program

4. What would happen if `username` was changed to `name` on lines 4 and 5?
   a. same output
   b. error/wrong output

Multiple arguments and parameters

Functions can have multiple parameters. Ex: A function uses two parameters, length and width, to compute the square footage of a room. Function calls must use the correct order and number of arguments to avoid undesired behavior and errors (unless using optional or keyword arguments as discussed later).

EXAMPLE 6.1

Using multiple arguments in a function call

```
def print_div(op_1, op_2):
  """ Prints division operation """
  print(f"{op_1}/{op_2} = {op_1/op_2}")

num_1 = 6
num_2 = 3
print_div(num_1, num_2) # Prints "6/3 = 2.0"
print_div(num_2, num_1) # Prints "3/6 = 0.5"
print_div(num_1) # Error: Missing argument: op_2
```

CONCEPTS IN PRACTICE

Multiple arguments and parameters

Consider the following:

```
def calc_distance(x1, y1, x2, y2):
  dist = math.sqrt((x2-x1)**2 + (y2-y1)**2)
  print(dist)

p1_x =int(input("Enter point 1's x: "))
p1_y =int(input("Enter point 1's y: "))
p2_x =int(input("Enter point 2's x: "))
p2_y =int(input("Enter point 2's y: "))
calc_distance(p1_x, p1_y, p2_x, p2_y)
```

5. Which is an argument?
   a. `p1_x`
   b. `x1`

6. Which is a parameter?
   a. `p1_y`
   b. `y1`

7. What would be the value of x2 for the function call, `calc_distance(2, 4, 3, 6)`?
   a. `2`
   b. `4`
   c. `3`
   d. `6`
   e. `Error`

Modifying arguments and mutability

In Python, a variable is a name that refers to an object stored in memory, aka an object reference, so Python

uses a **pass-by-object-reference** system. If an argument is changed in a function, the changes are kept or lost depending on the object's mutability. A **mutable** object can be modified after creation. A function's changes to the object then appear outside the function. An **immutable** object cannot be modified after creation. So a function must make a local copy to modify, and the local copy's changes don't appear outside the function.

Programmers should be cautious of modifying function arguments as these side effects can make programs difficult to debug and maintain.

EXAMPLE 6.2

Converting temperatures

What are the values of `weekend_temps` and `type` after `convert_temps()` finishes?

```
def convert_temps(temps, unit):
  if unit == "F":
    for i in range(len(temps)):
      temps[i] = (temps[i]-32) * 5/9
    unit = "C"
  else:
    for i in range(len(temps)):
      temps[i] = (temps[i]*9/5) + 32
    unit = "F"

# Weekend temperatures in Fahrenheit.
wknd_temps = [49.0, 51.0, 44.0]
deg_sign = u"\N{DEGREE SIGN}" # Unicode
metric = "F"

# Convert from Fahrenheit to Celsius.
convert_temps(wknd_temps, metric)
for temp in wknd_temps:
  print(f"{temp:.2f}{deg_sign}{metric}", end=" ")
```

The output is `9.44`°F `10.56`°F `6.67`°F. `type` was changed to `"C"` in the function but didn't keep the change outside of the function. Why is the list argument change kept and not the string argument change? (Hint: A list is mutable. A string is immutable.)

CHECKPOINT

Exploring a faulty function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-4-parameters)

CONCEPTS IN PRACTICE

Mutability and function arguments

8. In `convert_temps()`, `wknd_temps` and `temps` refer to _____ in memory.
   a. the same object
   b. different objects

9. After `unit` is assigned with `"C"`, `metric` and `unit` refer to _____ in memory.
   a. the same object
   b. different objects

10. `deg_sign` is a string whose value cannot change once created. `deg_sign` is _____.
   a. immutable
   b. mutable

11. On line 16, `unit` _____.
   a. refers to an object with the value `"C"`
   b. does not exist

TRY IT

Printing right triangle area

Write a function, `print_area()`, that takes in the base and height of a right triangle and prints the triangle's area. The area of a right triangle is $\frac{bh}{2}$, where b is the base and h is the height.

Given input:

```
3
4
```

The output is:

```
Triangle area: 6.0
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-4-parameters)

TRY IT

Curving scores

Write a function, `print_scores()`, that takes in a list of test scores and a number representing how many

points to add. For each score, print the original score and the sum of the score and bonus. Make sure not to change the list.

Given function call:

```
print_scores([67, 68, 72, 71, 69], 10)
```

The output is:

```
67 would be updated to 77
68 would be updated to 78
72 would be updated to 82
71 would be updated to 81
69 would be updated to 79
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-4-parameters)

6.5 Return values

Learning objectives

By the end of this section you should be able to

- Identify a function's return value.
- Employ return statements in functions to return values.

Returning from a function

When a function finishes, the function returns and provides a result to the calling code. A **return statement** finishes the function execution and can specify a value to return to the function's caller. Functions introduced so far have not had a `return` statement, which is the same as returning `None`, representing no value.

CHECKPOINT

Returning a value from a function

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-5-return-values)

CONCEPTS IN PRACTICE

Using return statements

1. What is returned by `calc_mpg(miles, gallons)`?

```
def calc_mpg(miles, gallons):
```

```
  mpg = miles/gallons
  return mpg
```

a. mpg
b. None
c. Error

2. What is returned by `calc_sqft()`?

```
def calc_sqft(length, width):
  sqft = length * width
  return
```

a. sqft
b. None
c. Error

3. What is the difference between `hw_1()` and `hw_2()`?

```
def hw_1():
  print("Hello world!")
  return

def hw_2():
  print("Hello world!")
```

a. `hw_1()` returns a string, `hw_2()` does not
b. `hw_1()` returns None, `hw_2()` does not
c. no difference

Using multiple return statements

Functions that have multiple execution paths may use multiple `return` statements. Ex: A function with an `if-else` statement may have two `return` statements for each branch. `Return` statements always end the function and return control flow to the calling code.

In the table below, `calc_mpg()` takes in miles driven and gallons of gas used and calculates a car's miles per gallon. `calc_mpg()` checks if gallons is `0` (to avoid division by 0), and if so, returns `-1`, a value often used to

indicate a problem.

```
def calc_mpg(miles, gallons):
  if gallons > 0:
    mpg = miles/gallons
    return mpg
  else:
    print("Gallons can't be 0")
    return -1

car_1_mpg = calc_mpg(448, 16)
print("Car 1's mpg is", car_1_mpg)
car_2_mpg = calc_mpg(300, 0)
print("Car 2's mpg is", car_2_mpg)
```

```
Car 1's mpg is 28.0
Gallons can't be 0
Car 2's mpg is -1
```

Table 6.1 Calculating miles-per-gallon and checking for division by zero.

CONCEPTS IN PRACTICE

Multiple return statements

4. What does `yarn_weight(3)` return?

```
def yarn_weight(num):
  if num == 0:
    return "lace"
  elif num == 1:
    return "sock"
  elif num == 2:
    return "sport"
  elif num == 3:
    return "dk"
  elif num == 4:
    return "worsted"
  elif num == 5:
    return "bulky"
  else:
    return "super bulky"
```

a. `"lace"`
b. `"dk"`
c. `"super bulky"`

5. What is the output?

```
def inc_volume(level, max):
```

```
    if level < max:
      return level
      level += 1
    else:
      return level

vol1 = inc_volume(9, 10)
print(vol1)
vol2 = inc_volume(10, 10)
print(vol2)
```

a. 9
   10
b. 10
   10
c. 10
   11

Using functions as values

Functions are objects that evaluate to values, so function calls can be used in expressions. A function call can be combined with other function calls, variables, and literals as long as the return value is compatible with the operation.

CHECKPOINT

Using function calls in expressions

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-5-return-values)

CONCEPTS IN PRACTICE

Using function values

6. What is the updated value of `bill`?

```
def tax(total):
  return .06 * total

def auto_tip(total):
  return .2 * total

bill = 100.0
bill += tax(bill) + auto_tip(bill)
```

a. 26.0

b. `126.0`

7. What is the value of `val2`?

```
def sq(num):
  return num * num

def offset(num):
  return num - 2

val = 5
val2 = sq(offset(val))
```

a. `9`
b. `23`

TRY IT

Estimated days alive

Write a function, `days_alive()`, that takes in an age in years and outputs the estimated days the user has been alive as an integer. Assume each year has 365.24 days. Use `round()`, which takes a number and returns the nearest whole number.

Then write a main program that reads in a user's age and outputs the result of `days_alive()`.

Given input:

```
21
```

The output is:

```
You have been alive about 7670 days.
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-5-return-values)

TRY IT

Averaging lists

Write a function, `avg_list()`, that takes in a list and returns the average of the list values.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-5-return-values)

6.6 Keyword arguments

Learning objectives

By the end of this section you should be able to

- Describe the difference between positional and keyword arguments.
- Create functions that use positional and keyword arguments and default parameter values.

Keyword arguments

So far, functions have been called using **positional arguments**, which are arguments that are assigned to parameters in order. Python also allows **keyword arguments**, which are arguments that use parameter names to assign values rather than order. When mixing positional and keyword arguments, positional arguments must come first in the correct order, before any keyword arguments.

CHECKPOINT

Using keyword arguments

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-6-keyword-arguments)

CONCEPTS IN PRACTICE

Using keyword and positional arguments

Consider the following function:

```
def greeting(msg, name, count):
  i = 0
  for i in range(0, count):
    print(msg, name)
```

1. Which is the positional argument in `greeting(count=1, name="Ash", msg="Hiya")`?
   a. `count=1`
   b. `name="Ash"`
   c. `msg="Hiya"`
   d. None

2. What is the output of `greeting(count=2, name="Ash", msg="Hiya")`?
   a. `Ash Hiya`
      `Ash Hiya`
   b. `Hiya Ash`
      `Hiya Ash`

3. Which is the positional argument in `greeting("Welcome", count=1, name="Anita")`?
   a. `"Welcome"`
   b. `count=1`

c. `name="Anita"`

4. Which function call would produce an error?
   a. `greeting("Morning", "Morgan", count=3)`
   b. `greeting(count=1,"Hi", "Bea")`
   c. `greeting("Cheers", "Colleagues", 10)`

Default parameter values

Functions can define default parameter values to use if a positional or keyword argument is not provided for the parameter. Ex: `def season(m, d, hemi="N"):` defines a default value of `"N"` for the hemi parameter. Note: Default parameter values are only defined once to be used by the function, so mutable objects (such as lists) should not be used as default values.

The physics example below calculates weight as a force in newtons given mass in kilograms and acceleration in $\frac{m}{s^2}$. Gravity on Earth is 9.8 $\frac{m}{s^2}$, and gravity on Mars is 3.7 $\frac{m}{s^2}$.

CHECKPOINT

Using default parameter values

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-6-keyword-arguments)

CONCEPTS IN PRACTICE

Using default parameter values

Consider the following updated version of `greeting()`:

```
def greeting(msg, name="Friend", count=1):
  i = 0
  for i in range(0, count):
    print(msg, name)
```

5. Which parameters have default values?
   a. `msg`
   b. `name` and `count`
   c. all

6. Which function call is correct?
   a. `greeting()`
   b. `greeting(name="Gina")`
   c. `greeting("Greetings")`

7. What is the output of `greeting(count=0, msg="Hello")`?

a. `Hello Friend`
b. nothing
c. `Error`

PEP 8 RECOMMENDATIONS: SPACING

The PEP 8 style guide recommends no spaces around = when indicating keyword arguments and default parameter values.

TRY IT

Stream donations

Write a function, `donate()`, that lets an online viewer send a donation to a streamer. `donate()` has three parameters:

- `amount`: amount to donate, default value: 5
- `name`: donor's name, default value: "Anonymous"
- `msg`: donor's message, default value: ""

Given:

```
donate(10, "gg")
```

The output is:

```
Anonymous donated 10 credits: gg
```

Write function calls that use the default values along with positional and keyword arguments.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/6-6-keyword-arguments)

6.7 Chapter summary

Highlights from this chapter include:

- Functions are named blocks of code that perform tasks when called and make programs more organized and optimized.
- Control flow is the sequence of program execution. Control flow moves between calling code and function code when a function is called.
- Variable scope refers to where a variable can be accessed. Global variables can be accessed anywhere in a program. Local variables are limited in scope, such as to a function.
- Parameters are function inputs defined with the function. Arguments are values passed to the function as

input by the calling code. Parameters are assigned with the arguments' values.
- Function calls can use positional arguments to map values to parameters in order.
- Function calls can use keyword arguments to map values using parameter names in any order.
- Functions can define default parameter values to allow for optional arguments in function calls.
- Python uses a pass-by-object-reference system to assign parameters with the object values referenced by the arguments.
- Functions can use `return` statements to return values back to the calling code.

At this point, you should be able to write functions that have any number of parameters and return a value, and programs that call functions using keyword arguments and optional arguments.

Construct	Description

Function definition	`def function_name():`
	`  """Docstring"""`
	`  # Function body`

Parameter	`def function_name(parameter_1):` `  # Function body`

Argument	`def function_name(parameter_1):`
	`  # Function body`
	
	`function_name(argument_1)`

Return statement	`def function_name():`
	`  # Function body`
	`  return result # Returns the value of result to the caller`

Variables (scope)	`def function_name(parameter_1):`
	`  # Function body`
	`  local_var = parameter_1 * 5`
	`  return local_var`
	
	`global_var = function_name(arg_1)`

Table 6.2 Chapter 6 reference.

Construct	Description

Keyword arguments	`def function_name(parameter_1, parameter_2):`
	`  # Function body`
	
	`function_name(parameter_2 = 5, parameter_1 = 2)`

Default parameter value	`def function_name(parameter_1 = 100):` `  # Function body`

Table 6.2 Chapter 6 reference.

7 Modules

Figure 7.1 credit: modification of work "Lone Pine Sunset", by Romain Guy/Flickr, Public Domain

Chapter Outline

Introduction

As programs get longer and more complex, organizing the code into modules is helpful. This chapter shows how to define, import, and find new modules. Python's standard library (https://openstax.org/r/100pythlibrary) provides over 200 built-in modules (https://openstax.org/r/100200modules). Hundreds of thousands of other modules (https://openstax.org/r/100pypi) are available online.

A **module** is a *.py* file containing function definitions and other statements. The module's name is the file's name without the *.py* extension at the end. Ex: The following code, written in a file named *greetings.py*, defines a module named `greetings`.

```
"""Functions that print standard greetings."""

def hello():
  print("Hello!")

def bye():
  print("Goodbye!")
```

Technically, every program in this book is a module. But not every module is designed to run like a program. Running *greetings.py* as a program would accomplish very little. Two functions would be defined, but the functions would never be called. These functions are intended to be called in other modules.

7.1 Module basics

Learning objectives

By the end of this section you should be able to

- Write a module that consists only of function definitions.
- Import the module and use the functions in a program.

Defining a module

Modules are defined by putting code in a *.py* file. The `area` module below is in a file named area.py. This module provides functions for calculating area.

EXAMPLE 7.1

The area module

```
"""Functions to calculate the area of geometric shapes."""

import math

# 2D shapes

def square(side):
    """Gets the area of a square."""
    return side**2

def rectangle(length, width):
    """Gets the area of a rectangle."""
    return length * width

def triangle(base, height):
    """Gets the area of a triangle."""
    return 0.5 * base * height

def trapezoid(base1, base2, height):
    """Gets the area of a trapezoid."""
    return 0.5 * (base1 + base2) * height

def circle(radius):
    """Gets the area of a circle."""
    return math.pi * radius**2

def ellipse(major, minor):
```

```
    """Gets the area of an ellipse."""
    return math.pi * major * minor

# 3D shapes

def cube(side):
    """Gets the surface area of a cube."""
    return 6 * side**2

def cylinder(radius, height):
    """Gets the surface area of a cylinder."""
    return 2 * math.pi * radius * (radius + height)

def cone(radius, height):
    """Gets the surface area of a cone."""
    return math.pi * radius * (radius + math.hypot(height, radius))

def sphere(radius):
    """Gets the surface area of a sphere."""
    return 4 * math.pi * radius**2
```

CONCEPTS IN PRACTICE

Defining a module

1. How many functions are defined in the `area` module?
   a. 6
   b. 10
   c. 49

2. What would be the result of running *area.py* as a program?
   a. Functions would be defined and ready to be called.
   b. Nothing; the module has no statements to be run.
   c. `SyntaxError`

3. What is the return value of `cube(5)`?
   a. `60`
   b. `125`
   c. `150`

Importing a module

The module defined in *area.py* can be used in other programs. When importing the `area` module, the suffix .py is removed:

```
import area

print("Area of a basketball court:", area.rectangle(94, 50))
print("Area of a circus ring:", area.circle(21))
```

The output is:

```
Area of a basketball court: 4700
Area of a circus ring: 1385.4423602330987
```

CHECKPOINT

Importing area in a Python shell

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-1-module-basics)

CONCEPTS IN PRACTICE

Importing a module

4. What statement would import a module from a file named *secret.py*?
   a. `import secret`
   b. `import secret.py`
   c. `import "secret.py"`

5. What code would return the area of a circle with a radius of 3 meters?
   a. `circle(3)`
   b. `area.circle(3)`
   c. `circle.area(3)`

6. A programmer would like to write a function that calculates the area of a hexagon. Where should the function be written?
   a. the main program
   b. the area module
   c. the secret module

TRY IT

Conversion module

Write a module that defines the following functions:

1. `cel2fah(c)` –
   Converts a temperature in Celsius to Fahrenheit.

The formula is `9/5 * c + 32`.

2. `fah2cel(f)` –
   Converts a temperature in Fahrenheit to Celsius.
   The formula is `5/9 * (f - 32)`.
3. `km2mi(km)` –
   Converts a distance in kilometers to miles.
   The formula is `km / 1.60934`.
4. `mi2km(mi)` –
   Converts a distance in miles to kilometers.
   The formula is `mi * 1.60934`.

Each function should include a docstring as the first line. A docstring for the module has been provided for you.

The module should not do anything except define functions. When you click the "Run" button, the module should run without error. No output should be displayed.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-1-module-basics)

TRY IT

European vacation

Write a program that uses the `conversion` module from the previous exercise to complete a short story. The program's output should match the following example (input in bold):

```
How fast were you driving? 180
Woah, that's like 112 mph!
What was the temperature? 35
That's 95 degrees Fahrenheit!
```

Notice this exercise requires two files:

1. *european.py*, the main program. Input and output statements are provided as a starting point. Edit the lines with TODO comments to use the conversion module.
2. *conversion.py*, the other module. Copy and paste your code from the previous exercise. Import this module in *european.py* after the docstring.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-1-module-basics)

7.2 Importing names

Learning objectives

By the end of this section you should be able to

- Import functions from a module using the `from` keyword.
- Explain how to avoid a name collision when importing a module.

The from keyword

Specific functions in a module can be imported using the `from` keyword:

```
from area import triangle, cylinder
```

These functions can be called directly, without referring to the module:

```
print(triangle(1, 2))
print(cylinder(3, 4))
```

EXPLORING FURTHER

As shown below, the `from` keyword can lead to confusing names.

CHECKPOINT

Importing functions

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-2-importing-names)

CONCEPTS IN PRACTICE

The from keyword

1. Which import statement would be needed before running `print(sqrt(25))`?
   a. `from math import sqrt`
   b. `import sqrt from math`
   c. `import math`

2. How many variables are defined by the statement `from math import sin, cos, tan`?
   a. 0
   b. 3
   c. 4

3. What error would occur if attempting to call a function that was not imported?
   a. `ImportError`
   b. `NameError`
   c. `SyntaxError`

Name collisions

Modules written by different programmers might use the same name for a function. A **name collision** occurs when a function is defined multiple times. If a function is defined more than once, the most recent definition is

used:

```
from area import cube

def cube(x):    # Name collision (replaces the imported function)
  return x ** 3

print(cube(2)) # Calls the local cube() function, not area.cube()
```

A programmer might not realize the cube function is defined twice because no error occurs when running the program. Name collisions are not considered errors and often lead to unexpected behavior.

Care should be taken to avoid name collisions. Selecting specific functions from a module to import reduces the memory footprint; however, importing a complete module can help to avoid collisions because a *module.name* format would be used. This is a tradeoff the programmer must consider.

CHECKPOINT

Module and function names

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-2-importing-names)

CONCEPTS IN PRACTICE

Name collisions

4. A program begins with `from area import square, circle`. What code causes a name collision?

a.
```
def area(phone):
  """Gets the area code of a phone number."""
```

b.
```
def circle(x, y, size):
  """Draws a circle centered at (x, y)."""
```

c.
```
def is_square(length, width):
  """Returns True if length and width are equal."""
```

5. A program begins with `import area`. What code causes a name collision?

a.
```
area = 51
```

b.
```
import cylinder from volume
```

c.
```
def cube(size):
  """Generates a "size X size" rubik's cube."""
```

6. Which line will cause an error?

```
1 def hello():
2   print("Hello!")
3
4 def hello(name):
5   print("Hello,", name)
```

```
6
7 hello()
8 hello("Chris")
```

a. line 4
b. line 7
c. line 8

EXPLORING FURTHER

If a name is defined, imported, or assigned multiple times, Python uses the most recent definition. Other languages allow multiple functions to have the same name if the parameters are different. This feature, known as **function overloading**, is not part of the Python language.

TRY IT

Missing imports

Add the missing import statements to the top of the file. Do not make any changes to the rest of the code. In the end, the program should run without errors.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-2-importing-names)

TRY IT

Party favors

The following program does not run correctly because of name collisions. Fix the program by modifying import statements, function calls, and variable assignments. The output should be:

```
Bouncy ball area: 13
Bouncy ball volume: 4
Cone hat area: 227
Cone hat volume: 209
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-2-importing-names)

7.3 Top-level code

Learning objectives

By the end of this section you should be able to

- Identify code that will run as a side effect of importing.
- Explain the purpose of `if __name__ == "__main__"`.

Side effects

Modules define functions and constants to be used in other programs. When importing a module, *all* code in the module is run from top to bottom. If a module is not designed carefully, unintended code might run as a side effect. The unintended code is generally at the top level, outside of function definitions.

