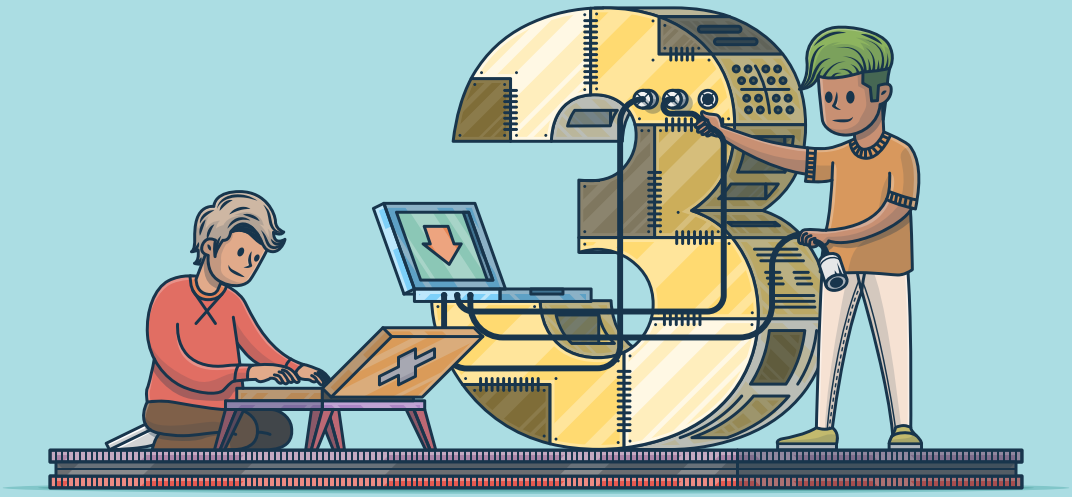
Real Python

CHOOLS

PYTHON BASICS



A PRACTICAL INTRODUCTION TO PYTHON 3

FOURTH EDITION

BY THE REALPYTHON.COM TUTORIAL TEAM
DAVID AMOS, DAN BADER, JOANNA JABLONSKI, FLETCHER HEISLER

Python Basics: A Practical Introduction to Python 3

Real Python

Contents

Foreword

Hello, and welcome to *Python Basics: A Practical Introduction to Python 3*. I hope you're ready to learn why so many professional and hobbyist developers are drawn to Python and how you can begin using it on your own projects, small and large, right away.

This book is targeted at beginners who either know a little programming but not the Python language and ecosystem or are starting fresh with no programming experience whatsoever.

If you don't have a computer science degree, don't worry. David, Dan, Joanna, and Fletcher will guide you through the important computing concepts while teaching you the Python basics and, just as importantly, skipping the unnecessary details at first.

Python Is a Full-Spectrum Language

When learning a new programming language, you don't yet have the experience to judge how well it will serve you in the long run. If you're considering learning Python, let me assure you that this is a good choice. One key reason is that Python is a **full-spectrum** language.

What do I mean by this? Some languages are very good for beginners. They hold your hand and make programming super easy. We can go to the extreme and look at visual languages such as Scratch.

In Scratch, you get blocks that represent programming concepts like variables, loops, method calls, and so on, and you drag and drop them on a visual surface. Scratch may be easy to get started with for sim-

ple programs, but you cannot build professional applications with it. Name one Fortune 500 company that powers its core business logic with Scratch.

Come up empty? Me too, because that would be insanity.

Other languages are incredibly powerful for expert developers. The most popular one in this category is likely C++ and its close relative, C. Whichever web browser you used today was likely written in C or C++. Your operating system running that browser was very likely also built with C/C++. Your favorite first-person shooter or strategy video game? You nailed it: C/C++.

You can do amazing things with these languages, but they are wholly unwelcoming to newcomers looking for a gentle introduction.

You might not have read a lot of C++ code. It can almost make your eyes burn. Here's an example, a real albeit complex one:

```
template <typename T>
_Defer<void(*(PID<T>, void (T::*)(void)))
    (const PID<T>&, void (T::*)(void))>
defer(const PID<T>& pid, void (T::*method)(void))
{
  void (*dispatch)(const PID<T>&, void (T::*)(void)) =
    &process::template dispatch<T>;
  return std::tr1::bind(dispatch, pid, method);
}
```

Please, just no.

Both Scratch and C++ are decidedly *not* what I would call full-spectrum languages. With Scratch, it's easy to start, but you have to switch to a "real" language to build real applications. Conversely, you can build real apps with C++, but there's no gentle on-ramp. You dive headfirst into all the complexity of the language, which exists to support these rich applications.

Python, on the other hand, is special. It *is* a full-spectrum language. We often judge the simplicity of a language based on the `Hello, World` test. That is, what syntax and actions are necessary to get the language to output `Hello, World` to the user? In Python, it couldn't be simpler:

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

That's it! However, I find this an unsatisfying test.

The `Hello, World` test is useful but really not enough to show the power or complexity of a language. Let's try another example. Not everything here needs to make total sense—just follow along to get the Zen of it. The book covers these concepts and more as you go through. The next example is certainly something you could write as you get near the end of the book.

Here's the new test: What would it take to write a program that accesses an external website, downloads the content to your app in memory, then displays a subsection of that content to the user? Let's try that experiment using Python 3 with the help of the `requests` package (which needs to be installed—more on that in chapter 12):

```
import requests
resp = requests.get("http://olympus.realpython.org")
html = resp.text
print(html[86:132])
```

Incredibly, that's it! When run, the program outputs something like this:

```
<h2>Please log in to access Mount Olympus:</h2>
```

This is the easy, getting-started side of the Python spectrum. A few trivial lines can unleash incredible power. Because Python has access to so many powerful but well-packaged libraries, such as `requests`, it's often described as *having batteries included*.

So there you have a simple yet powerful starter example. On the real-world side of things, many incredible applications have been written in Python as well.

YouTube, the world's most popular video streaming site, is written in Python and processes more than a million requests per second. Instagram is another example of a Python application. Closer to home, we even have realpython.com and my sites, such as talkpython.fm.

This full-spectrum aspect of Python means that you can start with the basics and adopt more advanced features as your application demands grow.

Python Is Popular

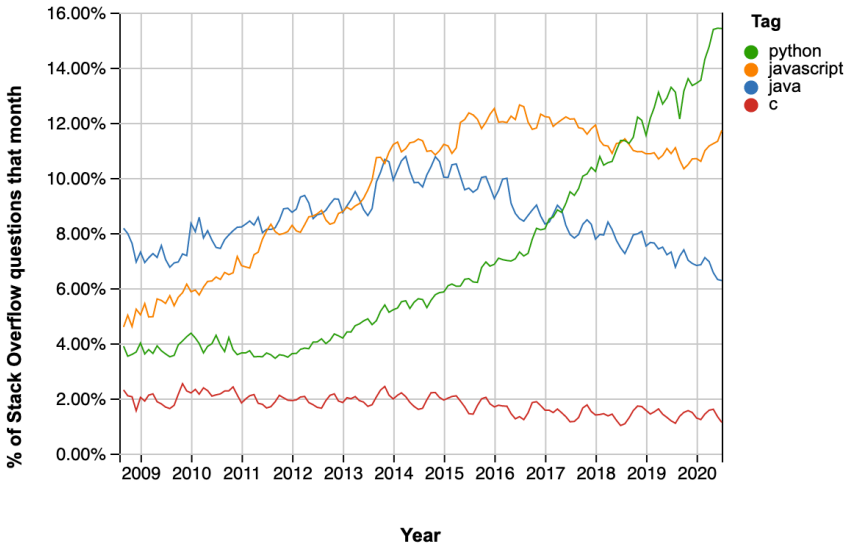
You might have heard that Python is popular. It may seem that it doesn't really matter how popular a language is so long as you can build the app you want to build with it.

But, for better or worse, the popularity of a programming language is a strong indicator of the quality of libraries you'll have available as well the number of job openings you'll find. In short, you should tend to gravitate toward more popular technologies as there will be more choices and integrations available.

So, is Python actually that popular? Yes it is. You'll find a lot of hype and hyperbole, but there are plenty of stats backing this claim. Let's look at some analytics presented by stackoverflow.com, a popular question-and-answer site for programmers.

Stack Overflow runs a site called Stack Overflow Trends where you can look at the trends for various technologies by tag. When you compare

Python to the other likely candidates you could pick to learn programming, you'll see one is unlike the others:



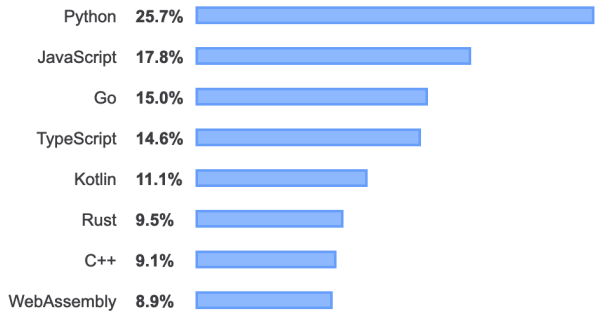
You can explore this chart and create similar charts to this one over at insights.stackoverflow.com/trends.

Notice the incredible growth of Python compared to the flat or even downward trend of the other usual candidates! If you're betting your future on the success of a given technology, which one would you choose from this list?

That's just one chart—what does it really tell us? Well, let's look at another. Stack Overflow does a yearly survey of developers. It's comprehensive and very well done. You can find the full 2020 results at insights.stackoverflow.com/survey/2020.

From that writeup, I'd like to call your attention to a section titled "Most Loved, Dreaded, and Wanted Languages." In the "Most Wanted" section, you'll find data on the share of "developers who are not developing with the language or technology but have expressed interest in developing with it."

Again, in the graph below, you'll see that Python is topping the charts and is well above even second place:



If you agree with me that the relative popularity of a programming language matters, then Python is clearly a good choice.

We Don't Need You to Be a Computer Scientist

One other point that I want to emphasize as you start your Python learning journey is that we don't need you to be a computer scientist. If that's your goal, then great. Learning Python is a powerful step in that direction. But the invitation to learn programming is often framed as "We have all these developer jobs going unfilled! We need software developers!"

That may or may not be true. But, more importantly, programming (even a little programming) can be a personal superpower for you.

To illustrate this idea, suppose you are a biologist. Should you drop out of biology and get a job as a front-end web developer? Probably not. But skills such as the one I opened this foreword with, using requests to get data from the Web, can be incredibly powerful for you as a biologist.

Rather than manually exporting and scraping data from the Web or from spreadsheets, you can use Python to scrape thousands of data sources or spreadsheets in the time it takes you to do just one man-

ually. Python skills can take your *biology power* and amplify it well beyond your colleagues' to make it your *superpower*.

