Real Python

UP TO DATE FOR PYTHON 3.7

PYTHON BASICS

A PRACTICAL INTRODUCTION TO PYTHON 3

FOURTH EDITION

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

Python Basics: A Practical Introduction to Python 3

Real Python

Python Basics

Fletcher Heisler, David Amos, Dan Bader

ISBN: 9781775093329 (paperback)

ISBN: 9781775093336 (electronic)

Cover design by Aldren Santos

Updated 2019-01-26 We would like to thank our early access readers for their excellent feedback.

This is an Early Access version of "Python Basics: A Practical Introduction to Python 3"

With your help we can make this book even better:

At the end of each section of the book you'll find a "magical" feedback link. Clicking the link takes you to an **online feedback form** where you can share your thoughts with us.

We welcome any and all feedback or suggestions for improvement you may have.

Please feel free to be as terse or detailed as you see fit. All feedback is stored anonymously, but you can choose to leave your name and contact information so we can follow up or mention you on our "Thank You" page.

We use a different feedback link for each section, so we'll always know which part of the book your notes refer to.

Thank you for helping us make this book an even more valuable learning resource for the Python community.

— Dan Bader, Editor-in-Chief at Real Python

What Pythonistas Say About *Python Basics: A Practical Introduction to Python 3*

"I love [the book]! The wording is casual, easy to understand, and makes the information flow well. I never feel lost in the material, and it's not too dense so it's easy for me to review older chapters over and over.

I've looked at over 10 different Python tutorials/books/online courses, and I've probably learned the most from Real Python!"

— Thomas Wong

"Three years later and I still return to my Real Python books when I need a quick refresher on usage of vital Python commands."

— Rob Fowler

"I floundered for a long time trying to teach myself. I slogged through dozens of incomplete online tutorials. I snoozed through hours of boring screencasts. I gave up on countless crufty books from big-time publishers. And then I found Real Python.

The easy-to-follow, step-by-step instructions break the big concepts down into bite-sized chunks written in plain English. The authors never forget their audience and are consistently thorough and detailed in their explanations. I'm up and running now, but I constantly refer to the material for guidance."

— Jared Nielsen

"I love the book because at the end of each particular lesson there are real world and interesting challenges. I just built a savings estimator that actually reflects my savings account – neat!"

— Drew Prescott

"As a practice of what you taught I started building simple scripts for people on my team to help them in their everyday duties. When my managers noticed that, I was offered a new position as a developer.

I know there is heaps of things to learn and there will be huge challenges, but I finally started doing what I really came to like.

Once again: MANY THANKS!"

— Kamil

"What I found great about the Real Python courses compared to others is how they explain things in the simplest way possible.

A lot of courses, in any discipline really, require the learning of a lot of jargon when in fact what is being taught could be taught quickly and succinctly without too much of it. The courses do a very good job of keeping the examples interesting."

— Stephen Grady

"After reading the first Real Python course I wrote a script to automate a mundane task at work. What used to take me three to five hours now takes less than ten minutes!"

— Brandon Youngdale

"Honestly, throughout this whole process what I found was just me looking really hard for things that could maybe be added or improved, but this tutorial is amazing! You do a wonderful job of explaining and teaching Python in a way that people like me, a complete novice, could really grasp.

The flow of the lessons works perfectly throughout. The exercises truly helped along the way and you feel very accomplished when you finish up the book. I think you have a gift for making Python seem more attainable to people outside the programming world.

This is something I never thought I would be doing or learning and with a little push from you I am learning it and I can see that it will be nothing but beneficial to me in the future!"

— Shea Klusewicz

"The authors of the courses have NOT forgotten what it is like to be a beginner – something that many authors do – and assume nothing about their readers, which makes the courses fantastic reads. The courses are also accompanied by some great videos as well as plenty of references for extra learning, homework assignments and example code that you can experiment with and extend.

I really liked that there was always full code examples and each line of code had good comments so you can see what is doing what.

I now have a number of books on Python and the Real Python ones are the only ones I have actually finished cover to cover, and they are hands down the best on the market. If like me, you're not a programmer (I work in online marketing) you'll find these courses to be like a mentor due to the clear, fluff-free explanations! Highly recommended!"

— Craig Addyman

About the Authors

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 a million Python developers each month with free programming tutorials and in-depth learning resources.

Everyone who worked on this book is a *practitioner* with several years of professional experience in the software industry. Here are the members of the Real Python Tutorial Team who worked on *Python Basics*:

Fletcher Heisler is the founder of Hunter2, where he teaches developers how to hack and secure modern web apps. As one of the founding members of Real Python, Fletcher wrote the original version of this book in 2012.

David Amos is a mathematician by training, a data scientist/Python developer by profession, and a coffee junkie by choice. He is a member of the Real Python tutorial team and rewrote large parts of this book to update it to Python 3.

Dan Bader is the editor-in-chief at Real Python and a complete Python nut. When he's not busy coding or sipping coffee he helps Python developers take their coding skills to the next level with tutorials, books, and online training.

Contents

Foreword

Hello and welcome to **Python Basics: A Practical Introduction to Python 3**. I hope you are ready to learn why so many professional and hobbyist developers are drawn to Python and how you can begin using it on your 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 as well as complete beginners.