CHECKPOINT

Sphere test code

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-3-top-level-code)

CONCEPTS IN PRACTICE

Side effects

1. Which line would cause a side effect when imported?

```
1 import math
2
3 print("Defining sphere function")
4
5 def sphere(radius):
6   """Gets the volume of a sphere."""
7   return 4/3 * math.pi * radius**3
```

   a. line 1
   b. line 3
   c. line 5

2. The following *volume.py* module causes a side effect.

```
import math

def sphere(radius):
  """Gets the volume of a sphere."""
  return 4/3 * math.pi * radius**3

for r in range(10000000):
  volume = sphere(r)
```

   a. true
   b. false

3. The following *greeting.py* module causes a side effect.

```
name = input("What is your name? ")
```

```
print(f"Nice to meet you, {name}!")
live = input("Where do you live? ")
print(f"{live} is a great place.")
```

a. true
b. false

Using __name__

Python modules often include the statement `if __name__ == "__main__"` to prevent side effects. This statement is true when the module is run as a program and false when the module is imported.

CHECKPOINT

The main module

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-3-top-level-code)

CONCEPTS IN PRACTICE

Using __name__

4. What is the output when **running** the following *test.py* module?

```
import math

print(math.__name__)
print(__name__)
```

a. >math
   test
b. __main__
   test
c. math
   __main__

5. What is the output when **importing** the following *test.py* module?

```
import math

print(math.__name__)
print(__name__)
```

a. math
   test
b. __main__
   test

c. `math`
`__main__`

6. What line is useful for preventing side effects when importing?
   a. `if __name__ == "main":`
   b. `if __name__ == __main__:`
   c. `if __name__ == "__main__":`

EXPLORING FURTHER

Variables that begin and end with double underscores have special meaning in Python. Double underscores are informally called "dunder" or "magic" variables. Other examples include `__doc__` (the module's docstring) and `__file__` (the module's filename).

TRY IT

Side effects

This exercise is a continuation of the "Missing imports" exercise. Previously, you added missing import statements to the top of the program. Now, modify the program to prevent side effects when importing the program as a module:

1. Add `if __name__ == "__main__"` at the end.
2. Move all test code under that `if` statement.

The program should run without errors and produce the same output as before.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-3-top-level-code)

TRY IT

Conversion test

This exercise is a continuation of the "Conversion module" exercise. Previously, you wrote the functions `cel2fah`, `fah2cel`, `km2mi`, and `mi2km`. Write test code at the end of *conversion.py* (the original file) for each of these functions. The test code must not run as a side effect when `conversion` is imported by other programs. When running *conversion.py* as the main program, the test output should be:

```
0 C is 32 F
5 C is 41 F
10 C is 50 F
15 C is 59 F
20 C is 68 F
```

```
20 F is -7 C
25 F is -4 C
30 F is -1 C
35 F is 2 C
40 F is 4 C

1 km is 0.6 mi
2 km is 1.2 mi
3 km is 1.9 mi
4 km is 2.5 mi
5 km is 3.1 mi

5 mi is 8.0 km
6 mi is 9.7 km
7 mi is 11.3 km
8 mi is 12.9 km
9 mi is 14.5 km
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-3-top-level-code)

7.4 The help function

Learning objectives

By the end of this section you should be able to

- Use the `help()` function to explore a module's contents.
- Identify portions of code included in the documentation.

Colors on websites

This section introduces an example module for working with HTML colors (https://openstax.org/r/100htmlcolors). HyperText Markup Language (HTML) is used to design websites and graphical applications. Web browsers like Chrome and Safari read HTML and display the corresponding contents. Ex: The HTML code `<p style="color: Red">Look out!</p>` represents a paragraph with red text.

HTML defines 140 standard color names (https://openstax.org/r/100colornames). Additional colors can be specified using a hexadecimal format: *#RRGGBB*. The digits RR, GG, and BB represent the red, green, and blue components of the color. Ex: *#DC143C* is 220 red + 20 green + 60 blue, which is the color `Crimson`.

Red, green, and blue values range from 0 to 255 (or 00 to FF in hexadecimal). Lower values specify darker colors, and higher values specify lighter colors. Ex: *#008000* is the color `Green`, and *#00FF00* is the color `Lime`.

CHECKPOINT

HTML color codes

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-4-the-help-function)

CONCEPTS IN PRACTICE

HTML color codes

1. What color is *#000080*?
   a. maroon red
   b. navy blue
   c. olive green

2. What is 255 in hexadecimal?
   a. 00
   b. 80
   c. FF

3. Which color is lighter?
   a. *#FFA500 (orange)*
   b. *#008000 (green)*

Example colors module

A module for working with HTML color codes would be helpful to graphic designers and web developers. The following Python code is in a file named *colors.py*.

- Line 1 is the docstring for the module.
- Lines 3–16 assign variables for frequently used colors.
- Lines 18–24 define a function to be used *within* the module.
- Lines 26–45 define functions to be used in other modules.

Note: The `tohex()` and `torgb()` functions use Python features (string formatting and slicing) described later in the book. For now, the documentation and comments are more important than the implementation details.

```
"""Functions for working with color names and hex/rgb values."""

# Primary colors
RED = "#FF0000"
YELLOW = "#FFFF00"
BLUE = "#0000FF"

# Secondary colors
ORANGE = "#FFA500"
GREEN = "#008000"
VIOLET = "#EE82EE"

# Neutral colors
BLACK = "#000000"
GRAY = "#808080"
WHITE = "#FFFFFF"

def _tohex(value):
```

```
    """Converts an integer to an 8-bit (2-digit) hexadecimal string."""
    if value <= 0:
      return "00"
    if value >= 255:
      return "FF"
    return format(value, "02X")

  def tohex(r, g, b):
    """Formats red, green, and blue integers as a color in hexadecimal."""
    return "#" + _tohex(r) + _tohex(g) + _tohex(b)

  def torgb(color):
    """Converts a color in hexadecimal to red, green, and blue integers."""
    r = int(color[1:3], 16) # First 2 digits
    g = int(color[3:5], 16) # Middle 2 digits
    b = int(color[5:7], 16) # Last 2 digits
    return r, g, b

  def lighten(color):
    """Increases the red, green, and blue values of a color by 32 each."""
    r, g, b = torgb(color)
    return tohex(r+32, g+32, b+32)

  def darken(color):
    """Decreases the red, green, and blue values of a color by 32 each."""
    r, g, b = torgb(color)
    return tohex(r-32, g-32, b-32)
```

CONCEPTS IN PRACTICE

The colors module

4. What are the components of the color YELLOW?
   a. red=0, green=255, blue=255
   b. red=255, green=255, blue=0
   c. red=255, green=0, blue=255

5. What code would return a darker shade of blue?
   a. `darken(BLUE)`
   b. `colors.darken(BLUE)`
   c. `colors.darken(colors.BLUE)`

6. What symbol indicates that a function is not intended to be called by other modules?
   a. underscore (_)
   b. number sign (#)
   c. colon (:)

Module documentation

The built-in `help()` function provides a summary of a module's functions and data. Calling `help(module_name)` in a shell is a convenient way to learn about a module.

EXAMPLE 7.2

Output of help(colors) in a shell

The documentation below is automatically generated from the docstrings in *colors.py*.

`help(colors)`

```
    Help on module colors:

    NAME
      colors - Functions for working with color names and hex/rgb values.

    FUNCTIONS
      darken(color)
        Decreases the red, green, and blue values of a color by 32 each.

      lighten(color)
        Increases the red, green, and blue values of a color by 32 each.

      tohex(r, g, b)
        Formats red, green, and blue integers as a color in hexadecimal.

      torgb(color)
        Converts a color in hexadecimal to red, green, and blue integers.

    DATA
      BLACK = '#000000'
      BLUE = '#0000FF'
      GRAY = '#808080'
      GREEN = '#008000'
      ORANGE = '#FFA500'
      RED = '#FF0000'
      VIOLET = '#EE82EE'
      WHITE = '#FFFFFF'
      YELLOW = '#FFFF00'

    FILE
      /home/student/Desktop/colors.py

    >>> help(colors)

    Help on module colors:

    NAME
```

```
    colors - Functions for working with color names and hex/rgb values.

FUNCTIONS
    darken(color)
        Decreases the red, green, and blue values of a color by 32 each.

    lighten(color)
        Increases the red, green, and blue values of a color by 32 each.

    tohex(r, g, b)
        Formats red, green, and blue integers as a color in hexadecimal.

    torgb(color)
        Converts a color in hexadecimal to red, green, and blue integers.

DATA
    BLACK = '#000000'
    BLUE = '#0000FF'
    GRAY = '#808080'
    GREEN = '#008000'
    ORANGE = '#FFA500'
    RED = '#FF0000'
    VIOLET = '#EE82EE'
    WHITE = '#FFFFFF'
    YELLOW = '#FFFF00'