Dan and Real Python

Finally, let me leave you with a comment on your authors. Dan Bader and the other *Real Python* authors work day in and day out to bring clear and powerful explanations of Python concepts to all of us via realpython.com.

They have a unique view into the Python ecosystem and are keyed into what beginners need to know.

I'm confident leaving you in their hands on this Python journey. Go forth and learn this amazing language using this great book. Most importantly, remember to have fun!

— **Michael Kennedy**, Founder of Talk Python (@mkennedy)

Chapter 1

Introduction

Welcome to *Real Python*'s *Python Basics* book, fully updated for Python 3.9! In this book, you'll learn real-world Python programming techniques, illustrated with useful and interesting examples.

Whether you're a new programmer or a professional software developer looking to dive into a new language, this book will teach you all the practical Python that you need to get started on projects of your own.

No matter what your ultimate goals may be, if you work with a computer at all, then you'll soon be finding endless ways to improve your life by automating tasks and solving problems through Python programs that you create.

But what's so great about Python as a programming language? For one, Python is open source freeware, meaning you can download it for free and use it for any purpose, commercial or not.

Python also has an amazing community that has built a number of useful tools that you can use in your own programs. Need to work with PDF documents? There's a comprehensive tool for that. Want to collect data from web pages? No need to start from scratch!

Python was built to be easier to use than other programming languages. It's usually much easier to read Python code and much faster to write code in Python than in other languages.

For instance, here's some basic code written in C, another commonly used programming language:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World\n");
}
```

All the program does is show the text `Hello, World` on the screen. That was a lot of work to output one phrase! Here's the same program written in Python:

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

That's pretty simple, right? The Python code is faster to write and easier to read. We find that it looks friendlier and more approachable, too!

At the same time, Python has all the functionality of other languages and more. You might be surprised by how many professional products are built on Python code: Instagram, YouTube, Reddit, Spotify, to name just a few.

Python is not only a friendly and fun language to learn, but it also powers the technology behind multiple world-class companies and offers fantastic career opportunities for any programmer who masters it.

1.1 Why This Book?

Let's face it: there's an overwhelming amount of information about Python on the Internet. But many beginners studying on their own have trouble figuring out *what* to learn and *in what order* to learn it.

You may be asking yourself, What should I learn about Python in the beginning to get a strong foundation? If so, then this book is for you, no matter if you're a complete beginner or if you've already dabbled in Python or other languages.

Python Basics is written in plain English and breaks down the core concepts that you really need to know into bite-sized chunks. This means you'll learn enough to be dangerous with Python, fast.

Instead of just going through a boring list of language features, you'll see exactly how the different building blocks fit together and what's involved in building real applications and scripts with Python.

Step by step, you'll master fundamental Python concepts that will help you get started on your journey toward learning Python.

Many programming books try to cover every last possible variation of every command, which makes it easy for readers to get lost in the details. This approach is great if you're looking for a reference manual, but it's a horrible way to learn a programming language. Not only do you spend most of your time cramming things into your head that you'll never use, but you also don't have any fun!

This book is built on the 80/20 principle, which suggests that you can learn most of what you need to know by focusing on a few crucial concepts. We'll cover the commands and techniques used in the vast majority of cases and focus on how to program real-world solutions to everyday problems.

This way, we guarantee that you will:

- Learn useful programming techniques quickly
- Spend less time struggling with unimportant complications
- Find more practical uses for Python in your own life
- Have more fun in the process

Once you've mastered the material in this book, you will have gained a strong enough foundation that venturing out on your own into more advanced territory will be a breeze.

What you'll learn here is based on the first part of the original *Real Python Course* initially released in 2012. Over the years, this Python curriculum has been battle-tested by thousands of Pythonistas, data scientists, and developers working for companies big and small, including Amazon, Red Hat, and Microsoft.

For *Python Basics*, we've thoroughly expanded, refined, and updated the material so you can build your Python skills quickly and efficiently.

1.2 About Real Python

At *Real Python*, you'll learn real-world programming skills from a community of professional Pythonistas from all around the world.

The realpython.com website launched in 2012 and currently helps more than three million Python developers each month with books, programming tutorials, and other in-depth learning resources.

Everyone who worked on this book is a *Python practitioner* recruited from the Real Python team with several years of professional experience in the software industry.

Here's where you can find *Real Python* on the Web:

- realpython.com
- @realpython on Twitter
- The *Real Python Newsletter*
- The *Real Python Podcast*

1.3 How to Use This Book

The first half of this book is a quick but thorough overview of all the Python fundamentals. You don't need any prior experience with programming to get started. The second half is focused on finding practical solutions to interesting, real-world coding problems.

If you're a beginner, then we recommend that you go through the first half of this book from beginning to end. The second half covers topics that don't overlap as much, so you can jump around more easily, but the chapters do increase in difficulty as you go along.

If you're a more experienced programmer, then you may find yourself heading toward the second part of the book right away. But don't neglect getting a strong foundation in the basics first, and be sure to fill in any knowledge gaps along the way.

Most sections within a chapter are followed by **review exercises** to help you make sure that you've mastered all the topics covered. There are also a number of **code challenges**, which are more involved and usually require you to tie together several different concepts from previous chapters.

The practice files that accompany this book also include full solutions to the challenges as well as some of the trickier exercises. But to get the most out of the material, you should try your best to solve the challenge problems on your own before looking at the example solutions.

If you're completely new to programming, then you may want to supplement the first few chapters with additional practice. We recommend working through the entry-level tutorials available for free at realpython.com to make sure you're on solid footing.

If you have any questions or feedback about the book, you're always welcome to contact us directly.

Learning by Doing

This book is all about learning by doing, so be sure to *actually type in* the code snippets you encounter in the book. For best results, we recommend that you avoid copying and pasting the code examples.

You'll learn the concepts better and pick up the syntax faster if you type out each line of code yourself. Plus, if you screw up—which is totally normal and happens to all developers on a daily basis—the simple act of correcting typos will help you learn how to debug your code.

Try to complete the review exercises and code challenges on your own before getting help from outside resources. With enough practice, you'll master this material—and have fun along the way!

How Long Will It Take to Finish This Book?

If you're already familiar with a programming language, then you could finish this book in as little as thirty-five to forty hours. If you're new to programming, then you may need to spend up to one hundred hours or more.

Take your time and don't feel like you have to rush. Programming is a super-rewarding but complex skill to learn. Good luck on your Python journey. We're rooting for you!

1.4 Bonus Material and Learning Resources

This book comes with a number of free bonus resources and downloads that you can access online at the link below. We're also maintaining an errata list with corrections there:

realpython.com/python-basics/resources

Interactive Quizzes

Most chapters in this book come with a free online quiz to check your learning progress. You can access the quizzes using the links provided at the end of the chapter. The quizzes are hosted on the *Real Python* website and can be viewed on your phone or computer.

Each quiz takes you through a series of questions related to a particular chapter in the book. Some of them are multiple choice, some will ask you to type in an answer, and some will require you to write actual Python code. As you make your way through each quiz, it will keep score of which questions you answered correctly.

At the end of the quiz, you'll receive a grade based on your result. If you don't score 100 percent on your first try, don't fret! These quizzes are meant to challenge you. It's expected that you'll go through them several times, improving your score with each run.

Exercises Code Repository

This book has an accompanying code repository on the Web containing example source code as well as the answers to exercises and code challenges. The repository is broken up by chapter, so you can check your code against the solutions provided by us after you finish each chapter. Here's the link:

realpython.com/python-basics/exercises

Note

The code found in this book has been tested with Python 3.9 on Windows, macOS, and Linux.

Example Code License

The example Python scripts associated with this book are licensed under a Creative Commons Public Domain (CC0) License. This means that you're welcome to use any portion of the code for any purpose in your own programs.

Formatting Conventions

Code blocks will be used to present example code:

```
# This is Python code:
print("Hello, World")
```

Terminal commands follow the Unix format:

```
$ # This is a terminal command:
$ python hello-world.py
```

(The dollar signs are not part of the command.)

`Monospace text` will be used to denote a filename: `hello-world.py`.

Bold text will be used to denote a new or important term.

Keyboard shortcuts will be formatted as follows: Ctrl + S

Menu shortcuts will be formatted as follows: File > New File

Notes and important information will be highlighted as follows:

> **Note**
>
> This is a note filled in with placeholder text. The quick brown fox jumps over the lazy dog. The quick brown Python slithers over the lazy hog.

Feedback and Errata

We welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Did you find an error in the text or code? Did we leave out a topic that you'd love to know more about?

We're always looking to improve our teaching materials. Whatever the reason, please send in your feedback at the link below:

realpython.com/python-basics/feedback

Chapter 2

Setting Up Python

This book is about programming computers with Python. You could read this book from cover to cover without ever touching a keyboard, but you'd miss out on the fun part—coding!

To get the most out of this book, you need a computer with Python installed on it and a way to create, edit, and save Python code files.

In this chapter, you'll learn how to:

- Install the latest version of Python 3 on your computer
- Open **IDLE**, Python's built-in **I**ntegrated **D**evelopment and **L**earning **E**nvironment

Let's get started!

2.1 A Note on Python Versions

Many operating systems, including macOS and Linux, come with Python preinstalled. The version of Python that comes with your operating system is called the **system Python**.

The system Python is used by your operating system and is usually out of date. It's essential that you have the most recent version of Python so that you can successfully follow along with the examples in this book.

Important

Do not attempt to uninstall the system Python!

You can have multiple versions of Python installed on your computer. In this chapter, you'll install the latest version of Python 3 alongside any system Python that may already exist on your machine.

Note

Even if you already have Python 3.9 installed, it's still a good idea to skim this chapter to double-check that your environment is set up for following along with this book.

This chapter is split into three sections: Windows, macOS, and Ubuntu Linux. Find the section for your operating system and follow the steps to get set up, then skip ahead to the next chapter.

If you have a different operating system, then check out *Real Python*'s "Python 3 Installation & Setup Guide" to see if your OS is covered. Readers on tablets and mobile devices can refer to the "Online Python Interpreters" section for some browser-based options.

2.2 Windows

Follow these steps to install Python 3 and open IDLE on Windows.

Important

The code in this book is tested only against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python

Windows doesn't typically come with a system Python. Fortunately, installation involves little more than downloading and running the Python installer from the Python.org website.

Step 1: Download the Python 3 Installer

Open a web browser and navigate to the following URL:

https://www.python.org/downloads/windows/

Click *Latest Python 3 Release - Python 3.x.x* located beneath the "Python Releases for Windows" heading near the top of the page. As of this writing, the latest version was Python 3.9.

Then scroll to the bottom and click *Windows x86-64 executable installer* to start the download.

Note

If your system has a 32-bit processor, then you should choose the 32-bit installer. If you aren't sure if your computer is 32-bit or 64-bit, stick with the 64-bit installer mentioned above.

Step 2: Run the Installer

Open your Downloads folder in Windows Explorer and double-click the file to run the installer. A dialog that looks like the following one will appear:

It's okay if the Python version you see is greater than 3.9.0 as long as the version is not less than 3.

Important

Make sure you select the box that says *Add Python 3.x to PATH*. If you install Python without selecting this box, then you can run the installer again and select it.

Click Install Now to install Python 3. Wait for the installation to finish, then continue to open IDLE.

Open IDLE

You can open IDLE in two steps:

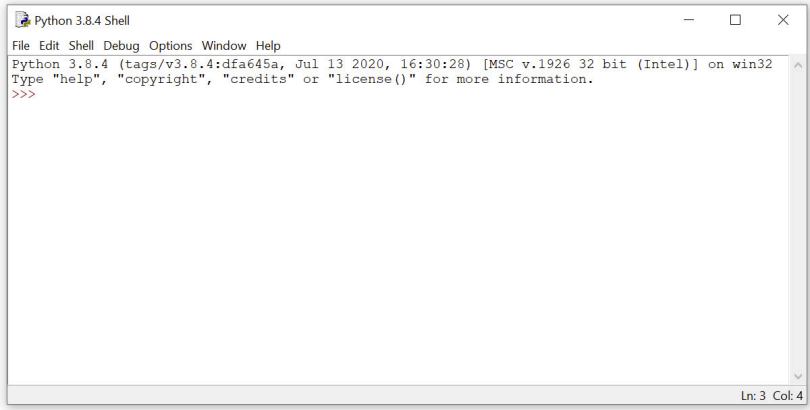
1. Click the Start menu and locate the Python 3.9 folder.
2. Open the folder and select *IDLE (Python 3.9)*.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It's a great way to get started with Python!

Note

While you're free to use a code editor other than IDLE if you prefer, note that some chapters, especially chapter 7, "Finding and Fixing Code Bugs," do contain material specific to IDLE.

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.

The >>> symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-setup

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to chapter 3.

2.3 macOS

Follow these steps to install Python 3 and open IDLE on macOS.

Important

The code in this book is tested only against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python

To install the latest version of Python 3 on macOS, download and run the official installer from the Python.org website.

Step 1: Download the Python 3 Installer

Open a web browser and navigate to the following URL:

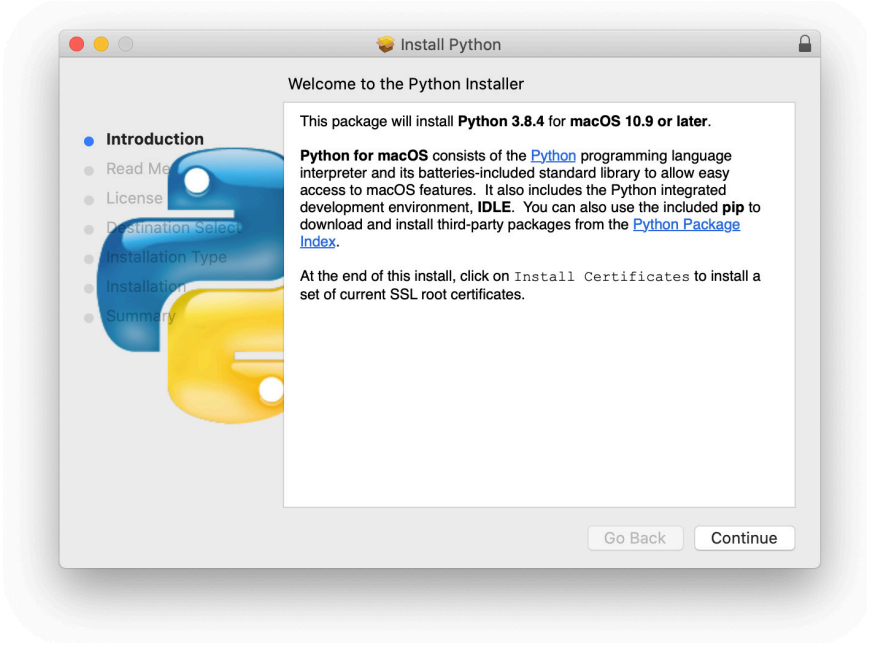
https://www.python.org/downloads/mac-osx/

Click *Latest Python 3 Release - Python 3.x.x* located beneath the "Python Releases for Mac OS X" heading near the top of the page. As of this writing, the latest version was Python 3.9.

Then scroll to the bottom of the page and click *macOS 64-bit installer* to start the download.

Step 2: Run the Installer

Open Finder and double-click the downloaded file to run the installer. A dialog box that looks like the following will appear:



Press Continue a few times until you are asked to agree to the software license agreement. Then click Agree.

You'll be shown a window that tells you where Python will be installed and how much space it will take. You most likely don't want to change the default location, so go ahead and click Install to start the installation.

When the installer is finished copying files, click Close to close the installer window.

Open IDLE

You can open IDLE in three steps:

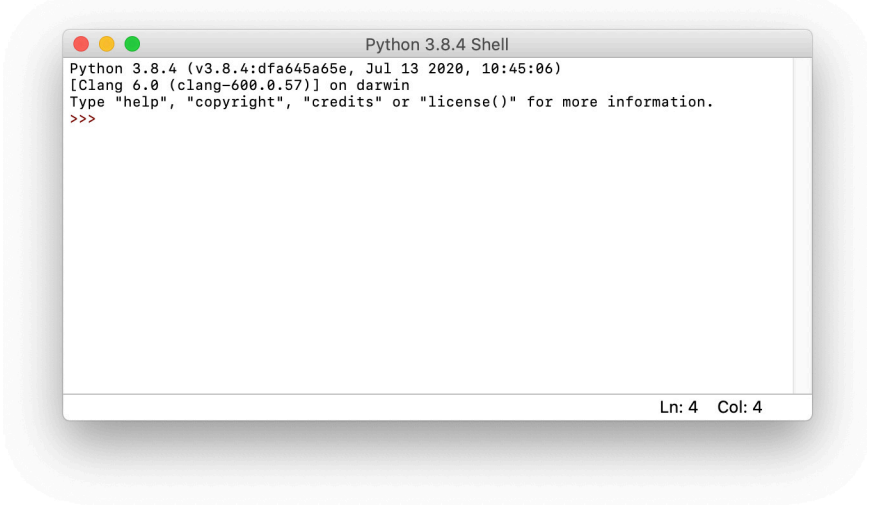
1. Open Finder and click *Applications*.
2. Double-click the Python 3.9 folder.
3. Double-click the IDLE icon.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It's a great way to get started with Python!

Note

While you're free to use a code editor other than IDLE if you prefer, note that some chapters, especially chapter 7, "Finding and Fixing Code Bugs," do contain material specific to IDLE.

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.

The >>> symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-setup

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to chapter 3.

2.4 Ubuntu Linux

Follow these steps to install Python 3 and open IDLE on Ubuntu Linux.

Important

The code in this book is tested only against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python

There's a good chance that your Ubuntu distribution already has Python installed, but it probably won't be the latest version, and it may be Python 2 instead of Python 3.

To find out what version(s) you have, open a terminal window and try the following commands:

```
$ python --version
$ python3 --version
```

One or more of these commands should respond with a version, as below:

```
$ python3 --version
Python 3.9.0
```

Your version number may vary. If the version shown is Python 2.x or a version of Python 3 that is less than 3.9, then you want to install the latest version. How you install Python on Ubuntu depends on which version of Ubuntu you're running. You can determine your local Ubuntu version by running the following command:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.1 LTS
Release:        18.04
Codename:       bionic
```

Look at the version number next to `Release` in the console output, and follow the corresponding instructions below.

Ubuntu 18.04 or Greater

Ubuntu version 18.04 does not come with Python 3.9 by default, but it is in the Universe repository. You can install it with the following commands in the Terminal application:

```
$ sudo apt-get update
$ sudo apt-get install python3.9 idle-python3.9 python3-pip
```

Note that because the Universe repository is usually behind the Python release schedule, you may not get the latest version of Python 3.9. However, any version of Python 3.9 will work for this book.

Ubuntu 17 and Lower

For Ubuntu versions 17 and lower, Python 3.9 is not in the Universe repository. You need to get it from a Personal Package Archive (PPA). To install Python from the deadsnakes PPA, run the following commands in the Terminal application:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install python3.9 idle-python3.9 python3-pip
```

You can check that the correct version of Python was installed by running `python3 --version`. If you see a version number less than 3.9, then you may need to type `python3.9 --version`. Now you can open IDLE and get ready to write your first Python program.

Open IDLE

You can open IDLE from the command line by typing the following:

```
$ idle-python3.9
```

On some Linux installations, you can open IDLE with the following shortened command:

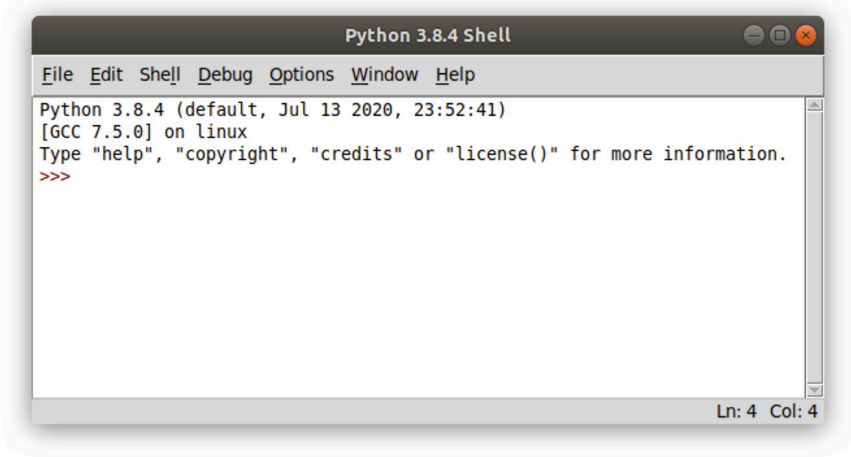
```
$ idle3
```

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It's a great way to get started with Python!

Note

While you're free to use a code editor other than IDLE if you prefer, note that some chapters, especially chapter 7, "Finding and Fixing Code Bugs," do contain material specific to IDLE.

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.

Important

If you open IDLE with the `idle3` command and see a version less than 3.9 displayed in the Python shell window, then you'll need to open IDLE with the `idle-python3.9` command.

The `>>>` symbol that you see in the IDLE window is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-setup

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to chapter 3.

Chapter 3

Your First Python Program

Now that you have the latest version of Python installed on your computer, it's time to start coding!

In this chapter, you will:

- Write your first Python program
- Learn what happens when you run a program with an error
- Learn how to declare a variable and inspect its value
- Learn how to write comments

Ready to begin your Python journey? Let's go!

3.1 Write a Python Program

If you don't already have IDLE open, then go ahead and open it. There are two main windows that you'll work with in IDLE: the **interactive window**, which is the one that opens when you start IDLE, and the **editor window**.

You can type code into both the interactive window and the editor window. The difference between the two windows is in how they execute code. In this section, you'll learn how to execute Python code in both windows.

The Interactive Window

IDLE's interactive window contains a **Python shell**, which is a textual user interface used to interact with the Python language. You can type a bit of Python code into the interactive window and press Enter to immediately see the results. Hence the name *interactive* window.

The interactive window opens automatically when you start IDLE. You'll see the following text, with some minor differences depending on your setup, displayed at the top of the window:

```
Python 3.9.0 (tags/v3.9.0:1b293b6)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This text shows the version of Python that IDLE is running. You can also see information about your operating system and some commands you can use to get help and view information about Python.