If you don't have a Computer Science degree, don't worry. Fletcher, David, and Dan 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 are considering 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. Here you get blocks that represent programming concepts (variables, loops, method calls, etc) and you drag and drop them on a visual surface. Scratch may be easy to get started with for simple programs. But you cannot build professional applications with it. Name one Fortune 500 company that powers its core business logic with Scratch. Came 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. Whatever web browser you used today was likely written in C/C++. Your operating system running that browser was also very likely 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. In the Scratch level, 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++, yet there is no gentle on-ramp. You dive head first into all the complexity of that 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 word" test. That is, what syntax and actions are necessary to get that 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 near the end.

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 with 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("https://realpython.com")
html = resp.text
print(html[205:294])
```

Incredibly, that's it. When run, the output is (something like):

```
<title>Python Tutorials – Real Python</title>
<meta name="author" content="Real Python">
```

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

So there you have a simple powerful starter example. On the real apps side of things, we have many incredible applications written in Python as well. You may have heard of, or even used, a site called YouTube. It's written in Python and processes 1,000,000 requests / second. Instagram is another example of a Python application. More close to home, we even have realpython.com (written in Django and Python 3) and my sites such as talkpython.fm (written in Pyramid and Python 3).

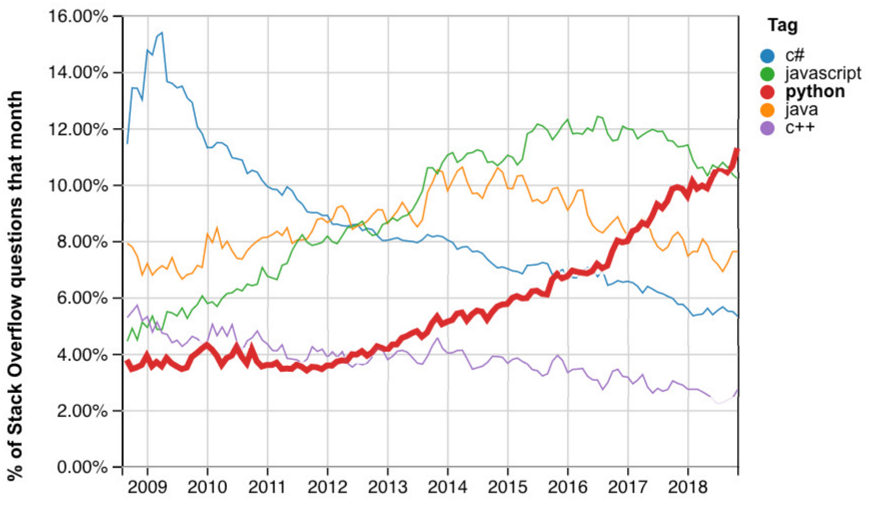
This full-spectrum aspect of Python means you can start easy and adopt more advanced features as you need them when your application demands grow.

Python Is Popular

You might have heard that Python is popular. On one hand, it may seem that it doesn't really matter how popular a language is if you can build the app you want to build with it. For better or worse, in software development popularity is a strong indicator of the quality of libraries you will have available as well the number of job openings there are. In short, you should tend to gravitate towards more popular technologies as there will be more choices and integrations available.

So, is Python actually that popular? Yes it is. You'll of course find a lot of hype and hyperbole. But there are plenty of stats to back this one. Let's look at some analytics available and presented by StackOverflow.com.

They run a site called **StackOverflow Trends**. Here you can look at the trends for various technologies by tag. When we compare Python to the other likely candidates you could pick to learn programming, you'll see one is unlike the others.



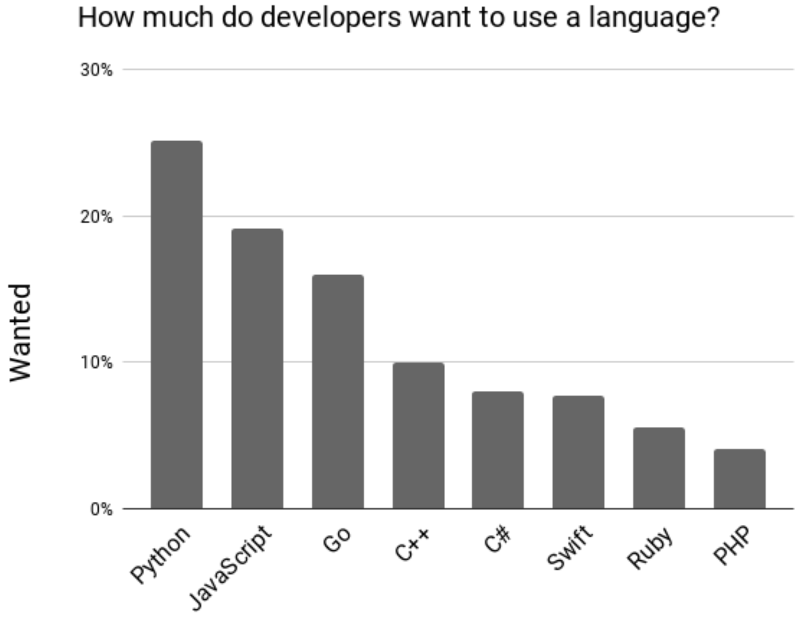
You can explore this chart and create similar charts to this one over at insights.stackoverflow.com/trends.

Notice the incredible growth of Python compared to the flatline or even downward trend of the other usual candidates! If you are 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. StackOverflow does a yearly survey of developers. It's comprehensive and very well done. You can find the full 2018 results at insights.stackoverflow.com/survey/2018/. From that writeup, I'd like to call your attention to a section entitled Most Loved, Dreaded, and Wanted Languages. In the most wanted section, you'll find responses for:

> *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 well above even 2nd place.



So if you agree with me that the relative popularity of a programming language matters. Python is clearly a good choice.

We Don't Need You to Be a Computer Scientist

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

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