FILE
    /home/student/Desktop/colors.py
```

CONCEPTS IN PRACTICE

The help() function

7. The documentation includes comments from the source code.
   a. true
   b. false

8. In what order are the functions listed in the documentation?
   a. alphabetical order
   b. definition order
   c. random order

9. Which function defined in *colors.py* is not included in the documentation?
   a. `_tohex`
   b. `tohex`

c. `torgb`

TRY IT

Help on modules

The `random` and `statistics` modules are useful for running scientific experiments. You can become familiar with these two modules by skimming their documentation.

Open a Python shell on your computer, or use the one at python.org/shell (https://openstax.org/r/100pythonshell). Type the following lines, one at a time, into the shell.

- `import random`
- `help(random)`
- `import statistics`
- `help(statistics)`

Many shell environments, including the one on python.org, display the output of `help()` one page at a time. Use the navigation keys on the keyboard (up/down arrows, page up/down, home/end) to read the documentation. When you are finished reading, press the Q key ("quit") to return to the Python shell.

TRY IT

Help on functions

The `help()` function can be called on specific functions in a module. Open a Python shell on your computer, or use the one at python.org/shell (https://openstax.org/r/100pythonshell). Type the following lines, one at a time, into the shell.

- `import random`
- `help(random.randint)`
- `help(random.choice)`
- `import statistics`
- `help(statistics.median)`
- `help(statistics.mode)`

Remember to use the navigation keys on the keyboard, and press the Q key ("quit") to return to the Python shell.

7.5 Finding modules

Learning objectives

By the end of this section you should be able to

- Explain differences between the standard library and PyPI.
- Search python.org and pypi.org for modules of interest.

Built-in modules

The **Python Standard Library** is a collection of built-in functions and modules that support common programming tasks. Ex: The `math` module provides functions like `sqrt()` and constants like `pi`. Python's official documentation includes a library reference (https://openstax.org/r/100pythlibrary) and a module index (https://openstax.org/r/100200modules) for becoming familiar with the standard library.

For decades, Python has maintained a "batteries included (https://openstax.org/r/100batteries)" philosophy. This philosophy means that the standard library should come with everything most programmers need. In fact, the standard library includes over 200 built-in modules!

Module	Description
`calendar`	General calendar-related functions.
`datetime`	Basic date and time types and functions.
`email`	Generate and process email messages.
`math`	Mathematical functions and constants.
`os`	Interact with the operating system.
`random`	Generate pseudo-random numbers.
`statistics`	Mathematical statistics functions.
`sys`	System-specific parameters and functions.
`turtle`	Educational framework for simple graphics.
`zipfile`	Read and write ZIP-format archive files.

Table 7.1 Example built-in modules in the standard library.

CONCEPTS IN PRACTICE

Built-in modules

Use the library reference, module index, and documentation links above to answer the questions.

1. Which page provides a list of built-in modules sorted by category?
   a. library reference
   b. module index
   c. PEP 2

2. What is the value of `calendar.SUNDAY`?
   a. 1
   b. 6
   c. 7

3. Which built-in module enables the development of graphical user interfaces?
   a. `tkinter`
   b. `turtle`
   c. `webbrowser`

Third-party modules

The **Python Package Index** (PyPI), available at pypi.org (https://openstax.org/r/100pypi), is the official third-party software library for Python. The abbreviation "PyPI" is pronounced like *pie pea eye* (in contrast to PyPy (https://openstax.org/r/100pypy), a different project).

PyPI allows anyone to develop and share modules with the Python community. Module authors include individuals, large companies, and non-profit organizations. PyPI helps programmers install modules and receive updates.

Most software available on PyPI is free and open source. PyPI is supported by the Python Software Foundation (https://openstax.org/r/100foundation) and is maintained by an independent group of developers.

Module	Description
`arrow`	Convert and format dates, times, and timestamps.
`BeautifulSoup`	Extract data from HTML and XML documents.
`bokeh`	Interactive plots and applications in the browser.
`matplotlib`	Static, animated, and interactive visualizations.
`moviepy`	Video editing, compositing, and processing.
`nltk`	Natural language toolkit for human languages.
`numpy`	Fundamental package for numerical computing.
`pandas`	Data analysis, time series, and statistics library.
`pillow`	Image processing for jpg, png, and other formats.

Table 7.2 Example third-party modules available from PyPI.

Module	Description
`pytest`	Full-featured testing tool and unit test framework.
`requests`	Elegant HTTP library for connecting to web servers.
`scikit-learn`	Simple, efficient tools for predictive data analysis.
`scipy`	Fundamental algorithms for scientific computing.
`scrapy`	Crawl websites and scrape data from web pages.
`tensorflow`	End-to-end machine learning platform for everyone.

Table 7.2 Example third-party modules available from PyPI.

CONCEPTS IN PRACTICE

Third-party modules

Use pypi.org and the links in the table above to answer the questions.

4. Which modules can be used to edit pictures and videos?
   a. `BeautifulSoup` and `Scrapy`
   b. `Bokeh` and `Matplotlib`
   c. `MoviePy` and `Pillow`

5. Which third-party module is a replacement for the built-in `datetime` module?
   a. `arrow`
   b. `calendar`
   c. `time`

6. Search for the webcolors module on PyPI. What function provided by webcolors looks up the color name for a hex code?
   a. `hex_to_name`
   b. `name_to_hex`
   c. `normalize_hex`

EXPLORING FURTHER

Programming blogs often highlight PyPI modules to demonstrate the usefulness of Python. The following examples provide more background information about the modules listed above.

- Top 20 Python Libraries for Data Science for 2023 (https://openstax.org/r/100simplelearn)
- 24 Best Python Libraries You Should Check in 2022 (https://openstax.org/r/100library2022)

- Most Popular Python Packages in 2021 (https://openstax.org/r/100packages2021)

TRY IT

Happy birthday

Module documentation pages often include examples to help programmers become familiar with the module. For this exercise, refer to the following examples from the `datetime` module documentation:

- Examples of Usage: date (https://openstax.org/r/100dateexamples)
- Examples of Usage: timedelta (https://openstax.org/r/100timedeltaex)

Write a program that creates a `date` object representing your birthday. Then get a `date` object representing today's date (the date the program is run). Calculate the difference between the two dates, and output the results in the following format:

```
Your birth date: 2005-03-14
Today's date is: 2023-06-01

You were born 6653 days ago
(that is 574819200 seconds)

You are about 18 years old
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/7-5-finding-modules)

TRY IT

More exact age

The `datetime` module does not provide a built-in way to display a person's exact age. Ex: The following program calculates an exact age (in years and days) using floor division and modulo. The output is: `You are 15 years and 4 days old.`

```
from datetime import date

birth = date(2005, 3, 14)
today = date(2020, 3, 14) # 15 years later
delta = today - birth

years = delta.days // 365
days = delta.days % 365
print("You are", years, "years and", days, "days old")
```

Notice how leap years are included in the calculation. February 29th occurs four times between `birth` and `today`. Therefore, the user is not only 15 years old, but 15 years and 4 days old.

Many commonly used modules from PyPI, including `arrow`, are installed in the Python shell at python.org/shell (https://openstax.org/r/100pythonshell). Open the Python shell and type the following lines:

```
import arrow
birth = arrow.get(2005, 3, 14)
birth.humanize()
```

Refer to the humanize() (https://openstax.org/r/100humanize) examples from the `arrow` module documentation. In the Python shell, figure out how to display the number of years and days since `birth` using *one line of code*. Then display the number of years, months, and days since `birth`. Finally, use the `print()` function to output the results in this format: `You are 18 years 4 months and 7 days old`.

As time permits, experiment with other functions provided by the `arrow` module.

7.6 Chapter summary

Highlights from this chapter include:

- Programs can be organized into multiple *.py* files (modules). The `import` keyword allows a program to use functions defined in another *.py* file.
- The `from` keyword can be used to import specific functions from a module. However, programs should avoid importing (or defining) multiple functions with the same name.
- Modules often include the line `if __name__ == "__main__"` to prevent code from running as a side effect when the module is imported by other programs.
- When working in a shell, the `help()` function can be used to look up the documentation for a module. The documentation is generated from the docstrings.
- Python comes with over 200 built-in modules and hundreds of thousands of third-party modules. Programmers can search for modules on docs.python.org (https://openstax.org/r/100docstrings) and pypi.org (https://openstax.org/r/100pypi).

Statement	Description
`import module`	Imports a module for use in another program.
`from module import function`	Imports a specific function from a module.
`if __name__ == "__main__":`	A line of code found at the end of many modules. This statement indicates what code to run if the module is executed as a program (in other words, what code *not* to run if this module is imported by another program).

Table 7.3 Chapter 7 reference.

Statement	Description
`help(module_name)`	Shows the documentation for the given module. The documentation includes the module's docstring, followed by a list of functions defined in the module, followed by a list of global variables assigned in the module, followed by the module's file name.
`help(function_name)`	Shows the docstring for the given function.
`date(2023, 2, 14)`	Creates a `date` object representing February 14, 2023. Requires: `from datetime import date`.

Table 7.3 Chapter 7 reference.

8 Strings

Figure 8.1 credit: modification of work "Project 366 #65: 050316 A Night On The Tiles", by Pete/Flickr, CC BY 2.0

Chapter Outline

Introduction

A **string** is a sequence of characters. Python provides useful methods for processing string values. In this chapter, string methods will be demonstrated including comparing string values, string slicing, searching, testing, formatting, and modifying.

8.1 String operations

Learning objectives

By the end of this section you should be able to

- Compare strings using logical and membership operators.
- Use `lower()` and `upper()` string methods to convert string values to lowercase and uppercase characters.

String comparison

String values can be compared using logical operators (<, <=, >, >=, ==, !=) and membership operators (`in` and `not in`). When comparing two string values, the matching characters in two string values are compared sequentially until a decision is reached. For comparing two characters, ASCII values are used to apply logical operators.

Operator	Description	Example	Output	Explanation
> or >=	Checks whether the first string value is greater than (or greater than or equal to) the second string value.	`"c" > "d"`	`False`	When comparing `"c"` operand to `"d"` operand, the ASCII value for `"c"` is smaller than the ASCII value for `"d"`. Therefore, `"c" < "d"`. The expression `"c" > "d"` evaluates to False.
< or <=	Checks whether the first string value is less than (or less than or equal to) the second string value.	`"ab" < "ac"`	`True`	When comparing `"ab"` operand to `"ac"` operand, the first characters are the same, but the second character of `"ab"` is less than the second character in `"ac"` and as such `"ab" < "ac"`.
==	Checks whether two string values are equal.	`"aa" == "aa"`	`True`	Since all characters in the first operand and the second operand are the same, the two string values are equal.
!=	Checks whether two string values are not equal.	`"a" != "b"`	`True`	The two operands contain different string values (`"a"` vs. `"b"`), and the result of checking whether the two are not the same evaluates to True.
`in`	Checks whether the second operand contains the first operand.	`"a" in "bc"`	`False`	Since string `"bc"` does not contain string `"a"`, the output of `"a" in "bc"` evaluates to False.
`not in`	Checks whether the second operand does not contain the first operand.	`"a" not in "bc"`	`True`	Since string `"bc"` does not contain string `"a"`, the output of `"a" not in "bc"` evaluates to True.

Table 8.1 Comparing string values.

CONCEPTS IN PRACTICE

Using logical and membership operators to compare string values

1. What is the output of (`"aaa" < "aab"`)?
   a. `True`
   b. `False`

2. What is the output of (`"aa" < "a"`)?
   a. `True`
   b. `False`

3. What is the output of `("aples" in "apples")`?
   a. `undefined`
   b. `True`
   c. `False`

lower() and upper()

Python has many useful methods for modifying strings, two of which are `lower()` and `upper()` methods. The **lower()** method returns the converted alphabetical characters to lowercase, and the **upper()** method returns the converted alphabetical characters to uppercase. Both the `lower()` and `upper()` methods do *not* modify the string.

EXAMPLE 8.1

Converting characters in a string

In the example below, the `lower()` and `upper()` string methods are called on the string variable x to convert all characters to lowercase and uppercase, respectively.

```
x = "Apples"

# The lower() method converts a string to all lowercase characters
print(x.lower())

# The upper() method converts a string to all uppercase characters
print(x.upper())
```

The above code's output is:

```
apples
APPLES
```

CONCEPTS IN PRACTICE

Using lower() and upper()

4. What is the output of `"aBbA".lower()`?
   a. `abba`
   b. `ABBA`
   c. `abbA`

5. What is the output of `"aBbA".upper()`?
   a. `abba`

b. ABBA
c. ABbA
d. aBBA

6. What is the output of `("a".upper() == "A")`?
   a. `True`
   b. `False`

TRY IT

Number of characters in the string

A string variable, `s_input`, is defined. Use `lower()` and `upper()` to convert the string to lowercase and uppercase, and print the results in the output. Also, print the number of characters in the string, including space characters.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-1-string-operations)

TRY IT

What is my character?

Given the string, `s_input`, which is a one-character string object, if the character is between `"a"` and `"t"` or `"A"` and `"T"`, print `True`. Otherwise, print `False`.
Hint: You can convert `s_input` to lowercase and check if `s_input` is between `"a"` and `"t"`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-1-string-operations)

8.2 String slicing

Learning objectives

By the end of this section you should be able to

- Use string indexing to access characters in the string.
- Use string slicing to get a substring from a string.
- Identify immutability characteristics of strings.

String indexing

A string is a type of sequence. A string is made up of a sequence of characters indexed from left to right, starting at 0. For a string variable s, the left-most character is indexed 0 and the right-most character is indexed `len(s) - 1`. Ex: The length of the string `"Cloud"` is 5, so the last index is 4.

Negative indexing can also be used to refer to characters from right to left starting at -1. For a string variable s, the left-most character is indexed `-len(s)` and the right-most character is indexed -1. Ex: The length of the string `"flower"` is 6, so the index of the first character with negative indexing is -6.