The `>>>` symbol in the last line is called the **prompt**. This is where you'll type in your code.

Go ahead and type `1 + 1` at the prompt and press Enter:

```
>>> 1 + 1
2
>>>
```

Python evaluates the expression, displays the result (`2`), then displays another prompt. Every time you run some code in the interactive window, a new prompt appears directly below the result.

Executing Python in the interactive window can be described as a loop with three steps:

1. Python **reads** the code entered at the prompt.
2. Python **evaluates** the code.
3. Python **prints** the result and waits for more input.

This loop is commonly referred to as a **r**ead-**e**valuate-**p**rint **l**oop and is abbreviated as **REPL**. Python programmers sometimes refer to the Python shell as the Python REPL, or just "the REPL" for short.

Let's try something a little more interesting than adding numbers. A rite of passage for every programmer is writing a program that prints the phrase "Hello, World" on the screen.

At the prompt in the interactive window, type the word `print` followed by a set of parentheses with the text `"Hello, World"` inside:

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

A **function** is code that performs some task and can be invoked by a name. The above code invokes, or **calls**, the `print()` function with the text `"Hello, World"` as input.

The parentheses tell Python to call the `print()` function. They also enclose everything that gets sent to the function as input. The quotation marks indicate that `"Hello, World"` really is text and not something else.

Note

IDLE **highlights** parts of your code in different colors as you type to make it easier for you to identify the different parts.

By default, functions are highlighted in purple and text is highlighted in green.

The interactive window executes a single line of code at a time. This is useful for trying out small code examples and exploring the Python language, but it has a major limitation: you have to enter your code one line at a time!

Alternatively, you can save Python code in a text file and execute all of the code in the file to run an entire program.

The Editor Window

You'll write your Python files using IDLE's editor window. You can open the editor window by selecting File > New File from the menu at the top of the interactive window.

The interactive window stays open when you open the editor window. It displays the output generated by code in the editor window, so you'll want to arrange the two windows so that you can see them both at the same time.

In the editor window, type in the same code you used to print `"Hello, World"` in the interactive window:

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

IDLE highlights code typed into the editor window just like in the interactive window.

Important

When you write code in a Python file, you don't need to include the `>>>` prompt.

Before you run your program, you need to save it. Select File > Save from the menu and save the file as `hello_world.py`.

Note

On some systems, the default directory for saving files in IDLE is the Python installation directory. **Do not** save your files to this directory. Instead, save them to your desktop or to a folder in your user's home directory.

The `.py` extension indicates that a file contains Python code. In fact, saving your file with any other extension removes the code highlighting. IDLE only highlights Python code when it's stored in a `.py` file.

Running Python Programs in the Editor Window

To run your program, select Run > Run Module from the menu in the editor window.

Note

Pressing F5 also runs a program from the editor window.

Program output always appears in the interactive window.

Every time you run code from a file, you'll see something like the following output in the interactive window:

```
>>> ================== RESTART ==================
```

IDLE restarts the Python interpreter, which is the computer program that actually executes your code, every time you run a file. This makes sure that programs are executed the same way each time.

Opening Python Files in the Editor Window

To open an existing file in IDLE, select File > Open from the menu, then select the file you want to open. IDLE opens every file in a new editor window, so you can have several files open at the same time.

You can also open a file from a file manager, such as Windows Explorer or macOS Finder. Right-click the file icon and select Edit with IDLE to open the file in IDLE's editor window.

Double-clicking on a `.py` file from a file manager executes the program. However, this usually runs the file with the system Python, and the program window disappears immediately after the program terminates—often before you can even see any output.

For now, the best way to run your Python programs is to open them in IDLE's editor window and run them from there.

3.2 Mess Things Up

Everybody makes mistakes—especially while programming! In case you haven't made any mistakes yet, let's get a head start and mess something up on purpose to see what happens.

Mistakes in programs are called **errors**. You'll experience two main types of errors: syntax errors and runtime errors.

Syntax Errors

A **syntax error** occurs when you write code that isn't allowed in the Python language.

Let's create a syntax error by removing the last quotation mark from the code in the `hello_world.py` file that you created in the last section:

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

Save the file and press F5 to run it. The code won't run! IDLE displays an alert box with the following message:

```
EOL while scanning string literal.
```

There are two terms in this message that may be unfamiliar:

1. A **string literal** is text enclosed in quotation marks. `"Hello, World"` is a string literal.
2. **EOL** stands for **e**nd **o**f **l**ine.

So, the message tells you that Python got to the end of a line while reading a string literal. String literals must be terminated with a quotation mark before the end of a line.

IDLE highlights the line containing `print("Hello, World)` in red to help you quickly find the line of code with the syntax error. Without the second quotation mark, everything after the first quotation mark—including the closing parenthesis—is part of a string literal.

Runtime Errors

IDLE catches syntax errors before a program starts running. In contrast, **runtime errors** only occur while a program is running.

To generate a runtime error, remove both quotation marks in the `hello_world.py` file:

```
print(Hello, World)
```

Did you notice how the text color changed to black when you removed the quotation marks? IDLE no longer recognizes `Hello, World` as text. What do you think will happen when you run the program? Press F5 to find out!

The following text displays in red in the interactive window:

```
Traceback (most recent call last):
  File "/home/hello_world.py", line 1, in <module>
    print(Hello, World)
NameError: name 'Hello' is not defined
```

Whenever an error occurs, Python stops executing the program and displays several lines of text called a **traceback**. The traceback shows useful information about the error.

Tracebacks are best read from the bottom up:

- The last line of the traceback tells you the name of the error and the error message. In this case, a `NameError` occurred because the name `Hello` is not defined anywhere.
- The second to last line shows you the code that produced the error. There's only one line of code in `hello_world.py`, so it's not hard to guess where the problem is. This information is more helpful for larger files.
- The third to last line tells you the name of the file and the line number so you can go to the exact spot in your code where the error occurred.

In the next section, you'll see how to define names for values in your code. Before you move on, though, you can get some practice with syntax errors and runtime errors by working on the review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that IDLE won't run because it has a syntax error.
2. Write a program that crashes only while it's running because it has a runtime error.

3.3 Create a Variable

In Python, **variables** are names that can be assigned a value and then used to refer to that value throughout your code.

Variables are fundamental to programming for two reasons:

1. **Variables keep values accessible:** For example, you can assign the result of some time-consuming operation to a variable so that your program doesn't have to perform the operation each time you need to use the result.
2. **Variables give values context:** The number `28` could mean lots of different things, such as the number of students in a class, the number of times a user has accessed a website, and so on. Giving the value `28` a name like `num_students` makes the meaning of the value clear.

In this section, you'll learn how to use variables in your code, as well as some of the conventions Python programmers follow when choosing names for variables.

The Assignment Operator

An **operator** is a symbol, such as `+`, that performs an operation on one or more values. For example, the `+` operator takes two numbers, one to the left of the operator and one to the right, and adds them together.

Values are assigned to variable names using a special symbol called the **assignment operator** (`=`) . The `=` operator takes the value to the right of the operator and assigns it to the name on the left.

Let's modify the `hello_world.py` file from the previous section to assign some text in a variable before printing it to the screen:

```
>>> greeting = "Hello, World"
>>> print(greeting)
Hello, world
```

On the first line, you create a variable named `greeting` and assign it the value `"Hello, World"` using the `=` operator.

`print(greeting)` displays the output `Hello, World` because Python looks for the name `greeting`, finds that it's been assigned the value `"Hello, World"`, and replaces the variable name with its value before calling the function.

If you hadn't executed `greeting = "Hello, World"` before executing `print(greeting)`, then you would have seen a `NameError` like you did when you tried to execute `print(Hello, World)` in the previous section.

Note

Although `=` looks like the equals sign from mathematics, it has a different meaning in Python. This distinction is important and can be a source of frustration for beginner programmers.

Just remember, whenever you see the `=` operator, whatever is to the right of it is being assigned to a variable on the left.

Variable names are **case sensitive**, so a variable named `greeting` is not the same as a variable named `Greeting`. For instance, the following code produces a `NameError`:

```
>>> greeting = "Hello, World"
>>> print(Greeting)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Greeting' is not defined
```

If you have trouble with an example in this book, double-check that every character in your code—including spaces—matches the example *exactly*. Computers have no common sense, so being almost correct isn't good enough!

Rules for Valid Variable Names

Variable names can be as long or as short as you like, but there are a few rules that you must follow. Variable names may contain uppercase and lowercase letters (A–Z, a–z), digits (0–9), and underscores (_), but they cannot begin with a digit.

For example, each of the following is a valid Python variable name:

- `string1`
- `_a1p4a`
- `list_of_names`

The following aren't valid variable names because they start with a digit:

- `9lives`
- `99_balloons`
- `2beOrNot2Be`

In addition to English letters and digits, Python variable names may contain many different valid Unicode characters.

Unicode is a standard for digitally representing characters used in most of the world's writing systems. That means variable names can contain letters from non-English alphabets, such as decorated letters

like é and ü, and even Chinese, Japanese, and Arabic symbols.

However, not every system can display decorated characters, so it's a good idea to avoid them if you're going to share your code with people in different regions.

Note

You'll learn more about Unicode in chapter 12.

You can also read about Python's support for Unicode in the official Python documentation.

Just because a variable name is valid doesn't necessarily mean that it's a good name.

Choosing a good name for a variable can be surprisingly difficult. Fortunately, there are some guidelines that you can follow to help you choose better names.

Descriptive Names Are Better Than Short Names

Descriptive variable names are essential, especially for complex programs. Writing descriptive names often requires using multiple words. Don't be afraid to use long variable names.