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

Rather than manually exporting and scraping data from the web or spreadsheets, with Python you can scrape 1,000's of data sources or spreadsheets in the time it takes you to do just one manually. Python skills can be what takes your **biology power** and amplifies it well beyond your colleagues' and makes it your **superpower**.

Dan and Real Python

Finally, let me leave you with a comment on your authors. Dan Bader along with the other Real Python authors work day in and 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 3.7! In this book you'll learn real-world Python programming techniques, illustrated with useful and interesting examples.

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

No matter what your ultimate goals may be, if you work with a computer at all, you will 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? 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 additional useful tools 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 simple 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 how many professional products are built on Python code: Instagram, YouTube, Reddit, Spotify, to name just a few.

Not only is Python a friendly and fun language to learn—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 who are 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, this book is for you—whether you’re a complete beginner or already dabbled in Python or other languages before.

Python Basics is written in plain English and breaks down the core concepts you really need to know into bite-sized chunks. This means you’ll know “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 to learn 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 you’ll never use, it also isn’t any fun!

This book is built on the 80/20 principle. We will cover the commands and techniques used in the vast majority of cases and focus on how to program real-world solutions to problems that will help make your life easier.

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 into more advanced territory on your own will be a breeze.

So dive in! Learn to program in a widely used, free language that can do more than you ever thought was possible.

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 a 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 Email Newsletter

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 do not need any prior experience with programming to get started. The second half is focused on finding practical solutions to interesting, real-world coding problems.

As a beginner, we recommend that you go through the first half of this book from start 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 are 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 a number of 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, you may want to supplement the first few chapters with additional practice. We recommend working through the *Python Fundamentals* tutorials available for free at realpython.com to make sure you are 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 will 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 will 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 you could finish the book in as little as 35 to 40 hours. If you're new to programming you may need to spend up to 100 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 & Learning Resources

Online Resources

This book comes with a number of free bonus resources that you can access at realpython.com/python-basics/resources. On this web page you can also find an errata list with corrections maintained by the Real Python team.

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 keeps score of which questions you answered correctly.

At the end of the quiz you receive a grade based on your result. If you don't score 100% on your first try—don't fret! These quizzes are meant to challenge you and it's expected that you 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

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.

Note

The code found in this book has been tested with Python 3.6 and Python 3.7 on Windows, macOS, and Linux.

Formatting Conventions

Code blocks will be used to present example code:

```python
# This is Python code:
print("Hello world!")
```

Terminal commands follow the Unix format:

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

(Dollar signs are not part of the command.)

Italic text will be used to denote a file name: *hello-world.py*.

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

Notes and Warning boxes appear 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.

Warning

This is a warning also filled in with placeholder text. The quick brown fox jumps over the lazy dog. The quick brown Python slithers over the lazy hog.

Feedback & 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 you would 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

Leave feedback on this section »

Chapter 2

Setting Up Python

This book is about programming computers with Python. You could read this book cover-to-cover and absorb the information 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 to have a computer with Python installed on it and a way to create, edit, and save Python code files.

In this chapter, you will 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

Note

Even if you already have Python 3.7 installed, it is still a good idea to skim this chapter to double check that your environment is set-up for following along with this book.

Throughout this book, IDLE will be used to create and modify Python code files. If you have a different preferred code editor, then you may follow along with the examples using that editor.

Just know that some sections, particularly the material covered in Chapter 7, will not apply to code editors other than IDLE.

Many operating systems, such as macOS and Linux, come with Python pre-installed. The version of Python that comes with your operating system is called your **system Python**.

The system Python is almost always out-of-date, and may not even be a full Python installation. It's essential that you have the most recent version of Python so that you can follow along successfully with the examples in this book.

There are two major versions of Python available: Python 2, also known as legacy Python, and Python 3. Python 2 was released in the year 2000 and will reach its end-of-life on January 1, 2020. This book focuses exclusively on Python 3.

The chapter is split into three sections: Windows, macOS, and Linux. Just find the section for your operating system and follow the steps to get your computer set-up, then skip ahead to the next chapter.

If you have a different operating system, check out the Python 3 Installation & Setup Guide maintained on realpython.com to see if your OS is covered.

Let's dig in!

Leave feedback on this section »

2.1 Windows

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

Install Python

Windows systems do not typically ship with Python pre-installed. Fortunately, installation does not involve much more than downloading the Python installer from the python.org website and running it.

Step 1: Download the Python 3 Installer

Open a browser window and navigate to the download page for Windows at python.org.

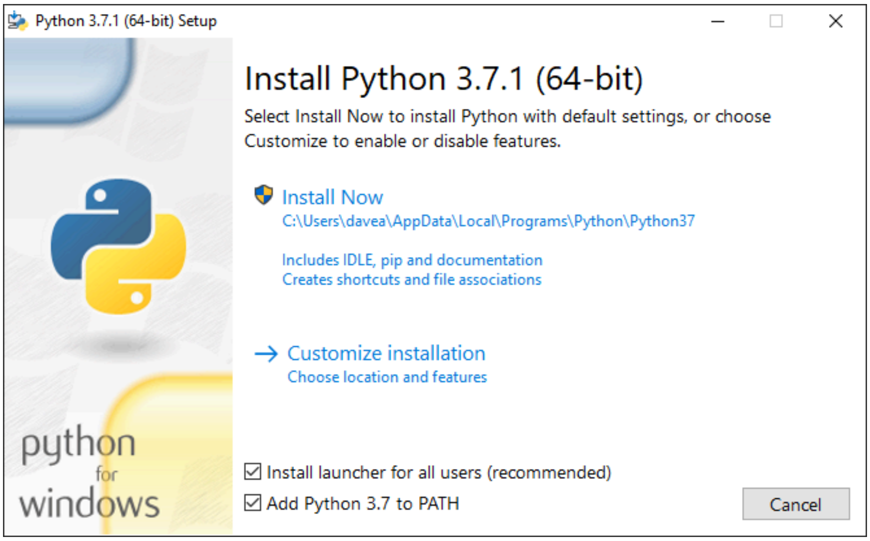
Underneath the heading at the top that says *Python Releases for Windows*, click on the link for the *Latest Python 3 Release - Python 3.x.x*. As of this writing, the latest version is Python 3.7. Then scroll to the bottom and select *Windows x86-64 executable installer*.

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

Run the installer by double-clicking on the downloaded file. You should see the following window:


Python 3.7.1 (64-bit) Setup

Install Python 3.7.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\davea\AppData\Local\Programs\Python\Python37

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.7 to PATH

Warning

Make sure you check the box that says *Add Python 3.x to PATH* as shown to ensure that the install places the interpreter in your execution path.

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

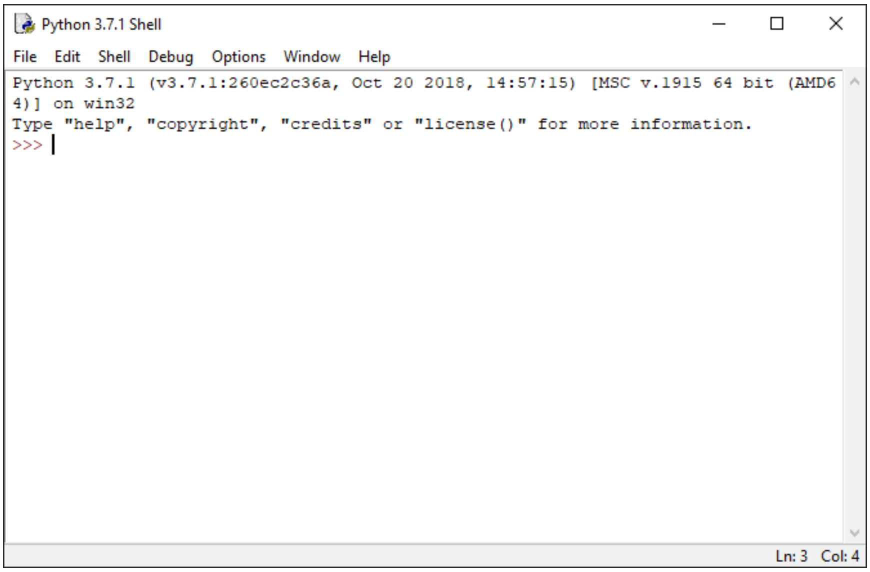
Open IDLE

You can open IDLE in two steps:

1. Click on the start menu and locate the *Python 3.7* folder.
2. Open the folder and select *IDLE (Python 3.7)*.

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

The Python shell window looks like this:



```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
```

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 3.7, 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/python-basics-2

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

Leave feedback on this section »

2.2 macOS

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

Note

Many resources recommend installing Python 3 on macOS with the Homebrew package manager. Community guides like The Hitchhiker's Guide to Python also recommend this approach, as does Real Python's Python 3 Installation & Setup Guide.

Homebrew is useful for installing packages for macOS, including Python, from the terminal. While Homebrew is something you may want to learn to use, the process of getting Homebrew installed and using it to install Python can be daunting for a beginner.

If you are interested in using Homebrew, check out the Python 3 Installation & Setup Guide for step-by-step instructions.

Install Python

Most macOS machines come with Python 2 installed. You'll want to install the latest version of Python 3. You can do this by downloading an installer from the python.org website.

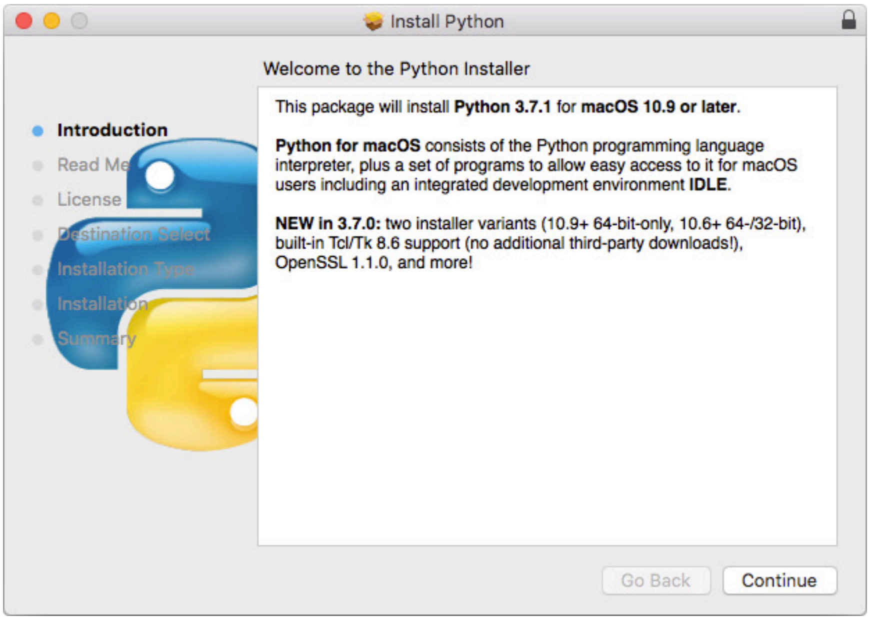
Step 1: Download the Python 3 Installer

Open a browser window and navigate to the download page for macOS at python.org.

Underneath the heading at the top that says *Python Releases for macOS*, click on the link for the *Latest Python 3 Release - Python 3.x.x*. As of this writing, the latest version is Python 3.7. Then scroll to the bottom of the page and select *macOS 64-bit/32-bit installer*. This starts the download.

Step 2: Run the Installer

Run the installer by double-clicking on the downloaded file. You should see the following window:



Welcome to the Python Installer

This package will install **Python 3.7.1** for **macOS 10.9 or later**.

Python for macOS consists of the Python programming language interpreter, plus a set of programs to allow easy access to it for macOS users including an integrated development environment **IDLE**.

NEW in 3.7.0: two installer variants (10.9+ 64-bit-only, 10.6+ 64-/32-bit), built-in Tcl/Tk 8.6 support (no additional third-party downloads!), OpenSSL 1.1.0, and more!

1. Press the *Continue* button a few times until you are asked to agree to the software license agreement. Then click *Agree*. You are shown a window that tells you where Python will be installed and how much space it will take.
2. You most likely don't want to change the default location, so go ahead and click *Install* to start the installation. The Python installer will tell you when it is finished copying files.
3. Click *Close* to close the installer window. Now that Python is installed, you can open up IDLE and get ready to write your first Python program.

Open IDLE

You can open IDLE in three steps:

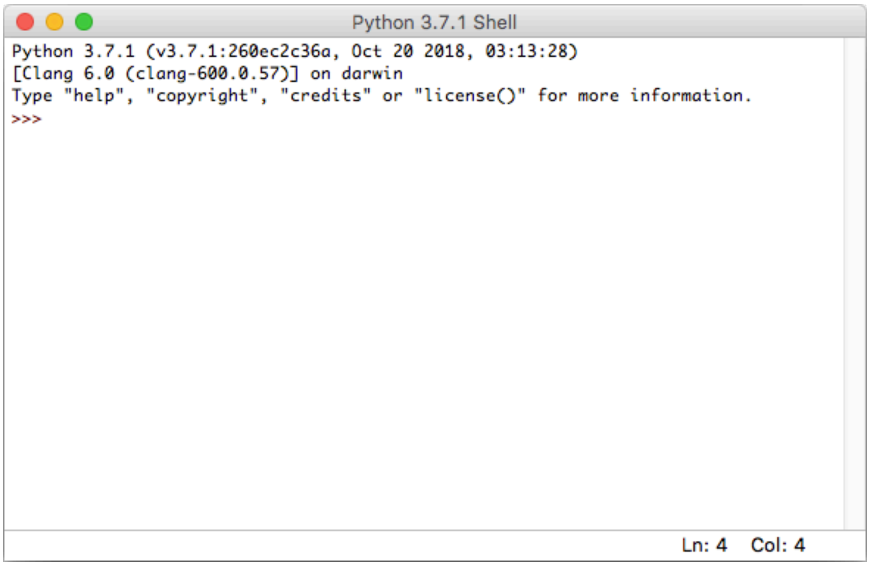
1. Open Finder and click on *Applications*.
2. Locate the *Python 3.7* folder and double-click on it.
3. Double-click on the IDLE icon.

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

Note

To open IDLE even more quickly, press `Cmd+Spacebar` to open the Spotlight search, type the word `idle`, and press `Return`.

The Python shell window looks like this:



```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 03:13:28)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

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 3.7, 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/python-basics-2

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

Leave feedback on this section »

2.3 Ubuntu Linux

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

Install Python

There is a good chance your Ubuntu distribution has Python installed already, 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 (your version number may vary):

```
$ python3 --version
Python 3.7.2
```

If the version shown is Python 2.x or a version of Python 3 that is less than 3.7, then you want to install the latest version. How you install Python on Ubuntu depends on which version of Ubuntu you are 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+** do not come with Python 3.7 by default, but it is in the Universe repository. You should be able to install it with the following commands:

  ```
  $ sudo apt-get update
  $ sudo apt-get install python3.7 idle-python3.7
  ```

- If you are using **Ubuntu 17 and lower**, Python 3.7 is not in the Universe repository, and you need to get it from a Personal Package Archive (PPA). To install Python from the "deadsnakes" PPA, do the following:

  ```
  $ sudo add-apt-repository ppa:deadsnakes/ppa
  $ sudo apt-get update
  $ sudo apt-get install python3.7 idle-python3.7
  ```

You can check that the correct version of Python was installed by running `python3 --version`. If you see a version number less than `3.7`, you may need to type `python3.7 --version`. Now you are ready to open IDLE and get ready to write your first Python program.

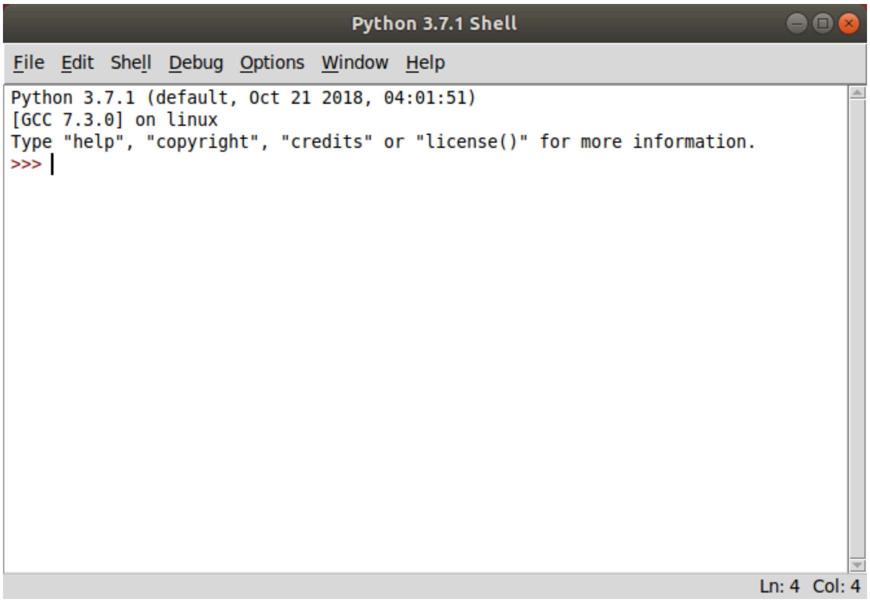
Open IDLE

On many Linux installations you can open IDLE from the command line by typing:

```
$ idle3
```

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

The Python shell window looks like this:



```
Python 3.7.1 (default, Oct 21 2018, 04:01:51)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
```

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 3.7, you may need to revisit the installation instructions in the previous section.

Warning

If you see a version other than `3.7`, or if the `idle3` command does not work, you may need to open IDLE with the following command:

```
$ idle-python3.7
```

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/python-basics-2

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

Leave feedback on this section »

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 script
- Learn what happens when you run a script 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!

Leave feedback on this section »

3.1 Write a Python Script

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

You can type code into both the interactive and script windows. The difference between the two is how the code is executed. In this section, you will write your first Python program and learn how to run it in both windows.

The Interactive Window

The interactive window contains a **Python shell**, which is a textual user interface used to interact with the Python language. Hence the name "interactive window."

When you first open IDLE, the text displayed looks something like this:

```
Python 3.7.2 (default, Dec 25 2018, 03:50:46)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

The first line tells you what version of Python is running. In this case, IDLE is running Python 3.7.2. The second and third lines give some information about the operating system and some commands you can use to get more information about Python.

The `>>>` symbol in the last line is called the **prompt**. This is where you will type in your code. Go ahead and type `1 + 1` at the prompt and press `Enter`.

When you hit `Enter`, Python evaluates the expression, displays the result `2`, and then prompts you for more input:

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

Notice that the Python prompt `>>>` appears again after your result. Python is ready for more instructions! Every time you run some code, a new prompt appears directly below the output.

The sequence of events in the interactive window can be described as a loop with three steps:

1. First, Python reads the code entered at the prompt.
2. Then the code is evaluated.
3. Finally, the output is printed in the window and a new prompt is displayed.

This loop is commonly referred to as a **R**ead-**E**valuate-**P**rint **L**oop, or **REPL**. Python programmers sometimes refer the the Python shell as a "Python REPL", or just "the REPL" for short.

Note

From this point on, the final `>>>` prompt displayed after executing code in the interactive window is excluded from code examples.

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

To print text to the screen in Python, you use the `print()` function. A **function** is a bit of code that typically takes some **input**, does something with that input, and produces some **output**.

Loosely speaking, functions in code work like mathematical functions. For example, the mathematical function $A(r)=\pi r^2$ takes the radius `r` of a circle as input and produces the area of the circle as output.

Warning

The analogy to mathematical functions has some problems, though, because code functions can have **side effects**. A side effect occurs anytime a function performs some operation that changes something about the program or the computer running the program.

For example, you can write a function in Python that takes someone's name as input, stores the name in a file on the computer, and then outputs the path to the file with the name in it. The operation of saving the name to a file is a side effect of the function.

You'll learn more about functions, including how to write your own, in Chapter 6.

Python's `print()` function takes some text as input and then displays that text on the screen. To use `print()`, type the word `print` at the prompt in the interactive window, followed by the text `"Hello, world"` inside of parentheses:

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

`"Hello, world"` must be written with double quotation marks so that Python interprets it as text and not something else.

Note

As you type code into the interactive window, you may notice that the font color changes for certain parts of the code. IDLE **highlights** parts of your different colors to help make it easier for you to identify what the different parts are.

By default, built-in functions, such as `print()` are displayed in purple, and text is displayed in green.

The interactive window can execute only 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. Code must be entered in by a person one line at a time!

Alternatively, you can store some Python code in a text file and then execute all of the code in the file with a single command. The code in the file is called a **script**, and files containing Python scripts are called **script files**.

Script files are nice not only because they make it easier to run a program, but also because they can be shared with other people so that they can run your program, too.

The Script Window

Scripts are written using IDLE's script window. You can open the script window by selecting *File › New File* from the menu at the top of the interactive window.

Notice that when the script window opens, the interactive window stays open. Any output generated by code run in the script window is displayed in the interactive window, so you may want to rearrange the two windows so that you can see both of them at the same time.

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