CHECKPOINT

String indexing

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-2-string-slicing)

CONCEPTS IN PRACTICE

Accessing characters in a string using indexing

1. Which character is at index 1 in the string `"hello"`?
   a. `"h"`
   b. `"e"`
   c. `"o"`

2. What is the character at index -2 in the string `"Blue"`?
   a. `"e"`
   b. `"u"`
   c. `"l"`

3. What is the output of the following code?

```
word = "chance"
print(word[-1] == word[5])
```

   a. `True`
   b. `False`

String slicing

String slicing is used when a programmer must get access to a sequence of characters. Here, a string slicing operator can be used. When `[a:b]` is used with the name of a string variable, a sequence of characters starting from index a (inclusive) up to index b (exclusive) is returned. Both a and b are optional. If a or b are not provided, the default values are 0 and `len(string)`, respectively.

EXAMPLE 8.2

Getting the minutes

Consider a time value is given as `"hh:mm"` with `"hh"` representing the hour and `"mm"` representing the minutes. To retrieve only the string's minutes portion, the following code can be used:

```
time_string = "13:46"
minutes = time_string[3:5]
print(minutes)
```

The above code's output is:

```
46
```

EXAMPLE 8.3

Getting the hour

Consider a time value is given as `"hh:mm"` with `"hh"` representing the hour and `"mm"` representing the minutes. To retrieve only the string's hour portion, the following code can be used:

```
time_string = "14:50"
hour = time_string[:2]
print(hour)
```

The above code's output is:

```
14
```

CONCEPTS IN PRACTICE

Getting a substring using string slicing

4. What is the output of the following code?

```
a_string = "Hello world"
print(a_string[2:4])
```

   a. `"el"`
   b. `"ll"`
   c. `"llo"`

5. What is the output of the following code?

```
location = "classroom"
print(location[-3:-1])
```

   a. `"ro"`
   b. `"oo"`
   c. `"oom"`

6. What is the output of the following code?

```
greeting = "hi Leila"
name = greeting[3:]
```

a. `" Leila"`
b. `"Leila"`
c. `"ila"`

String immutability

String objects are **immutable** meaning that string objects cannot be modified or changed once created. Once a string object is created, the string's contents cannot be altered by directly modifying individual characters or elements within the string. Instead, to make changes to a string, a new string object with the desired changes is created, leaving the original string unchanged.

CHECKPOINT

Strings are immutable

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-2-string-slicing)

CONCEPTS IN PRACTICE

Modifying string content

7. What is the correct way of replacing the first character in a string to character "*" in a new string?

a.
```
x = "string"
x[0] = "*"
```
b.
```
x = "string"
x = "*" + x[1:]
```
c.
```
x = "string"
x = "*" + x
```

8. What type of error will result from the following code?

```
string_variable = "example"
string_variable[-1] = ""
```

a. `TypeError`
b. `IndexError`
c. `NameError`

9. What is the output of the following code?

```
str = "morning"
str = str[1]
print(str)
```

a. `TypeError`
b. `m`
c. `o`

TRY IT

Changing the greeting message

Given the string `"Hello my fellow classmates"` containing a greeting message, print the first word by getting the beginning of the string up to (and including) the 5th character. Change the first word in the string to `"Hi"` instead of `"hello"` and print the greeting message again.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-2-string-slicing)

TRY IT

Editing the string at specified locations

Given a string variable, `string_variable`, and a list of indexes, remove characters at the specified indexes and print the resulting string.

```
Input:
string_variable = "great"
indices = [0, 1]



prints eat
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-2-string-slicing)

8.3 Searching/testing strings

Learning objectives

By the end of this section you should be able to

- Use the `in` operator to identify whether a given string contains a substring.
- Call the `count()` method to count the number of substrings in a given string.
- Search a string to find a substring using the `find()` method.
- Use the `index()` method to find the index of the first occurrence of a substring in a given string.
- Write a for loop on strings using in operator.

in operator

The **in** Boolean operator can be used to check if a string contains another string. `in` returns `True` if the first string exists in the second string, `False` otherwise.

CHECKPOINT

What is in the phrase?

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-3-searchingtesting-strings)

CONCEPTS IN PRACTICE

Using in operator to find substrings

1. What is the output of (`"a" in "an umbrella"`)?
   a. `2`
   b. `False`
   c. `True`
   d. `1`

2. What is the output of (`"ab" in "arbitrary"`)?
   a. `True`
   b. `False`

3. What is the output of (`"" in "string"`)?
   a. `True`
   b. `False`

For loop using in operator

The `in` operator can be used to iterate over characters in a string using a `for` loop. In each `for` loop iteration, one character is read and will be the loop variable for that iteration.

CHECKPOINT

for loop using in operator

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-3-searchingtesting-strings)

CONCEPTS IN PRACTICE

Using in operator within for loop

4. What is the output of the following code?

```
for c in "string":
  print(c, end = "")
```

a. `string`
b. `s t r i n g`
c. `s t r i n g`

5. What is the output of the following code?

```
count = 0
for c in "abca":
  if c == "a":
    count += 1
print(count)
```

a. `0`
b. `1`
c. `2`

6. What is the output of the following code?

```
word = "cab"
for i in word:
  if i == "a":
    print("A", end = "")
  if i == "b":
    print("B", end = "")
  if i == "c":
    print("C", end = "")
```

a. `cab`
b. `abc`
c. `CAB`
d. `ABC`

count()

The **count()** method counts the number of occurrences of a substring in a given string. If the given substring does not exist in the given string, the value `0` is returned.

CHECKPOINT

Counting the number of occurrences of a substring

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-3-searchingtesting-strings)

CONCEPTS IN PRACTICE

Using count() to count the number of substrings

7. What is the output of (`aaa".count("a")`)?
   a. `True`
   b. `1`
   c. `3`

8. What is the output of (`"weather".count("b")`)?
   a. `0`
   b. `-1`
   c. `False`

9. What is the output of (`"aaa".count("aa")`)?
   a. `1`
   b. `2`
   c. `3`

find()

The **find()** method returns the index of the first occurrence of a substring in a given string. If the substring does not exist in the given string, the value of `-1` is returned.

CHECKPOINT

Finding the first index of a substring

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-3-searchingtesting-strings)

CONCEPTS IN PRACTICE

Using find() to locate a substring

10. What is the output of `"banana".find("a")`?
    a. `1`
    b. `3`
    c. `5`

11. What is the output of `"banana".find("c")`?

a. `0`
b. `-1`
c. `ValueError`

12. What is the output of `"b".find("banana")`?
a. `-1`
b. `0`
c. `ValueError`

index()

The **index()** method performs similarly to the `find()` method in which the method returns the index of the first occurrence of a substring in a given string. The `index()` method assumes that the substring exists in the given string; otherwise, throws a `ValueError`.

EXAMPLE 8.4

Getting the time's minute portion

Consider a time value is given as part of a string using the format of `"hh:mm"` with `"hh"` representing the hour and `"mm"` representing the minutes. To retrieve only the string's minute portion, the following code can be used:

```
time_string = "The time is 12:50"
index = time_string.index(":")
print(time_string[index+1:index+3])
```

The above code's output is:

```
50
```

CONCEPTS IN PRACTICE

Using index() to locate a substring

13. What is the output of `"school".index("o")`?
a. `3`
b. `4`
c. `-3`

14. What is the output of `"school".index("ooo")`?
a. `3`
b. `4`

c. `ValueError`

15. What is the output of the following code?

```
sentence = "This is a sentence"
index = sentence.index(" ")
print(sentence[:index])
```

a. `"This"`
b. `"This "`
c. `"sentence"`

TRY IT

Finding all spaces

Write a program that, given a string, counts the number of space characters in the string. Also, print the given string with all spaces removed.

```
Input: "This is great"



prints:
2
Thisisgreat
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-3-searchingtesting-strings)

8.4 String formatting

Learning objectives

By the end of this section you should be able to

- Format a string template using input arguments.
- Use `format()` to generate numerical formats based on a given template.

String format specification

Python provides string substitutions syntax for formatting strings with input arguments. **Formatting string** includes specifying string pattern rules and modifying the string according to the formatting specification. Examples of formatting strings include using patterns for building different string values and specifying modification rules for the string's length and alignment.

String formatting with replacement fields

Replacement fields are used to define a pattern for creating multiple string values that comply with a given

format. The example below shows two string values that use the same template for making requests to different individuals for taking different courses.

EXAMPLE 8.5

String values from the same template

```
Dear John, I'd like to take a programming course with Prof. Potter.

Dear Kishwar, I'd like to take a math course with Prof. Robinson.
```

In the example above, replacement fields are 1) the name of the individual the request is being made to, 2) title of the course, and 3) the name of the instructor. To create a template, replacement fields can be added with `{}` to show a placeholder for user input. The **format()** method is used to pass inputs for replacement fields in a string template.

EXAMPLE 8.6

String template formatting for course enrollment requests

A string template with replacement fields is defined below to create string values with different input arguments. The `format()` method is used to pass inputs to the template in the same order.

```
s = "Dear {}, I'd like to take a {} course with Prof. {}."

print(s)
print(s.format("John", "programming", "Potter"))
print(s.format("Kishwar", "math", "Robinson"))
```

The above code's output is:

```
Dear {}, I'd like to take a {} course with Prof. {}.
Dear John, I'd like to take a programming course with Prof. Potter.
Dear Kishwar, I'd like to take a math course with Prof. Robinson.
```

CONCEPTS IN PRACTICE

String template and formatting

1. What is the output of `print("Hello {}!".format("Ana"))`?
   a. `Ana`

b. `Hello Ana`
c. `Hello Ana!`

2. What is the output of `print("{}:{}".format("One", "1"))`?
   a. `One1`
   b. `One:1`
   c. `1:One`

3. What is the output of `print("{}".format("one", "two", "three"))`?
   a. `one`
   b. `two`
   c. `onetwothree`

Named replacement fields

Replacement fields can be tagged with a label, called **named replacement fields**, for ease of access and code readability. The example below illustrates how named replacement fields can be used in string templates.

EXAMPLE 8.7

Season weather template using named replacement fields

A named replacement argument is a convenient way of assigning name tags to replacement fields and passing values associated with replacement fields using corresponding names (instead of passing values in order).

```
s = "Weather in {season} is {temperature}."

print(s)
print(s.format(season = "summer", temperature = "hot"))
print(s.format(season = "winter", temperature = "cold"))
```

The above code's output is:

```
Weather in {season} is {temperature}.
Weather in summer is hot.
Weather in winter is cold.
```

MULTIPLE USE OF A NAMED ARGUMENT

Since named replacement fields are referred to using a name key, a named replacement field can appear and be used more than once in the template. Also, positional ordering is not necessary when named replacement fields are used.

```
s = "Weather in {season} is {temperature}; very very {temperature}."

print(s)
print(s.format(season = "summer", temperature = "hot"))
print(s.format(temperature = "cold", season = "winter"))
```

The above code's output is:

```
Weather in {season} is {temperature}; very very {temperature}.
Weather in summer is hot; very very hot.
Weather in winter is cold; very very cold.
```

CONCEPTS IN PRACTICE

Named replacement field examples

4. What is the output of `print("Hey {name}!".format(name = "Bengio"))`?
   a. `Hey name!`
   b. `Hey Bengio`
   c. `Hey Bengio!`

5. What is the output of `print("Hey {name}!".format("Bengio"))`?
   a. `Hey name!`
   b. `KeyError`
   c. `Hey Bengio!`

6. What is the output of the following code?

```
greeting = "Hi"
name = "Jess"
print("{greeting} {name}".format(greeting = greeting, name = name))
```

   a. `greeting name`
   b. `Hi Jess`
   c. `Jess Hi`

Numbered replacement fields

Python's string `format()` method can use positional ordering to match the numbered arguments. The replacement fields that use the positional ordering of arguments are called **numbered replacement fields**. The indexing of the arguments starts from 0. Ex: `print("{1}{0}".format("Home", "Welcome"))` outputs the string value `"Welcome Home"` as the first argument. `"Home"` is at index 0, and the second argument, `"Welcome"`, is at index 1. Replacing these arguments in the order of `"{1}{0}"` creates the string `"Welcome`

Home".