In the following example, the value `3600` is assigned to the variable `s`:

```
s = 3600
```

The name `s` is totally ambiguous. Using a full word makes it a lot easier to understand what the code means:

```
seconds = 3600
```

`seconds` is a better name than `s` because it provides more context. But it still doesn't convey the full meaning of the code. Is `3600` the number of seconds it takes for a process to finish, or is it the length of a movie? There's no way to tell.

The following name leaves no doubt about what the code means:

```
seconds_per_hour = 3600
```

When you read the above code, there's no question that `3600` is the number of seconds in an hour. `seconds_per_hour` takes longer to type than both the single letter `s` and the word `seconds`, but the payoff in clarity is massive.

Although naming variables descriptively means using longer variable names, you should avoid using excessively long names. A good rule of thumb is to limit variable names to three or four words maximum.

Python Variable Naming Conventions

In many programming languages, it's common to write variable names in **mixedCase**. In this system, you capitalize the first letter of every word except the first and leave all other letters in lowercase. For example, `numStudents` and `listOfNames` are written in mixedCase.

In Python, however, it's more common to write variable names in **lower_case_with_underscores**. In this system, you leave every letter in lowercase and separate each word with an underscore. For instance, both `num_students` and `list_of_names` are written using the lower_case_with_underscores system.

There's no rule mandating that you write your variable names in lower_case_with_underscores. The practice is codified, though, in a document called PEP 8, which is widely regarded as the official style guide for writing Python.

Note

PEP stands for **P**ython **E**nhancement **P**roposal. A PEP is a design document used by the Python community to propose new features to the language.

Following the standards outlined in PEP 8 ensures that your Python code is readable by most Python programmers. This makes sharing code and collaborating with other people easier for everyone involved.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Using the interactive window, display some text using `print()`.
2. Using the interactive window, assign a string literal to a variable. Then print the contents of the variable using the `print()` function.
3. Repeat the first two exercises using the editor window.

3.4 Inspect Values in the Interactive Window

Type the following into IDLE's interactive window:

```
>>> greeting = "Hello, World"
>>> greeting
'Hello, World'
```

When you press Enter after typing `greeting` a second time, Python prints the string literal assigned to `greeting` even though you didn't use the `print()` function. This is called **variable inspection**.

Now print the string assigned to `greeting` using the `print()` function:

```
>>> print(greeting)
Hello, World
```

Can you spot the difference between the output displayed by using `print()` and the output displayed by just entering the variable name and pressing Enter?

When you type the variable name `greeting` and press Enter, Python prints the value assigned to the variable as it appears in your code. You assigned the string literal `"Hello, World"` to `greeting`, which is why `'Hello, World'` is displayed with quotation marks.

Note

String literals can be created with single or double quotation marks in Python. At *Real Python*, we use double quotes wherever possible, whereas IDLE output appears in single quotes by default.

Both `"Hello, World"` and `'Hello, World'` mean the same thing in Python—what's most important is that you be consistent in your usage. You'll learn more about strings in chapter 4.

On the other hand, `print()` displays a more human-readable representation of the variable's value which, for string literals, means displaying the text without quotation marks.

Sometimes, both printing and inspecting a variable produce the same output:

```
>>> x = 2
>>> x
2
>>> print(x)
2
```

Here, you assign the number `2` to `x`. Both using `print(x)` and inspecting `x` display output without quotation marks because `2` is a number and not text. In most cases, though, variable inspection gives you more useful information than `print()`.

Suppose you have two variables: `x`, which is assigned the number `2`, and `y`, which is assigned the string literal `"2"`. In this case, `print(x)` and `print(y)` both display the same thing:

```
>>> x = 2
>>> y = "2"
>>> print(x)
2
>>> print(y)
2
```

However, inspecting `x` and `y` shows the difference between each variable's value:

```
>>> x
2
>>> y
'2'
```

The key takeaway here is that `print()` displays a readable representation of a variable's value, while variable inspection displays the value as it appears in the code.

Keep in mind that variable inspection works only in the interactive window. For example, try running the following program from the editor window:

```
greeting = "Hello, World"
greeting
```

The program executes without any errors, but it doesn't display any output!

3.5 Leave Yourself Helpful Notes

Programmers sometimes read code they wrote a while ago and wonder, "What does this do?" When you haven't looked at code in a while, it can be difficult to remember why you wrote it the way you did!

To help avoid this problem, you can leave comments in your code. **Comments** are lines of text that don't affect the way a program runs. They document what code does or why the programmer made certain decisions.

How to Write a Comment

The most common way to write a comment is to begin a new line in your code with the `#` character. When you run your code, Python ignores lines starting with `#`.

Comments that start on a new line are called **block comments**. You can also write **inline comments**, which are comments that appear on the same line as the code they reference. Just put a `#` at the end of the line of code, followed by the text in your comment.

Here's an example of a program with both kinds of comments:

```
# This is a block comment.
greeting = "Hello, World"
print(greeting)  # This is an inline comment.
```

Of course, you can still use the `#` symbol inside a string. For instance, Python won't mistake the following for the start of a comment:

```
>>> print("#1")
#1
```

In general, it's a good idea to keep comments as short as possible, but sometimes you need to write more than reasonably fits on a single line. In that case, you can continue your comment on a new line that also begins with the `#` symbol:

```
# This is my first program.
# It prints the phrase "Hello, World"
# The comments are longer than the code!
greeting = "Hello, World"
print(greeting)
```

You can also use comments to **comment out** code while you're testing a program. Putting a # at the beginning of a line of code lets you run your program as if that line of code didn't exist, but it doesn't actually delete the code.

To comment out a section of code in IDLE, highlight one or more lines to be commented and press:

- **Windows:** Alt + 3
- **macOS:** Ctrl + 3
- **Ubuntu Linux:** Ctrl + D

To remove comments, highlight the commented lines and press:

- **Windows:** Alt + 4
- **macOS:** Ctrl + 4
- **Ubuntu Linux:** Ctrl + Shift + D

Now let's look at some common conventions for code comments.

Conventions and Pet Peeves

According to PEP 8, comments should always be written in complete sentences with a single space between the # and the first word of the comment:

```
# This comment is formatted to PEP 8.

#this one isn't
```

For inline comments, PEP 8 recommends at least two spaces between

the code and the # symbol:

```
phrase = "Hello, World"  # This comment is PEP 8 compliant.
print(phrase)# This comment isn't.
```

PEP 8 recommends that comments be used sparingly. A major pet peeve among programmers is comments that describe what is already obvious from reading the code.

For example, the comment in the following code is unnecessary:

```
# Print "Hello, World"
print("Hello, World")
```

The comment is unnecessary because the code itself explicitly describes what's happening. Comments are best used to clarify code that may be difficult to understand or to explain why something is coded a certain way.

3.6 Summary and Additional Resources

In this chapter, you wrote and executed your first Python program! You wrote a small program that displays the text `"Hello, World"` using the `print()` function.

Then you learned about **syntax errors**, which occur *before* IDLE executes a program that contains invalid Python code, and **runtime errors**, which only occur *while* a program is running.

You saw how to assign values to **variables** using the **assignment operator** (=) and how to inspect variables in the interactive window.

Finally, you learned how to write helpful **comments** in your code for when you or someone else looks at it in the future.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-first-program

Additional Resources

To learn more, check out the following resources:

- "11 Beginner Tips for Learning Python Programming"
- "Writing Comments in Python (Guide)"

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 4

Strings and String Methods

Many programmers, regardless of their specialty, deal with text on a daily basis. For example, web developers work with text input from web forms. Data scientists process text to extract data and perform tasks like sentiment analysis, which can help identify and classify opinions in a body of text.

Collections of text in Python are called **strings**. Special functions called **string methods** are used to manipulate strings. There are string methods for changing a string from lowercase to uppercase, removing whitespace from the beginning or end of a string, replacing parts of a string with different text, and much more.

In this chapter, you'll learn how to:

- Manipulate strings with string methods
- Work with user input
- Deal with strings of numbers
- Format strings for printing

Let's get started!

4.1 What Is a String?

In chapter 3, you created the string `"Hello, World"` and printed it in IDLE's interactive window using `print()`. In this section, you'll get a deeper look into exactly what strings are and the various ways you can create them in Python.

The String Data Type

Strings are one of the fundamental Python data types. The term **data type** refers to what kind of data a value represents. Strings are used to represent text.

> **Note**
>
> There are several other data types built into Python. For example, you'll learn about numerical data types in chapter 5 and Boolean data types in chapter 8.

We say that strings are a **fundamental data type** because they can't be broken down into smaller values of a different type. Not all data types are fundamental. You'll learn about compound data types, also known as **data structures**, in chapter 9.

The string data type has a special abbreviated name in Python: `str`. You can see this by using `type()`, which is a function used to determine the data type of a given value.

Type the following into IDLE's interactive window:

```
>>> type("Hello, World")
<class 'str'>
```

The output `<class 'str'>` indicates that the value `"Hello, World"` is an instance of the `str` data type. That is, `"Hello, World"` is a string.

Note

For now, you can think of the word *class* as a synonym for *data type*, although it actually refers to something more specific. You'll see just what a class is in chapter 10.

`type()` also works for values that have been assigned to a variable:

```
>>> phrase = "Hello, World"
>>> type(phrase)
<class 'str'>
```

Strings have three important properties:

1. Strings contain individual letters or symbols called **characters**.
2. Strings have a **length**, defined as the number of characters the string contains.
3. Characters in a string appear in a **sequence**, which means that each character has a numbered position in the string.

Let's take a closer look at how strings are created.

String Literals

As you've already seen, you can create a string by surrounding some text with quotation marks:

```
string1 = 'Hello, World'
string2 = "1234"
```

You can use either single quotes (`string1`) or double quotes (`string2`) to create a string as long as you use the same type at the beginning and end of the string.

Whenever you create a string by surrounding text with quotation marks, the string is called a **string literal**. The name indicates that the string is literally written out in your code. All the strings you've seen thus far are string literals.

Note

Not every string is a string literal. Sometimes strings are input by a user or read from a file. Since they're not typed out with quotation marks in your code, they're not string literals.

The quotes surrounding a string are called **delimiters** because they tell Python where a string begins and where it ends. When one type of quotes is used as the delimiter, the other type can be used inside the string:

```
string3 = "We're #1!"
string4 = 'I said, "Put it over by the llama."'
```

After Python reads the first delimiter, it considers all the characters after it part of the string until it reaches a second matching delimiter. This is why you can use a single quote in a string delimited by double quotes, and vice versa.