```
print("Hello, world")
```

Just like the interactive window, code typed into the script window is highlighted.

Warning

When you write code in a script, you do not need to include the >>> prompt that you see in IDLE's interactive window. Keep this in mind if you copy and paste code from examples that show the REPL prompt.

Remember, though, that it's not recommended that you copy and paste examples from the book. Typing each example in yourself really pays off!

Before you can run your script, you must save it. From the menu at the top of the window, select *File › Save As …* and save the script as `hello_world.py`. The `.py` file extension is the conventional extension used to indicate that a file contains Python code.

In fact, if you save your script with any extension other than `.py`, the code highlighting will disappear and all the text in the file will be displayed in black. IDLE will only highlight Python code when it is stored in a `.py` file.

Once the script is saved, all you have to do to run the program is select *Run › Run Module* from the script window and you'll see `Hello, world` appear in the interactive window:

```
Hello, world
```

Note

You can also press `F5` to run a script from the script window.

Every time you run, or re-run, a script, you may see the following output in interactive window:

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

This is IDLE's way of separating output from distinct runs of a script.

Otherwise, if you run one script after another, it may not be clear what output belongs to which script.

To open an existing script in IDLE, select *File › Open...* from the menu in either the script window or the interactive window. Then browse for and select the script file you want to open. IDLE opens scripts in a new script window, so you can have several scripts open at a time.

Note

Double-clicking on a `.py` file from a file manager, such as Windows Explorer, does execute the script in a new window. However, the window is closed immediately when the script is done running—often before you can even see what happened.

To open the file in IDLE so that you can run it and see the output, you can right-click on the file icon (`Ctrl-Click` on macOS) and choose to *Edit with IDLE*.

Leave feedback on this section »

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 on that and mess something up on purpose to see what happens.

Mistakes made in a program are called **errors**, and there are two main types of errors you'll experience:

1. Syntax errors
2. Run-time errors

In this section you'll see some examples of code errors and learn how to use the output Python displays when an error occurs to understand what error occurred and which piece of code caused it.

Syntax Errors

In loose terms, a **syntax error** occurs when you write some code that isn't allowed in the Python language. You can create a syntax error by changing the contents of the `hello_world.py` script from the last section to the following:

```
print("Hello, world)
```

In this example, the double quotation mark at the end of `"Hello, world"` has been removed. Python won't be able to tell where the string of text ends. Save the altered script and then try to run it. What happens?

The code won't run! IDLE displays an alert box with the following message:

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

EOL stands for **E**nd **O**f **L**ine, so this message tells you that Python read all the way to the end of the line without finding the end of something called a string literal.

A **string literal** is text contained in-between two double quotation marks. The text `"Hello, world"` is an example of a string literal.

Note

For brevity, string literals are often referred to as **strings**, although the term "string" technically has a more general meaning in Python. You will learn more about strings in Chapter 4.

Back in the script window, notice that the line containing with `"Hello, world` is highlighted in red. This handy features helps you quickly find which line of code caused the syntax error.

Run-time Errors

IDLE catches syntax errors before a program starts running, but some errors can't be caught until a program is executed. These errors are known as **run-time errors** because the only occur at the time that a program is run.

To generate a run-time error, change the code in `hello_world.py` to the following:

```
print(Hello, world)
```

Now both quotation marks from the phrase `"Hello, world"` have been removed. Did you notice how the text changes color when remove the quotation marks? IDLE no longer recognizes `Hello, world` as a string.

What do you think happens when you run the script? Try it out and see!

Some red text is displayed 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
```

What happened? Python is telling you a few things:

- A `NameError` occurred.
- The error happened on line 1 of the script.
- The line that generated the error was: `print(Hello, world)`.
- The specific error was `name 'Hello' is not defined`

A `NameError` is an example of a **run-time error** because it occurs only once the program is running. Since the quotation marks around `Hello, world` are missing, Python doesn't understand that it is a string of text.

Instead, Python thinks that `Hello` and `world` are names of something else in the code, the same way that `print` is the name of a function. But the names `Hello` and `world` haven't been defined anywhere, so the program crashes.

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 run-time errors by working on the review exercises.

Review Exercises

1. Write a script that IDLE won't let you run because it has a syntax error.
2. Write a script that only crashes your program once it is already running because it has a run-time error.

Leave feedback on this section »

3.3 Create a Variable

Variables are names that you can assign to different objects and use to reference those objects throughout your code. Variables are fundamental parts of any Python program.

Values can be assigned to a variable using a special operator called the assignment operator. There are also some rules governing what names can be used for variables, as well as some conventions to guide you when choosing a name for a variable.

The Assignment Operator

Let's modify the `hello_world.py` script from the last section, again. This time, we'll use a variable to store some text before printing it to the screen:

```python
phrase = "Hello, world"
print(phrase)
```

In the first line, a variable named `phrase` is created and the value `"Hello, world"` is assigned to it using the `=` operator. Then `phrase` is displayed with the `print()` function.

Note

The `=` operator is called the **assignment operator** because it is used to assign a value to a variable. Although `=` looks like the "equals sign" from mathematics, it has a different meaning in Python.

Distinguishing the `=` operator from the "equals sign" 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.

Notice the difference in where the quotation marks are located in the script above, as compared to the `hello_word.py` script in the previous section. The quotation marks are no longer inside the parentheses of the `print()` function.

When you save and run the new script, the same output as before is displayed in the interactive window:

```
Hello, world
```

Look closely at the `print()` function in the second line of the above script. Compare that to the following Python statement:

```python
print("phrase")
```

Putting quotation marks around `phrase` just prints the word "phrase" instead of printing the text assigned to the variable named `phrase`.