Numbered replacement fields can use argument's values for multiple replacement fields by using the same argument index. The example below illustrates how an argument is used for more than one numbered replacement field.

EXAMPLE 8.8

Numbered replacement field to build a phrase

Numbered replacement fields are used in this example to build phrases like "very very cold" or "very hot".

```
template1 = "{0} {0} {1}"
template2 = "{0} {1}"

print(template1.format("very", "cold"))
print(template2.format("very", "hot"))
```

The above code's output is:

```
very very cold
very hot
```

String length and alignment formatting

Formatting the string length may be needed for standardizing the output style when multiple string values of the same context are being created and printed. The example below shows a use case of string formatting in printing a table with minimum-length columns and specific alignment.

EXAMPLE 8.9

A formatted table of a class roster

A formatted table of a class roster

```
Student Name              Major          Grade
-----------------------------------------------
Manoj Sara          Computer Science        A-
Gabriel Wang     Electrical Engineering      A
Alex Narayanan       Social Sciences        A+
```

In the example above, the table is formatted into three columns. The first column takes up 15 characters and is left-aligned. The second column uses 25 characters and is center-aligned, and the last column uses two characters and is right aligned. Alignment and length format specifications controls are used to create the

formatted table.

The field width in string format specification is used to specify the minimum length of the given string. If the string is shorter than the given minimum length, the string will be padded by space characters. A **field width** is included in the format specification field using an integer after a colon. Ex: `{name:15}` specifies that the minimum length of the string values that are passed to the `name` field is 15.

Since the field width can be used to specify the minimum length of a string, the string can be padded with space characters from right, left, or both to be left-aligned, right-aligned, and centered, respectively. The **string alignment type** is specified using <, >, or ^characters after the colon when field length is specified. Ex: `{name:^20}` specifies a named replacement field with the minimum length of 20 characters that is center-aligned.

Alignment Type	Symbol	Example	Output
Left-aligned	<	`template = "{hex:<7}{name:<10}"` `print(template.format(hex = "#FF0000", name = "Red")) print(template.format(hex = "#00FF00", name = "green"))`	`#FF0000Red` `#00FF00green`
Right-aligned	>	`template = "{hex:>7}{name:>10}"` `print(template.format(hex = "#FF0000", name = "Red")) print(template.format(hex = "#00FF00", name = "green"))`	`#FF0000 Red` `#00FF00 green`
Centered	^	`template = "{hex:^7}{name:^10}"` `print(template.format(hex = "#FF0000", name = "Red")) print(template.format(hex = "#00FF00", name = "green"))`	`#FF0000 Red` `#00FF00 green`

Table 8.2 String alignment formatting.

CONCEPTS IN PRACTICE

Specifying field width and alignment

7. What is the output of the following code?

```
template = "{name:12}"
formatted_name = template.format(name = "Alice")
print(len(formatted_name))
```

a. `5`
b. `12`
c. `"Alice"`

8. What is the output of the following code?

```
template = "{greeting:>6}"
formatted_greeting = template.format(greeting = "Hello")
print(formatted_greeting[0])
```

a. H
b. " Hello"
c. Space character

9. What is the output of the following code?

```
template = "{:5}"
print(template.format("123456789"))
```

a. 56789
b. 123456
c. 123456789

Formatting numbers

The `format()` method can be used to format numerical values. Numerical values can be padded to have a given minimum length, precision, and sign character. The syntax for modifying numeric values follows the `{[index]:[width][.precision][type]}` structure. In the given syntax,

- The `index` field refers to the index of the argument.
- The `width` field refers to the minimum length of the string.
- The `precision` field refers to the floating-point precision of the given number.
- The `type` field shows the type of the input that is passed to the `format()` method. Floating-point and decimal inputs are identified by `"f"` and `"d"`, respectively. String values are also identified by `"s"`.

The table below summarizes formatting options for modifying numeric values.

Example	Output	Explanation
`print("{:.7f}".format(0.9795))`	`0.9795000`	The format specification `.7` shows the output must have seven decimal places. The `f` specification is an identifier of floating-point formatting.
`print("{:.3f}".format(12))`	`12.000`	The format specification `.3` shows the output must have three decimal places. The `f` specification is an identifier of floating-point formatting.

Table 8.3 Numerical formatting options.

Example	Output	Explanation
`print("{:+.2f}".format(4))`	`+4.00`	The format specification `.2` shows the output must have two decimal places. The `f` specification is an identifier of floating-point formatting. The + sign before the precision specification adds a sign character to the output.
`print("{:0>5d}".format(5))`	`00005`	The format specification `0>5` defines the width field as 5, and thus the output must have a minimum length of 5. And, if the number has fewer than five digits, the number must be padded with 0's from the left side. The d specification is an identifier of a decimal number formatting.
`print("{:.3s}".format("12.50"))`	`12.`	The format specification `.3` shows the output will have three characters. The s specification is an identifier of string formatting.

Table 8.3 Numerical formatting options.

CONCEPTS IN PRACTICE

Numeric value formatting examples

10. What is the output of `print('{0:.3f}'.format(3.141592))`?

a. `3.141592`
b. `3.1`
c. `3.142`

11. What is the output of `print('{:1>3d}'.format(3))`?

a. `113`
b. `311`
c. `3.000`

12. What is the output of `print('{:+d}'.format(123))`?

a. `123`
b. `+123`
c. `:+123`

TRY IT

Formatting a list of numbers

Given a list of numbers (floating-point or integer), print numbers with two decimal place precision and at least six characters.

```
Input: [12.5, 2]


Prints: 012.50

002:00
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-4-string-formatting)

8.5 Splitting/joining strings

Learning objectives

By the end of this section you should be able to

- Use the `split()` method to split a string into substrings.
- Combine objects in a list into a string using `join()` method.

split()

A string in Python can be broken into substrings given a **delimiter**. A delimiter is also referred to as a separator. The **split()** method, when applied to a string, splits the string into substrings by using the given argument as a delimiter. Ex: `"1-2".split('-')` returns a list of substrings `["1", "2"]`. When no arguments are given to the `split()` method, blank space characters are used as delimiters. Ex: `"1\t2\n3 4".split()` returns `["1", "2", "3", "4"]`.

CHECKPOINT

split() for breaking down the string into tokens

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-5-splittingjoining-strings)

CONCEPTS IN PRACTICE

Examples of string delimiters and split() method

1. What is the output of `print("1*2*3*".split('*'))`?
   a. `["1", "*", "2", "*", "3", "*"]`

b. `["1", "2', "3"]`
c. `[1, 2, 3]`

2. What is the output of `print("a year includes 12 months".split())`?
   a. `["a year includes 12 months"]`
   b. `["a", "year", "includes", 12, "months"]`
   c. `["a", "year", "includes", "12", "months"]`

3. What is the output of the following code?

```
s = """This is a test"""

out = s.split()
print(out)
```

   a. Error
   b. `['This', 'is', 'a', 'test']`
   c. `>['This', 'is a', 'test']`

join()

The **join()** method is the inverse of the `split()` method: a list of string values are concatenated together to form one output string. When joining string elements in the list, the delimiter is added in-between elements. Ex: `','.join(["this", "is", "great"])` returns `"this,is,great"`.

CHECKPOINT

join() for combining tokens into one string

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-5-splittingjoining-strings)

CONCEPTS IN PRACTICE

Applying join() method on list of string values

4. What is the output of the following code?

```
elements = ['A', 'beautiful', 'day', 'for', 'learning']

print(",".join(elements))
```

   a. `'A beautiful day for learning'`
   b. `['A, beautiful, day, for, learning']`
   c. `'A,beautiful,day,for,learning'`

5. What is the length of the string `"sss".join(["1","2"])`?
   a. 2

b. 5
c. 8

6. What is the value stored in the variable `out`?

```
s = ["1", "2"]
out = "".join(s)
```

a. `12`
b. `"12"`
c. `"1 2"`

TRY IT

Unique and comma-separated words

Write a program that accepts a comma-separated sequence of words as input, and prints words in separate lines. Ex: Given the string `"happy,smiling,face"`, the output would be:

```
happy
smiling
face
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/8-5-splittingjoining-strings)

TRY IT

Lunch order

Use the `join()` method to repeat back a user's order at a restaurant, separated by commas. The user will input each food item on a separate line. When finished ordering, the user will enter a blank line. The output depends on how many items the user orders:

- If the user inputs nothing, the program outputs:
  `You ordered nothing.`
- If the user inputs one item (Ex: eggs), the program outputs:
  `You ordered eggs.`
- If the user inputs two items (Ex: eggs, ham), the program outputs:
  `You ordered eggs and ham.`
- If the user inputs three or more items (Ex: eggs, ham, toast), the program outputs:
  `You ordered eggs, ham, and toast.`

In the general case with three or more items, each item should be separated by a comma and a space. The word "and" should be added before the last item.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/

8-5-splittingjoining-strings)

8.6 Chapter summary

Highlights from this chapter include:

- A string is a sequence of characters.
- Logical operators can be used to compare two string values. String comparison is done by comparing corresponding ASCII values of characters in the order of appearance in the string.
- String indexing is used to access a character or a sequence of characters in the string.
- String objects are immutable.
- String splicing.

At this point, you should be able to write programs dealing with string values.