If you try to use double quotes inside a string delimited by double quotes, you'll get an error:

```
>>> text = "She said, "What time is it?""
  File "<stdin>", line 1
    text = "She said, "What time is it?""
                        ^
SyntaxError: invalid syntax
```

Python throws a `SyntaxError` because it thinks the string ends after the second `"`, and it doesn't know how to interpret the rest of the line. If you need to include a quotation mark that matches the delimiter inside a string, then you can **escape** the character using a backslash:

```
>>> text = "She said, \"What time is it?\""
>>> print(text)
She said, "What time is it?"
```

Note

When you work on a project, it's a good idea to use only single quotes or only double quotes to delimit every string.

Keep in mind that there really isn't a right or wrong choice! The goal is to be consistent because consistency helps make your code easier to read and understand.

Strings can contain any valid Unicode character. For example, the string `"We're #1!"` contains the pound sign (`#`) and `"1234"` contains numbers. `"×Pýthøŋ×"` is also a valid Python string!

Determine the Length of a String

The number of characters contained in a string, including spaces, is called the **length** of the string. For example, the string `"abc"` has a length of `3`, and the string `"Don't Panic"` has a length of `11`.

Python has a built-in `len()` function that you can use to determine the length of a string. To see how it works, type the following into IDLE's interactive window:

```
>>> len("abc")
3
```

You can also use `len()` to get the length of a string that's assigned to a variable:

```
>>> letters = "abc"
>>> len(letters)
3
```

First, you assign the string `"abc"` to the variable `letters`. Then you use `len()` to get the length of `letters`, which is `3`.

Multiline Strings

The PEP 8 style guide recommends that each line of Python code contain no more than seventy-nine characters—including spaces.

Note

PEP 8's seventy-nine-character line length is a recommendation, not a rule. Some Python programmers prefer a slightly longer line length.

In this book, we'll strictly follow PEP 8's recommended line length.

Whether you follow PEP 8 or choose a longer line length, sometimes you'll need to create string literals with more characters than your chosen limit.

To deal with long strings, you can break them up across multiple lines into **multiline strings**. For example, suppose you need to fit the following text into a string literal:

> This planet has—or rather had—a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movements of small green pieces of paper, which is odd because on the whole it wasn't the small green pieces of paper that were unhappy.
>
> — Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

This paragraph contains far more than seventy-nine characters, so any line of code containing the paragraph as a string literal violates PEP 8. So, what do you do?

There are a couple of ways to tackle this. One way is to break the string up across multiple lines and put a backslash (\) at the end of all but the

last line. To be PEP 8 compliant, the total length of the line, including the backslashes, must be seventy-nine characters or fewer.

Here's how you could write the paragraph as a multiline string using the backslash method:

```
paragraph = "This planet has—or rather had—a problem, which was \
this: most of the people living on it were unhappy for pretty much \
of the time. Many solutions were suggested for this problem, but \
most of these were largely concerned with the movements of small \
green pieces of paper, which is odd because on the whole it wasn't \
the small green pieces of paper that were unhappy."
```

Notice that you don't have to close each line with a quotation mark. Normally, Python would get to the end of the first line and complain that you didn't close the string with a matching double quote. With a backslash at the end, you can keep writing the same string on the next line.

When you `print()` a multiline string that's broken up by backslashes, the output is displayed on a single line:

```
>>> long_string = "This multiline string is \
displayed on one line"
>>> print(long_string)
This multiline string is displayed on one line
```

You can also create multiline strings using triple quotes (`"""` or `'''`) as delimiters. Here's how to write a long paragraph using this approach:

```
paragraph = """This planet has—or rather had—a problem, which was
this: most of the people living on it were unhappy for pretty much
of the time. Many solutions were suggested for this problem, but
most of these were largely concerned with the movements of small
green pieces of paper, which is odd because on the whole it wasn't
the small green pieces of paper that were unhappy."""
```

Triple-quoted strings preserve whitespace, including newlines. This means that running `print(paragraph)` would display the string on multiple lines, just as it appears in the string literal. This may or may not be what you want, so you'll need to think about the desired output before you choose how to write a multiline string.

To see how whitespace is preserved in a triple-quoted string, type the following into IDLE's interactive window:

```
>>> print("""An example of a
...     string that spans across multiple lines
...         and also preserves whitespace.""")
An example of a
    string that spans across multiple lines
        and also preserves whitespace.
```

Notice how the second and third lines in the output are indented in exactly the same way as the string literal.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Print a string that uses double quotation marks inside the string.
2. Print a string that uses an apostrophe inside the string.
3. Print a string that spans multiple lines with whitespace preserved.
4. Print a string that is coded on multiple lines but gets printed on a single line.

4.2 Concatenation, Indexing, and Slicing

Now that you know what a string is and how to declare string literals in your code, let's explore some of the things you can do with strings.

In this section, you'll learn about three basic string operations:

1. **Concatenation**, which joins two strings together
2. **Indexing**, which gets a single character from a string
3. **Slicing**, which gets several characters from a string at once

Let's dive in!

String Concatenation

You can combine, or **concatenate**, two strings using the `+` operator:

```
>>> string1 = "abra"
>>> string2 = "cadabra"
>>> magic_string = string1 + string2
>>> magic_string
'abracadabra'
```

In this example, the string concatenation occurs on the third line. You concatenate `string1` and `string2` using `+`, and then you assign the result to the variable `magic_string`. Notice that the two strings are joined without any whitespace between them.

You can use string concatenation to join two related strings, such as joining a first name and a last name into a full name:

```
>>> first_name = "Arthur"
>>> last_name = "Dent"
>>> full_name = first_name + " " + last_name
>>> full_name
'Arthur Dent'
```

Here, you use string concatenation twice on the same line. First, you concatenate `first_name` with `" "` to ensure a space appears after the first name in the final string. This produces the string `"Arthur "`, which you then concatenate with `last_name` to produce the full name `"Arthur Dent"`.

String Indexing

Each character in a string has a numbered position called an **index**. You can access the character at the *n*th position by putting the number *n* between two square brackets (`[]`) immediately after the string:

```
>>> flavor = "fig pie"
>>> flavor[1]
'i'
```

`flavor[1]` returns the character at position `1` in `"fig pie"`, which is `i`.

Wait. Isn't `f` the first character of `"fig pie"`?

In Python—and in most other programming languages—counting always starts at zero. To get the character at the beginning of a string, you need to access the character at position `0`:

```
>>> flavor[0]
'f'
```

Important

Forgetting that counting starts with zero and trying to access the first character in a string with the index `1` results in an **off-by-one error**.

Off-by-one errors are a common source of frustration for beginning and experienced programmers alike!

The following figure shows the index for each character of the string `"fig pie"`:

If you try to access an index beyond the end of a string, then Python raises an `IndexError`:

```
>>> flavor[9]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    flavor[9]
IndexError: string index out of range
```

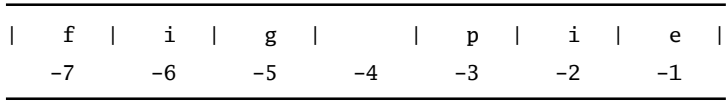
The largest index in a string is always one less than the string's length. Since `"fig pie"` has a length of seven, the largest index allowed is `6`.

Strings also support negative indices:

```
>>> flavor[-1]
'e'
```

The last character in a string has index `-1`, which for `"fig pie"` is the letter `e`. The second to last character `i` has index `-2`, and so on.

The following figure shows the negative index for each character in the string `"fig pie"`:



Just like with positive indices, Python raises an `IndexError` if you try to access a negative index less than the index of the first character in the string:

```
>>> flavor[-10]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    flavor[-10]
IndexError: string index out of range
```

Negative indices may not seem useful at first, but sometimes they're a better choice than a positive index.

For example, suppose a string input by a user is assigned to the variable `user_input`. If you need to get the last character of the string, how do you know what index to use?

One way to get the last character of a string is to calculate the final index using `len()`:

```
final_index = len(user_input) - 1
last_character = user_input[final_index]
```

Getting the final character with the index `-1` takes less typing and doesn't require an intermediate step to calculate the final index:

```
last_character = user_input[-1]
```

String Slicing

Suppose you need a string containing just the first three letters of the string `"fig pie"`. You could access each character by index and concatenate them like this:

```
>>> first_three_letters = flavor[0] + flavor[1] + flavor[2]
>>> first_three_letters
'fig'
```

If you need more than just the first few letters of a string, then getting each character individually and concatenating them together is clumsy and long-winded. Fortunately, Python provides a way to do this with much less typing.

You can extract a portion of a string, called a **substring**, by inserting a colon between two index numbers set inside square brackets like this:

```
>>> flavor = "fig pie"
>>> flavor[0:3]
'fig'
```

`flavor[0:3]` returns the first three characters of the string assigned to `flavor`, starting with the character at index `0` and going up to but not including the character at index `3`. The `[0:3]` part of `flavor[0:3]` is called a **slice**. In this case, it returns a slice of `"fig pie"`. Yum!

String slices can be confusing because the substring returned by the slice includes the character whose index is the first number but doesn't include the character whose index is the second number.

To remember how slicing works, you can think of a string as a sequence of square slots. The left and right boundaries of each slot are numbered sequentially from zero up to the length of the string, and each slot is filled with a character in the string.

Here's what this looks like for the string `"fig pie"`:

```
| f | i | g |   | p | i | e |
0   1   2   3   4   5   6   7
```

So, for `"fig pie"`, the slice `[0:3]` returns the string `"fig"`, and the slice `[3:7]` returns the string `" pie"`.

If you omit the first index in a slice, then Python assumes you want to start at index `0`:

```
>>> flavor[:3]
'fig'
```

The slice `[:3]` is equivalent to the slice `[0:3]`, so `flavor[:3]` returns the first three characters in the string `"fig pie"`.

Similarly, if you omit the second index in the slice, then Python assumes you want to return the substring that begins with the character

whose index is the first number in the slice and ends with the last character in the string:

```
>>> flavor[3:]
' pie'
```

For `"fig pie"`, the slice `[3:]` is equivalent to the slice `[3:7]`. Since the character at index `3` is a space, `flavor[3:9]` returns the substring that starts with the space and ends with the last letter: `" pie"`.

If you omit both the first and second numbers in a slice, you get a string that starts with the character at index `0` and ends with the last character. In other words, omitting both numbers in a slice returns the entire string:

```
>>> flavor[:]
'fig pie'
```

It's important to note that, unlike with string indexing, Python won't raise an `IndexError` when you try to slice between boundaries that fall outside the beginning or ending boundaries of a string:

```
>>> flavor[:14]
'fig pie'
>>> flavor[13:15]
''
```

In this example, the first line gets the slice from the beginning of the string up to but not including the fourteenth character. The string assigned to `flavor` has a length of seven, so you might expect Python to throw an error. Instead, it ignores any nonexistent indices and returns the entire string `"fig pie"`.

The third line shows what happens when you try to get a slice in which the entire range is out of bounds. `flavor[13:15]` attempts to get the thirteenth and fourteenth characters, which don't exist. Instead of raising an error, Python returns the **empty string** (`""`).

Note

The empty string is called empty because it doesn't contain any characters. You can create it by writing two quotation marks with nothing between them:

```
empty_string = ""
```

A string with anything in it—even a space—is not empty. All the following strings are non-empty:

```
non_empty_string1 = " "
non_empty_string2 = "    "
non_empty_string3 = "          "
```

Even though these strings don't contain any *visible* characters, they are non-empty because they do contain spaces.

You can use negative numbers in slices. The rules for slices with negative numbers are exactly the same as the rules for slices with positive numbers. It helps to visualize the string as slots with the boundaries labeled with negative numbers:

```
 |  f  |  i  |  g  |     |  p  |  i  |  e  |
-7    -6    -5    -4    -3    -2    -1
```

Just like before, the slice `[x:y]` returns the substring starting at index `x` and going up to but not including `y`. For instance, the slice `[-7:-4]` returns the first three letters of the string `"fig pie"`:

```
>>> flavor[-7:-4]
'fig'
```

Notice, however, that the rightmost boundary of the string does not have a negative index. The logical choice for that boundary would seem to be the number `0`, but that doesn't work.

Instead of returning the entire string, `[-7:0]` returns the empty string:

```
>>> flavor[-7:0]
''
```

This happens because the second number in a slice must correspond to a boundary that is to the right of the boundary corresponding to the first number, but both `-7` and `0` correspond to the leftmost boundary in the figure.

If you need to include the final character of a string in your slice, then you can omit the second number:

```
>>> flavor[-7:]
'fig pie'
```

Of course, using `flavor[-7:]` to get the entire string is a bit odd considering that you can use the variable `flavor` without the slice to get the same result!

Slices with negative indices are useful, though, for getting the last few characters in a string. For example, `flavor[-3:]` is `"pie"`.

Strings Are Immutable

To wrap this section up, let's discuss an important property of string objects. Strings are **immutable**, which means that you can't change them once you've created them. For instance, see what happens when you try to assign a new letter to one particular character of a string:

```
>>> word = "goal"
>>> word[0] = "f"
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    word[0] = "f"
TypeError: 'str' object does not support item assignment
```

Python throws a `TypeError` and tells you that `str` objects don't support item assignment.

If you want to alter a string, then you must create an entirely new string. To change the string `"goal"` to the string `"foal"`, you can use a string slice to concatenate the letter `"f"` with everything but the first letter of the word `"goal"`:

```
>>> word = "goal"
>>> word = "f" + word[1:]
>>> word
'foal'
```

First, you assign the string `"goal"` to the variable `word`. Then you concatenate the slice `word[1:]`, which is the string `"oal"`, with the letter `"f"` to get the string `"foal"`. If you're getting a different result here, then make sure you're including the colon character (`:`) as part of the string slice.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a string and print its length using `len()`.
2. Create two strings, concatenate them, and print the resulting string.
3. Create two strings, use concatenation to add a space between them, and print the result.
4. Print the string `"zing"` by using slice notation to specify the correct range of characters in the string `"bazinga"`.

4.3 Manipulate Strings With Methods

Strings come bundled with special functions called **string methods** that you can use to work with and manipulate strings. There are numerous string methods available, but we'll focus on some of the most commonly used ones.

In this section, you'll learn how to:

- Convert a string to uppercase or lowercase
- Remove whitespace from a string
- Determine if a string begins or ends with certain characters

Let's go!

Converting String Case

To convert a string to all lowercase letters, you use the string's `.lower()` method. This is done by tacking `.lower()` onto the end of the string itself:

```
>>> "Jean-Luc Picard".lower()
'jean-luc picard'
```

The dot (`.`) tells Python that what follows is the name of a method—the `lower()` method in this case.

Note

We'll refer to string methods with a dot (`.`) at the beginning of their names. For example, `.lower()` is written with a leading dot instead of as `lower()`.

This makes it easier to differentiate functions that are string methods from built-in functions like `print()` and `type()`.

String methods don't just work on string literals. You can also use `.lower()` on a string assigned to a variable:

```
>>> name = "Jean-Luc Picard"
>>> name.lower()
'jean-luc picard'
```

The opposite of `.lower()` is `.upper()`, which converts every character in a string to uppercase:

```
>>> name.upper()
'JEAN-LUC PICARD'
```

Compare the `.upper()` and `.lower()` string methods to the `len()` function you saw in the last section. Aside from the different results of these functions, the important distinction here is how they're used.

`len()` is a stand-alone function. If you want to determine the length of the `name` string, then you call the `len()` function directly:

```
>>> len(name)
15
```

On the other hand, `.upper()` and `.lower()` must be used in conjunction with a string. They do not exist independently.

Removing Whitespace From a String

Whitespace is any character that is printed as blank space. This includes things like spaces and **line feeds**, which are special characters that move output to a new line.

Sometimes you need to remove whitespace from the beginning or end of a string. This is especially useful when working with strings that come from user input, which may include extra whitespace characters by accident.

There are three string methods that you can use to remove whitespace from a string:

1. `.rstrip()`
2. `.lstrip()`
3. `.strip()`

`.rstrip()` removes whitespace from the right side of a string:

```
>>> name = "Jean-Luc Picard     "
>>> name
'Jean-Luc Picard     '
>>> name.rstrip()
'Jean-Luc Picard'
```

In this example, the string `"Jean-Luc Picard     "` has five trailing spaces. You use `.rstrip()` to remove trailing spaces from the right-hand side of the string. This returns the new string `"Jean-Luc Picard"`, which no longer has the spaces at the end.

`.lstrip()` works just like `.rstrip()`, except that it removes whitespace from the left-hand side of the string:

```
>>> name = "     Jean-Luc Picard"
>>> name
'     Jean-Luc Picard'
>>> name.lstrip()
'Jean-Luc Picard'
```

To remove whitespace from both the left and the right sides of the string at the same time, use `.strip()`:

```
>>> name = "     Jean-Luc Picard     "
>>> name
'     Jean-Luc Picard     '
>>> name.strip()
'Jean-Luc Picard'
```

It's important to note that none of `.rstrip()`, `.lstrip()`, or `.strip()` removes whitespace from the middle of the string. In each of the previous examples, the space between `"Jean-Luc"` and `"Picard"` is preserved.

Determine If a String Starts or Ends With a Particular String

When you work with text, sometimes you need to determine if a given string starts with or ends with certain characters. You can use two string methods to solve this problem: `.startswith()` and `.endswith()`.

Let's look at an example. Consider the string `"Enterprise"`. Here's how you use `.startswith()` to determine if the string starts with the letters `e` and `n`:

```
>>> starship = "Enterprise"
>>> starship.startswith("en")
False
```

You tell `.startswith()` which characters to search for by providing a string containing those characters. So, to determine if `"Enterprise"` starts with the letters `e` and `n`, you call `.startswith("en")`. This returns `False`. Why do you think that is?

If you guessed that `.startswith("en")` returns `False` because `"Enterprise"` starts with a capital `E`, then you're absolutely right! The `.startswith()` method is **case sensitive**. To get `.startswith()` to return `True`, you need to provide it with the string `"En"`:

```
>>> starship.startswith("En")
True
```

You can use `.endswith()` to determine if a string ends with certain characters:

```
>>> starship.endswith("rise")
True
```

Just like `.startswith()`, the `.endswith()` method is case sensitive:

```
>>> starship.endswith("risE")
False
```

Note

The `True` and `False` values are not strings. They are a special kind of data type called a **Boolean value**. You'll learn more about Boolean values in chapter 8.

String Methods and Immutability

Recall from the previous section that strings are immutable—they can't be changed once they've been created. Most string methods that alter a string, like `.upper()` and `.lower()`, actually return copies of the original string with the appropriate modifications.

If you aren't careful, this can introduce subtle bugs into your program. Try this out in IDLE's interactive window:

```
>>> name = "Picard"
>>> name.upper()
'PICARD'
>>> name
'Picard'
```

When you call `name.upper()`, nothing about `name` actually changes. If you need to keep the result, then you need to assign it to a variable:

```
>>> name = "Picard"
>>> name = name.upper()
>>> name
'PICARD'
```

`name.upper()` returns a new string `"PICARD"`, which is reassigned to the `name` variable. This **overrides** the original string `"Picard"` that you first assigned to `name`.

Use IDLE to Discover Additional String Methods

Strings have lots of methods associated with them, and the methods introduced in this section barely scratch the surface. IDLE can help you find new string methods. To see how, first assign a string literal to a variable in the interactive window:

```
>>> starship = "Enterprise"
```

Next, type `starship` followed by a period, but do not hit Enter. You should see the following in the interactive window:

```
>>> starship.
```

Now wait for a couple of seconds. IDLE displays a list of every string method, which you can scroll through using the arrow keys.

A related shortcut in IDLE is the ability to use Tab to automatically fill in text without having to type long names. For instance, if you type only `starship.u` and hit Tab, then IDLE automatically fills in `starship.upper` because only one method that begins with a `u` belongs to `starship`.

This even works with variable names. Try typing just the first few letters of `starship` and pressing Tab. If you haven't defined any other names that share those first letters, then IDLE completes the name `starship` for you.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that converts the following strings to lowercase: `"Animals"`, `"Badger"`, `"Honey Bee"`, `"Honey Badger"`. Print each lowercase string on a separate line.
2. Repeat exercise 1, but convert each string to uppercase instead of lowercase.

3. Write a program that removes whitespace from the following strings, then print out the strings with the whitespace removed:

```
string1 = "    Filet Mignon"
string2 = "Brisket    "
string3 = "  Cheeseburger   "
```

4. Write a program that prints out the result of `.startswith("be")` on each of the following strings:

```
string1 = "Becomes"
string2 = "becomes"
string3 = "BEAR"
string4 = "  bEautiful"
```

5. Using the same four strings from exercise 4, write a program that uses string methods to alter each string so that `.startswith("be")` returns `True` for all of them.

4.4 Interact With User Input

Now that you've seen how to work with string methods, let's make things interactive!

In this section, you'll learn how to get some input from a user with `input()`. You'll write a program that asks a user to input some text and then displays that text back to them in uppercase.

Enter the following into IDLE's interactive window:

```
>>> input()
```

When you press Enter, it looks like nothing happens. The cursor moves to a new line, but a new >>> doesn't appear. Python is waiting for you to enter something!

Go ahead and type some text and press Enter:

```
>>> input()
Hello there!
'Hello there!'
>>>
```

The text you entered is repeated on a new line with single quotes. That's because `input()` returns as a string any text entered by the user.