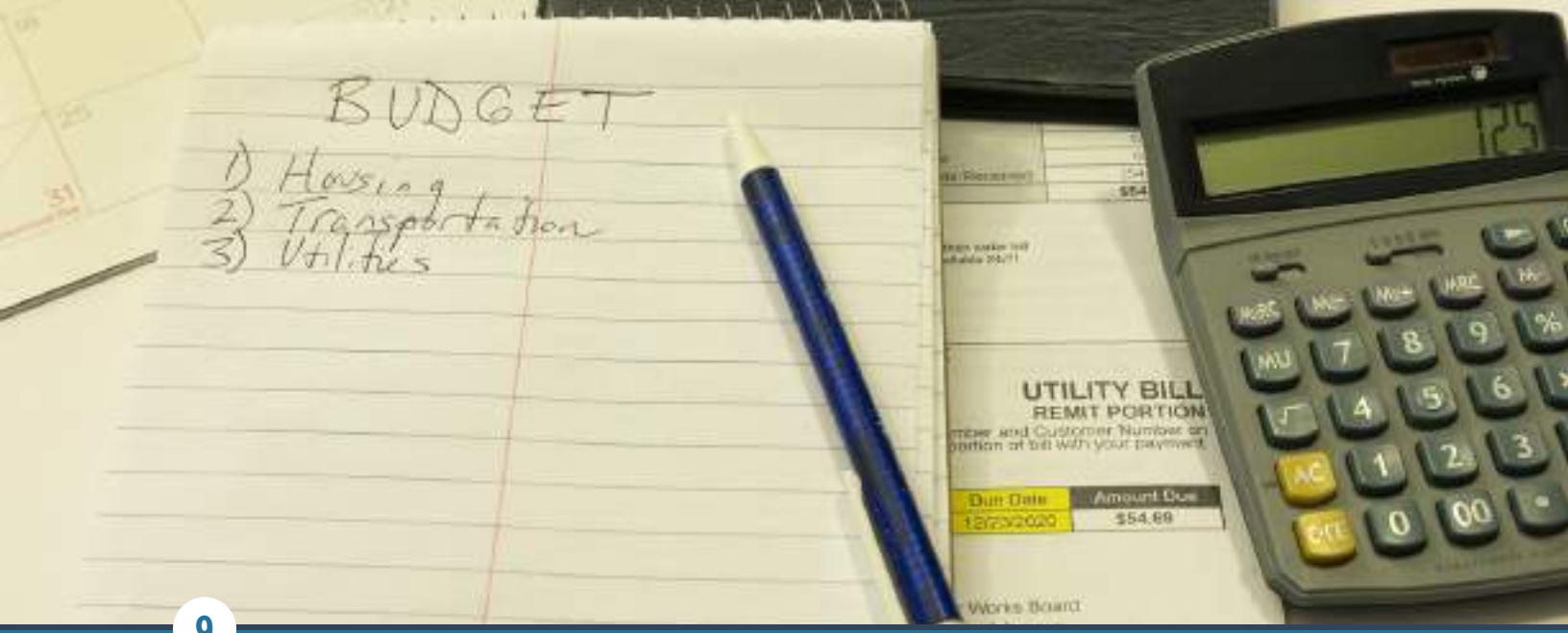
Method	Description
`len()`	Returns the string length.
`upper()`	Returns uppercase characters.
`lower()`	Returns lowercase characters.
`count()`	Returns the number of a given substring in a string.
`find()`	Returns the index of the first occurrence of a given substring in a string. If the substring does not exist in the string, -1 is returned.
`index()`	Returns the index of the first occurrence of a given substring in a string. If the substring does not exist in the string, a `ValueError` is returned.
`format()`	Used to create strings with specified patterns using arguments.
`join()`	Takes a list of string values and combines string values into one string by placing a given separator between values.
`split()`	Separates a string into tokens based on a given separator string. If no separator string is provided, blank space characters are used as separators.
Operator	**Description**

Table 8.4 Chapter 8 reference.

Method	Description
`in`	Checks if a substring exists in a string.

`in` operator in a `for` loop	
	`for character in string:`
	`  # loop body`

Table 8.4 Chapter 8 reference.



9

Lists

Figure 9.1 credit: modification of work "Budget and Bills" by Alabama Extension/Flickr, Public Domain

Chapter Outline

Introduction

Programmers often work on collections of data. Lists are a useful way of collecting data elements. Python lists are extremely flexible, and, unlike strings, a list's contents can be changed.

The Objects chapter introduced lists. This chapter explores operations that can be performed on lists.

9.1 Modifying and iterating lists

Learning objectives

By the end of this section you should be able to

- Modify a list using `append()`, `remove()`, and `pop()` list operations.
- Search a list using a `for` loop.

Using list operations to modify a list

An `append()` operation is used to add an element to the end of a list. In programming, **append** means add to the end. A `remove()` operation removes the specified element from a list. A `pop()` operation removes the last item of a list.

EXAMPLE 9.1

Simple operations to modify a list

The code below demonstrates simple operations for modifying a list.

Line 8 shows the `append()` operation, line 12 shows the `remove()` operation, and line 17 shows the `pop()` operation. Since the `pop()` operation removes the last element, no parameter is needed.

```
1  """Operations for adding and removing elements from a list."""
2
3      # Create a list of students working on a project
4      student_list = ["Jamie", "Vicky", "DeShawn", "Tae"]
5      print(student_list)
6
7      # Another student joins the project. The student must be added
   to the list.
8      student_list.append("Ming")
9      print(student_list)
10
11     # "Jamie" withdraws from the project. Jamie must be removed
   from the list.
12     student_list.remove("Jamie")
13     print(student_list)
14
15     # Suppose "Ming" had to be removed from the list.
16     # A pop() operation can be used since Ming is last in the
   list.
17     student_list.pop()
18     print(student_list)
```

The above code's output is:

```
['Jamie', 'Vicky', 'DeShawn', 'Tae']
['Jamie', 'Vicky', 'DeShawn', 'Tae', 'Ming']
['Vicky', 'DeShawn', 'Tae', 'Ming']
['Vicky', 'DeShawn', 'Tae']
```

CONCEPTS IN PRACTICE

Modifying lists

1. Which operation can be used to add an element to the end of a list?
   a. `add()`
   b. `append()`
   c. `pop()`

2. What is the correct syntax to remove the element 23 from a list called `number_list`?
   a. `remove()`
   b. `number_list.remove()`
   c. `number_list.remove(23)`

3. Which operation can be used to remove an element from the end of a list?
   a. only `pop()`
   b. only `remove()`
   c. either `pop()` or `remove()`

Iterating lists

An iterative for loop can be used to iterate through a list. Alternatively, lists can be iterated using list indexes with a counting for loop. The animation below shows both ways of iterating a list.

CHECKPOINT

Using len() to get the length of a list

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-1-modifying-and-iterating-lists)

CONCEPTS IN PRACTICE

Iterating lists

For the following questions, consider the list:

```
my_list = [2, 3, 5, 7, 9]
```

4. How many times will the following `for` loop execute?
   ```
   for element in my_list:
   ```
   a. 5
   b. 4

5. What is the final value of `i` for the following counting `for` loop?
   ```
   for i in range(0, len(my_list)):
   ```
   a. 9
   b. 4
   c. 5

6. What is the output of the code below?

   ```
   for i in range(0, len(my_list), 2):
     print(my_list[i], end=' ')
   ```

   a. 2 5 9
   b. 2 3 5 7 9

```
c. 2
   5
   9
```

TRY IT

Sports list

Create a list of sports played on a college campus. The sports to be included are baseball, football, tennis, and table tennis.

Next, add volleyball to the list.

Next, remove `"football"` from the list and add `"soccer"` to the list.

Show the list contents after each modification.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-1-modifying-and-iterating-lists)

TRY IT

Simple Searching

Write a program that prints `"found!"` if `"soccer"` is found in the given list.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-1-modifying-and-iterating-lists)

9.2 Sorting and reversing lists

Learning objectives

By the end of this section you should be able to

- Understand the concept of sorting.
- Use built-in `sort()` and `reverse()` methods.

Sorting

Ordering elements in a sequence is often useful. **Sorting** is the task of arranging elements in a sequence in ascending or descending order.

Sorting can work on numerical or non-numerical data. When ordering text, dictionary order is used. Ex: "bat" comes before "cat" because "b" comes before "c".

CHECKPOINT

Sorting

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/

9-2-sorting-and-reversing-lists)

CONCEPTS IN PRACTICE

Sorting

1. What would be the last element of the following list if it is sorted in descending order?
   `[12, 3, 19, 25, 16, -3, 5]`
   a. `25`
   b. `-3`
   c. `5`

2. Arrange the following list in ascending order.
   `["cat", "bat", "dog", "coyote", "wolf"]`
   a. `["bat", "cat", "coyote", "dog", "wolf"]`
   b. `["wolf", "coyote", "dog", "cat", "bat"]`

3. How are the words `"flask"` and `"flash"` related in Python?
   a. `"flask" < "flash"`
   b. `"flask" == "flash"`
   c. `"flask" > "flash"`

Using `sort()` and `reverse()`

Python provides methods for arranging elements in a list.

- The `sort()` method arranges the elements of a list in ascending order. For strings, ASCII values are used and uppercase characters come before lowercase characters, leading to unexpected results. Ex: `"A"` is ordered before `"a"` in ascending order but so is `"G"`; thus, `"Gail"` comes before `"apple"`.
- The `reverse()` method reverses the elements in a list.

EXAMPLE 9.2

Sorting and reversing lists

```
    # Setup a list of numbers
    num_list = [38, 92, 23, 16]
    print(num_list)

    # Sort the list
    num_list.sort()
    print(num_list)

    # Setup a list of words
    dance_list = ["Stepping", "Ballet", "Salsa", "Kathak", "Hopak", "Flamenco",
"Dabke"]
```

```
# Reverse the list
dance_list.reverse()
print(dance_list)

# Sort the list
dance_list.sort()
print(dance_list)
```

The above code's output is:

```
[38, 92, 23, 16]
[16, 23, 38, 92]
["Dabke", "Flamenco", "Hopak", "Kathak", "Salsa", "Ballet", "Stepping"]
["Ballet", "Dabke", "Flamenco", "Hopak", "Kathak", "Salsa", "Stepping"]
```

CONCEPTS IN PRACTICE

sort() and reverse() methods

Use the following list for the questions below.

```
board_games = ["go", "chess", "scrabble", "checkers"]
```

4. What is the correct way to sort the list `board_games` in ascending order?
   a. `sort(board_games)`
   b. `board_games.sort()`
   c. `board_games.sort('ascending')`

5. What is the correct way to reverse the list `board_games`?
   a. `board_games.reverse()`
   b. `reverse(board_games)`

6. What would be the last element of `board_games` after the `reverse()` method has been applied?
   a. `'go'`
   b. `'checkers'`
   c. `'scrabble'`

TRY IT

Sorting and reversing

Complete the program below to arrange and print the numbers in ascending and descending order.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-2-sorting-and-reversing-lists)

9.3 Common list operations

Learning objectives

By the end of this section you should be able to

- Use built-in functions `max()`, `min()`, and `sum()`.
- Demonstrate how to copy a list.

Using built-in operations

The `max()` function called on a list returns the largest element in the list. The `min()` function called on a list returns the smallest element in the list. The `max()` and `min()` functions work for lists as long as elements within the list are comparable.

The `sum()` function called on a list of numbers returns the sum of all elements in the list.

EXAMPLE 9.3

Common list operations

```
    """Common list operations."""

    # Set up a list of number
    snum_list = [28, 92, 17, 3, -5, 999, 1]

    # Set up a list of words
    city_list = ["New York", "Missoula", "Chicago", "Bozeman",
    "Birmingham", "Austin", "Sacramento"]

    # Usage of the max() funtion
    print(max(num_list))

    # max() function works for strings as well
    print(max(city_list))

    # Usage of the min() funtion which also works for strings
    print(min(num_list))

    print(min(city_list))

    # sum() only works for a list of numbers
    print(sum(num_list))
```

The above code's output is:

```
999
Sacramento
-5
Austin
1135
```

CONCEPTS IN PRACTICE

List operations

1. What is the correct way to get the minimum of a list named `nums_list`?
   a. `min(nums_list)`
   b. `nums_list.min()`
   c. `minimum(nums_list)`

2. What is the minimum of the following list?
   `["Lollapalooza", "Coachella", "Newport Jazz festival", "Hardly Strictly Bluegrass", "Austin City Limits"]`
   a. `Coachella`
   b. `Austin City Limits`
   c. The minimum doesn't exist.

3. What value does the function call return?
   `sum([1.2, 2.1, 3.2, 5.9])`
   a. `sum()` only works for integers.
   b. `11`
   c. `12.4`

Copying a list

The `copy()` method is used to create a copy of a list.

CHECKPOINT

Copying a list

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-3-common-list-operations)

CONCEPTS IN PRACTICE

Copying a list

4. What is the output of the following code?

```
my_list = [1, 2, 3]
list2 = my_list
list2[0] = 13
print(sum(my_list))
```

a. 6
b. 13
c. 18

5. What is the output of the following code?

```
my_list = [1, 2, 3]
list2 = my_list.copy()
list2[0] = 13
print(max(my_list))
```

a. 3
b. 13
c. 18

6. What is the output of the following code?

```
my_list = ["Cat", "Dog", "Hamster"]
list2 = my_list
list2[2] = "Pigeon"
print(sum(my_list))
```

a. CatDogPigeon
b. Error

TRY IT

Copy

Make a copy of `word_list` called wisdom. Sort the list called wisdom. Create a sentence using the words in each list and print those sentences (no need to add periods at the end of the sentences).

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-3-common-list-operations)

9.4 Nested lists

Learning objectives

By the end of this section you should be able to

- Demonstrate the use of a list-of-lists to structure data.
- Demonstrate individual element addressing using multi-dimensional indexing.
- Use nested loops to iterate a list-of-lists.

List-of-lists

Lists can be made of any type of element. A list element can also be a list. Ex: `[2, [3, 5], 17]` is a valid list with the list `[3, 5]` being the element at index 1.

When a list is an element inside a larger list, it is called a **nested list**. Nested lists are useful for expressing multidimensional data. When each of the elements of a larger list is a smaller list, the larger list is called a **list-of-lists**.

Ex: A table can be stored as a two-dimensional list-of-lists, where each row of data is a list in the list-of-lists.

CHECKPOINT

List-of-lists

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-4-nested-lists)

CONCEPTS IN PRACTICE

Lists

For each of the questions below, consider the following matrix:

$$\text{mat A} = \begin{bmatrix} 7 & 4 & 5 \\ 3 & 9 & 6 \\ 1 & 2 & 8 \end{bmatrix}$$

1. What would be the correct way to represent `matA` in Python?
   a. `[[7, 4, 5], [3, 9, 6], [1, 2, 8]]`
   b.
   ```
   [7, 4, 5
    3, 9, 6
    1, 2, 8]
   ```
   c. `[[7, 3, 1], [4, 9, 2], [1, 2, 8]`

2. What would be the correct index for the number `6` in the above list?
   a. `[5]`
   b. `[2][1]`
   c. `[1][2]`

3. What would be the result of the following code:

   ```
   print(matA[0])
   ```

   a. `Error`
   b. `7`
   c. `[7, 4, 5]`

Using nested loops to iterate nested lists

A nested loop structure can be used to iterate a list-of-lists. For a two-dimensional list-of-lists, an outer `for` loop can be used for rows, and an inner `for` loop can be used for columns.

EXAMPLE 9.4

Iterating a list-of-lists

The code below demonstrates how to iterate a list-of-lists.

The outer loop on line 9 goes element by element for the larger list. Each element in the larger list is a list. The inner loop on line 10 iterates through each element in each nested list.

```
1  """Iterating a list-of-lists."""
2
3      # Create a list of numbers
4      list1 = [[1, 2, 3],
5               [1, 4, 9],
6               [1, 8, 27]]
7
8      # Iterating the list-of-lists
9      for row in list1:
10       for num in row:
11         print(num, end=" ")
12       print()
```

The above code's output is:

```
1 2 3
1 4 9
1 8 27
```

CONCEPTS IN PRACTICE

Iterating a list-of-lists

For each question below, consider the following list:

```
my_list = [[7, 4, 5, 12],
           [24, 3, 9, 16],
           [12, 8, 91, -5]]
```

4. Which code prints each number in `my_list` starting from `7`, then `4`, and so on ending with `-5`?

a.
```
for row in my_list:
    for elem in row:
      print(elem)
```

b.
```
for elem in my_list:
      print(elem)
```

5. The `range()` function can also be used to iterate a list-of-lists. Which code prints each number in `my_list` starting from `7`, then `4`, and so on, ending with `-5`, using counting `for` loops?

```
a. for column_index in range(0, len(my_list[0])):
     for row_index in range (0, len(my_list)):
       print(my_list[row_index][column_index])
b. for row_index in range(0, len(my_list)):
     for column_index in range (0, len(my_list)):
       print(my_list[row_index][column_index])
     print()
c. for row_index in range(0, len(my_list)):
     for column_index in range (0, len(my_list[0])):
       print(my_list[row_index][column_index])
```

TRY IT

Matrix multiplication

Write a program that calculates the matrix multiplication product of the matrices matW and matZ below and prints the result. The expected result is shown.

$$\text{mat W} = \begin{bmatrix} 13 & 4 & 5 \\ 2 & -9 & 7 \\ 7 & 3 & 19 \end{bmatrix} \text{mat Z} = \begin{bmatrix} 2 & 1 & 5 \\ 3 & 7 & 9 \\ -1 & 13 & 19 \end{bmatrix}$$

$$\text{result} = \begin{bmatrix} 33 & 106 & 196 \\ -30 & 30 & 62 \\ 4 & 275 & 423 \end{bmatrix}$$

In the result matrix, each element is calculated according to the position of the element. The result at position `[i][j]` is calculated using row `i` from the first matrix, W, and column `j` from the second matrix, Z.

Ex:

`result[1][2]` = (row 1 in W) times (column 2 in Z)

$$\text{result[1][2]} = \begin{bmatrix} 2 & -9 & 7 \end{bmatrix} * \begin{bmatrix} 5 \\ 9 \\ 19 \end{bmatrix}$$

$$\text{result[1][2]} = 2 * 5 + (-9) * 9 + 7 * 19 = 10 - 81 + 133 = 62$$

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-4-nested-lists)

9.5 List comprehensions

Learning objectives

By the end of this section you should be able to

- Identify the different components of a list comprehension statement.
- Implement filtering using list comprehension.

List comprehensions

A **list comprehension** is a Python statement to compactly create a new list using a pattern.

The general form of a list comprehension statement is shown below.

`list_name = [expression for loop_variable in iterable]`

`list_name` refers to the name of a new list, which can be anything, and the `for` is the `for` loop keyword. An expression defines what will become part of the new list. `loop_variable` is an iterator, and `iterable` is an object that can be iterated, such as a list or string.

EXAMPLE 9.5

Creating a new list with a list comprehension

A list comprehension shown below in the second code has the same effect as the regular `for` loop shown in the first code. The resultant list is `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]` in both cases.

Creating a list of squares using a `for` loop.

```
# Create an empty List.
squares_list = []

# Add items to a list, as squares of numbers starting at 0 and ending at 9.
for i in range(10):
  squares_list.append(i*i)
```

Creating a list of squares using the list comprehension.

```
square_list = [i*i for i in range(10)]
```

The expression `i*i` is applied for each value of the `loop_variable` `i`.

EXAMPLE 9.6

A Dr. Seuss poem

A list comprehension can be used to create a list based on another list. In line 6, the `for` loop is written on the list `poem_lines`.

```
1 # Create a list of words
2     words_list = ["one", "two", "red", "blue"]
3
4     # Use a list comprehension to create a new list called
  poem_lines
5     # Inserting the word "fish" attached to each word in words_list
6     poem_lines = [w + " fish" for w in words_list]
```

```
    for line in poem_lines:
      print(line)
```

The above code's output is:

```
one fish
two fish
red fish
blue fish
```

CONCEPTS IN PRACTICE

List comprehensions

1. The component of a list comprehension defining an element of the new list is the _____.
   a. expression
   b. loop_variable
   c. container

2. What would be the contents of `b_list` after executing the code below?

```
a_list = [1, 2, 3, 4, 5]
b_list = [i+2 for i in a_list]
```

   a. `[1, 2, 3, 4, 5]`
   b. `[0, 1, 2, 3, 4]`
   c. `[3, 4, 5, 6, 7]`

3. What does `new_list` contain after executing the statement below?

```
new_list = [i//3 for i in range(1, 15, 3)]
```

   a. `[0.3333333333333333, 1.3333333333333333, 2.3333333333333335, 3.3333333333333335, 4.333333333333333]`
   b. `[0, 1, 2, 3, 4]`
   c. `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]`

Filtering using list comprehensions

List comprehensions can be used to filter items from a given list. A *condition* is added to the list comprehension.

```
list_name = [expression for loop_variable in container if condition]
```

In a filter list comprehension, an element is added into `list_name` only if the condition is met.

CHECKPOINT

Filtering a list

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-5-list-comprehensions)

CONCEPTS IN PRACTICE

Filtering using list comprehensions

For each code using list comprehension, select the correct resultant list in `new_list`.

4.
```
my_list = [21, -1, 50, -9, 300, -50, 2]

new_list = [m for m in my_list if m < 0]
```

   a. `[21, 50, 300, 2]`
   b. `[21, -1, 50, -9, 300, -50, 2]`
   c. `[-1, -9, -50]`

5.
```
my_string = "This is a home."

new_list = [i for i in my_string if i in 'aeiou']
```

   a. `[i, i, a, o, e]`
   b. `['i', 'i'', 'a', 'o', 'e']`
   c. `Error`

6.
```
new_list = [r for r in range (0, 21, 2) if r%2 != 0]
```

   a. `[]`
   b. `[21]`
   c. `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]`

TRY IT

Selecting five-letter words

Write a program that creates a list of only five-letter words from the given list and prints the new list.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-5-list-comprehensions)

TRY IT

Books starting with "A"

Write a program that selects words that begin with an "A" in the given list. Make sure the new list is then sorted in dictionary order. Finally, print the new sorted list.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/9-5-list-comprehensions)

9.6 Chapter summary

Highlights from this chapter include:

- Lists are mutable and can be easily modified by using `append()`, `remove()`, and `pop()` operations.
- Lists are iterable and can be iterated using an iterator or element indexes.
- The `sort()` operation arranges the elements of a list in ascending order if all elements of the list are of the same type.
- The `reverse()` operation reverses a list.
- The `copy()` method is used to create a copy of a list.
- Lists have built-in functions for finding the maximum, minimum, and summation of a list for lists with only numeric values.
- Lists can be nested to represent multidimensional data.
- A list comprehension is a compact way of creating a new list, which can be used to filter items from an existing list.

At this point, you should be able to write programs using lists.

Function	Description
`append(element)`	Adds the specified element to the end of a list.
`remove(element)`	Removes the specified element from the list if the element exists.
`pop()`	Removes the last element of a list.
`max(list)`	Returns the maximum element of the list specified.
`min(list)`	Returns the maximum element of the list specified.
`sum(list)`	Returns the summation of a list composed of numbers.
`sort()`	Sorts a list on which the method is called in ascending order.

Table 9.1 Chapter 9 reference.

Function	Description
`reverse()`	Reverses the order of elements in a list.
`copy()`	Makes a complete copy of a list.

Table 9.1 Chapter 9 reference.