To make `input()` a bit more user-friendly, you can give it a **prompt** to display to the user. The prompt is just a string that you put between the parentheses of `input()`. It can be anything you want: a word, a symbol, a phrase—anything that is a valid Python string.

`input()` displays the prompt and waits for the user to type something. When the user hits Enter, `input()` returns their input as a string that can be assigned to a variable and used to do something in your program.

To see how `input()` works, type the following code into IDLE's editor window:

```
prompt = "Hey, what's up? "
user_input = input(prompt)
print("You said: " + user_input)
```

Press F5 to run the program. The text `Hey, what's up?` displays in the interactive window with a blinking cursor.

The single space at the end of the string `"Hey, what's up? "` makes sure that when the user starts to type, the text is separated from the prompt with a space. When the user types a response and presses Enter, their response is assigned to the `user_input` variable.

Here's a sample run of the program:

```
Hey, what's up? Mind your own business.


You said: Mind your own business.
```

Once you have input from a user, you can do something with it. For example, the following program takes user input, converts it to uppercase with `.upper()`, and prints the result:

```
response = input("What should I shout? ")
shouted_response = response.upper()
print("Well, if you insist..." + shouted_response)
```

Try typing this program into IDLE's editor window and running it. What else can you think of to do with the input?

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that takes input from the user and displays that input back.
2. Write a program that takes input from the user and displays the input in lowercase.
3. Write a program that takes input from the user and displays the number of characters in the input.

4.5 Challenge: Pick Apart Your User's Input

Write a program named `first_letter.py` that prompts the user for input with the string `"Tell me your password:"`. The program should then determine the first letter of the user's input, convert that letter to uppercase, and display it back.

For example, if the user input is `"no"`, then the program should display the following output:

```
The first letter you entered was: N
```

For now, it's okay if your program crashes when the user enters nothing as input—that is, when they just hit Enter instead of typing something. You'll learn a couple of ways to deal with this situation in an upcoming chapter.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

4.6 Working With Strings and Numbers

When you get user input using `input()`, the result is always a string. There are many other situations in which input is given to a program as a string. Sometimes those strings contain numbers that need to be fed into calculations.

In this section, you'll learn how to deal with strings of numbers. You'll see how arithmetic operations work on strings and how they often lead to surprising results. You'll also learn how to convert between strings and number types.

Using Strings With Arithmetic Operators

You've seen that string objects can hold many types of characters, including numbers. However, don't confuse numerals in a string with

actual numbers. For instance, try this bit of code out in IDLE's interactive window:

```
>>> num = "2"
>>> num + num
'22'
```

The `+` operator concatenates two strings together, which is why the result of `"2" + "2"` is `"22"` and not `"4"`.

You can multiply strings by a number as long as that number is an integer or whole number. Type the following into the interactive window:

```
>>> num = "12"
>>> num * 3
'121212'
```

`num * 3` concatenates three instances of the string `"12"` and returns the string `"121212"`.

Compare this operation to arithmetic with numbers. When you multiply the number 12 by the number 3, the result is the same as adding three 12s together. The same is true for a string. That is, `"12" * 3` can be interpreted as `"12" + "12" + "12"`. In general, multiplying a string by an integer *n* concatenates *n* copies of that string.

You can move the number on the right-hand side of the expression `num * 3` to the left, and the result is unchanged:

```
>>> 3 * num
'121212'
```

What do you think happens if you use the `*` operator between two strings?

Type `"12" * "3"` in the interactive window and press Enter:

```
>>> "12" * "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Python raises a `TypeError` and tells you that you can't multiply a sequence by a non-integer.

Note

A **sequence** is any Python object that supports accessing elements by index. Strings are sequences. You'll learn about other sequence types in chapter 9.

When you use the `*` operator with a string, Python always expects an integer on the other side of the operator.

What do you think happens when you try to add a string and a number?

```
>>> "3" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Python throws a `TypeError` because it expects the objects on both sides of the `+` operator to be of the same type.

If an object on either side of `+` is a string, then Python tries to perform string concatenation. It will only perform addition if both objects are numbers. So, to add `"3" + 3` and get `6`, you must first convert the string `"3"` to a number.

Converting Strings to Numbers

The `TypeError` examples in the previous section highlight a common problem when applying user input to an operation that requires a number and not a string: type mismatches.

Let's look at an example. Save and run the following program:

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

If you entered the number `2` at the prompt, then you would expect the output to be `4`. But in this case, you would get `22`. Remember, `input()` always returns a string, so if you input `2`, then `num` is assigned the string `"2"`, not the integer `2`. Therefore, the expression `num * 2` returns the string `"2"` concatenated with itself, which is `"22"`.

To perform arithmetic on numbers contained in a string, you must first convert them from a string type to a number type. There are two functions that you can use to do this: `int()` and `float()`.

`int()` stands for **integer** and converts objects into whole numbers, whereas `float()` stands for **floating-point number** and converts objects into numbers with decimal points. Here's what using each one looks like in the interactive window:

```
>>> int("12")
12

>>> float("12")
12.0
```

Notice how `float()` adds a decimal point to the number. Floating-point numbers always have at least one decimal place of precision. For this reason, you can't change a string that looks like a floating-point number into an integer because you would lose everything after the decimal point.

Try converting the string `"12.0"` to an integer:

```
>>> int("12.0")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.0'
```

Even though the extra `0` after the decimal place doesn't add any value to the number, Python won't change `12.0` into `12` because it would result in a loss of precision.

Let's revisit the program from the beginning of this section and see how to fix it. Here's the code again:

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

The issue is on the line `doubled_num = num * 2` because `num` is a string and `2` is an integer.

You can fix the problem by passing `num` to either `int()` or `float()`. Since the prompts asks the user to input a number, and not specifically an integer, let's convert `num` to a floating-point number:

```
num = input("Enter a number to be doubled: ")
doubled_num = float(num) * 2
print(doubled_num)
```

Now when you run this program and input `2`, you get `4.0` as expected. Try it out!

Converting Numbers to Strings

Sometimes you need to convert a number to a string. You might do this, for example, if you need to build a string from some preexisting variables that are assigned to numeric values.

As you've already seen, concatenating a number with a string produces a `TypeError`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + num_pancakes + " pancakes."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Since `num_pancakes` is a number, Python can't concatenate it with the string `"I'm going to eat"`. To build the string, you need to convert `num_pancakes` to a string using `str()`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + str(num_pancakes) + " pancakes."
'I am going to eat 10 pancakes.'
```

You can also call `str()` on a number literal:

```
>>> "I am going to eat " + str(10) + " pancakes."
'I am going to eat 10 pancakes.'
```

`str()` can even handle arithmetic expressions:

```
>>> total_pancakes = 10
>>> pancakes_eaten = 5
>>> "Only " + str(total_pancakes - pancakes_eaten) + " pancakes left."
'Only 5 pancakes left.'
```

In the next section, you'll learn how to format strings neatly to display values in a nice, readable manner. Before you move on, though, check your understanding with the following review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a string containing an integer, then convert that string into an actual integer object using `int()`. Test that your new object is a number by multiplying it by another number and displaying the result.
2. Repeat the previous exercise, but use a floating-point number and `float()`.
3. Create a string object and an integer object, then display them side by side with a single print statement using `str()`.
4. Write a program that uses `input()` twice to get two numbers from the user, multiplies the numbers together, and displays the result. If the user enters `2` and `4`, then your program should print the following text:

```
The product of 2 and 4 is 8.0.
```

4.7 Streamline Your Print Statements

Suppose you have a string, `name = "Zaphod"`, and two integers, `heads = 2` and `arms = 3`. You want to display them in the string `"Zaphod has 2 heads and 3 arms"`. This is called **string interpolation**, which is just a fancy way of saying that you want to insert some variables into specific locations in a string.

One way to do this is with string concatenation:

```
>>> name + " has " + str(heads) + " heads and " + str(arms) + " arms"
'Zaphod has 2 heads and 3 arms'
```

This code isn't the prettiest, and keeping track of what goes inside or outside the quotes can be tough. Fortunately, there's another way of interpolating strings: **formatted string literals**, more commonly

known as **f-strings**.

The easiest way to understand f-strings is to see them in action. Here's what the above string looks like when written as an f-string:

```
>>> f"{name} has {heads} heads and {arms} arms"
'Zaphod has 2 heads and 3 arms'
```

There are two important things to notice about the above example:

1. The string literal starts with the letter `f` before the opening quotation mark.
2. Variable names surrounded by curly braces (`{}`) are replaced by their corresponding values without using `str()`.

You can also insert Python expressions between the curly braces. The expressions are replaced with their result in the string:

```
>>> n = 3
>>> m = 4
>>> f"{n} times {m} is {n*m}"
'3 times 4 is 12'
```

It's a good idea to keep any expressions used in an f-string as simple as possible. Packing a bunch of complicated expressions into a string literal can result in code that is difficult to read and difficult to maintain.

f-strings are available only in Python version 3.6 and above. In earlier versions of Python, you can use `.format()` to get the same results. Returning to the Zaphod example, you can use `.format()` to format the string like this:

```
>>> "{} has {} heads and {} arms".format(name, heads, arms)
'Zaphod has 2 heads and 3 arms'
```

f-strings are shorter and sometimes more readable than using `.format()`. You'll see f-strings used throughout this book.

For an in-depth guide to f-strings and comparisons to other string formatting techniques, check out *Real Python*'s "Python 3's f-Strings: An Improved String Formatting Syntax (Guide)."

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a `float` object named `weight` with the value 0.2, and create a string object named `animal` with the value `"newt"`. Then use these objects to print the following string using only string concatenation:

   ```
   0.2 kg is the weight of the newt.
   ```

2. Display the same string by using `.format()` and empty `{}` placeholders.
3. Display the same string using an f-string.

4.8 Find a String in a String

One of the most useful string methods is `.find()`. As its name implies, this method allows you to find the location of one string in another string—commonly referred to as a **substring**.

To use `.find()`, tack it to the end of a variable or a string literal with the string you want to find typed between the parentheses:

```
>>> phrase = "the surprise is in here somewhere"
>>> phrase.find("surprise")
4
```

The value that `.find()` returns is the index of the first occurrence of the string you pass to it. In this case, `"surprise"` starts at the fifth character of the string `"the surprise is in here somewhere"`, which has index `4` because counting starts at zero.

This is a sample from "Python Basics: A Practical Introduction to Python 3"

With the full version of the book you get a complete Python curriculum to go all the way from beginner to intermediate-level. Every step along the way is explained and illustrated with short & clear code samples.

Coding exercises within each chapter and our interactive quizzes help fast-track your progress and ensure you always know what to focus on next.

Become a fluent Pythonista and gain programming knowledge you can apply in the real-world, today:

If you enjoyed the sample chapters you can purchase a full version of the book at realpython.com/pybasics-book

---