Note

Text that appears in quotation marks is called a **string**. This name makes sense because a string is just a group of characters that have been strung together. You'll learn more about strings in Chapter 4.

Variable Names Are Case Sensitive

Take a look at the following script and compare it to the script at the beginning of this section:

```
Phrase = "Hello, world"
print(phrase)
```

Can you spot the difference? In this example, the first line defines the variable `Phrase`—with a capital "P"—but the second line prints the variable `phrase`. Since no variable named `phrase` is defined, running the above script will produce a `NameError`.

Python variables are case-sensitive, so the variables `Phrase` and `phrase` are two entirely different things. Likewise, functions, like `print()`, almost always start with lowercase letters. You can tell Python to `print()`, but it doesn't know how to `Print()`.

When you run into trouble with the code examples in this book, be sure to double-check that every character in your code—including spaces—exactly matches the examples. Computers can't use common sense to interpret what you meant to say, so being *almost* correct won't get a computer to do the right thing!

Rules for Valid Variable Names

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

For example, `phrase`, `string1`, `_a1p4a`, and `list_of_names` are all valid variable names, but `9lives` is not.

Note

Python variable names can contain many different valid Unicode characters. 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.

Unicode is a standard for digitally representing text used in most of the world's writing systems. You can learn more about Unicode on Wikipedia. Python's support for Unicode is also well documented in the official Python documentation.

Just because a variable name is valid doesn't necessarily mean that it is a good name. Choosing a good name for a variable can be a surprisingly difficult task. However, there are some guidelines that you can follow to help you choose better names.

Python Variable Naming Conventions

Giving your variables descriptive names is essential, especially as your programs become more complex. Descriptive variable names often require using multiple words. Don't be afraid to use long variable names if they help make it clear what the variable is referencing.

In many programming languages, it is common to write variable names in all lowercase except for the first letters of all but the first word in the variable name, such as `myPhrase` or `listOfNames`.

The practice of capitalizing the first letters of words in this manner is known as **CamelCase** because the juxtaposition of lower- and uppercase characters look like humps on a camel.

In Python, however, it is more common to write the whole variable name in lowercase with distinct words in a variable name separated by an underscore. For example, `myPhrase` is written as `my_phrase` and `listOfNames` as `list_of_names`. This style is known as **snake case**.

While there is no hard-and-fast rule mandating that you write your variable names in snake case, the practice is codified in a document called PEP 8, which is widely regarded as the official style guide for writing Python.

Following the standards outlined in PEP 8 ensures that your Python code is readable by a large number of Python programmers. This makes sharing and collaborating on code easier for everyone involved.

Note

All of the code examples in this course follow PEP 8 guidelines, so you will get a lot of exposure to what "well formatted" Python code looks like.

In this section, you learned what variables are and how to declare them in your scripts. In the following section, you'll see how to inspect the value of a variable in IDLE's interactive window. But first, make sure you've mastered the concepts in this section by working through the following review exercises.

Review Exercises

1. Using the interactive window, display some text on the screen by using the `print()` function.
2. Using the interactive window, display a string of text by saving the string to a variable, then reference the string in a `print()` function using the variable name.
3. Do each of the first two exercises again by first saving your code in

a script and running it.

Leave feedback on this section »

3.4 Inspect Values in the Interactive Window

You have seen how to assign a string to a variable and display that string with the `print()` function by saving and running a script. There is another way to display the value of a variable when you are working in the interactive window.

Type the following into IDLE's interactive window:

```
>>> my_phrase = "Hello, world"
>>> my_phrase
```

When you press `Enter` after typing `my_phrase` a second time, the following output is displayed:

```
'Hello, world'
```

Python prints the string `"Hello, world"`, and you didn't have to type `print(my_phrase)`!

Now type the following:

```
>>> print(my_phrase)
```

This time, when you hit `Enter` you see:

```
Hello, world
```

Do you see the difference in the output? It doesn't have any single quotes surrounding it. What's going on here?

When you just type `my_phrase` and press `Enter`, you are telling Python to inspect the variable `my_phrase`. The output displayed is a useful representation of the object assigned to the variable. In this case, `my_phrase` is a reference to the string `"Hello, world"`, so the output is surrounded with single quotes to indicate that it is a string object.

On the other hand, when you `print()` a variable, Python displays a more human-readable representation of the object referenced by the variable. For strings, both ways of being displayed are human-readable, but this is not the case for every kind of object.

Sometimes, the two ways of displaying a variable produce the same output:

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

Here, `x` references the integer 2. The output is not displayed in surrounding quotes because 2 is a number and not a string.

Inspecting a variable, instead of printing it, can be useful for a couple of reasons. You can use it to display the value of a variable without typing `print()`. More importantly, though, inspecting a variable often gives you more useful information than `print()`.

Suppose you have two variables: `x = 2` and `y = "2"`. Then `print(x)` and `print(y)` both display the same thing. However, typing `x` and `y` by themselves shows the difference between the values of the two variables:

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

```
>>> print(y)
2
>>> x
2
>>> y
'2'
```

The key takeaway here is that `print()` displays a readable representation of a variable's value, while simply typing the name of a variable and pressing `Enter` displays some useful representation of the object referenced by the variable.

Check out what happens when you type `print` and hit `Enter`:

```
>>> print
<built-in function print>
```

Keep in mind that you can only inspect variables like this in the interactive window. For example, save and run the following script:

```
my_phrase = "Hello, world"
my_phrase
```

The script executes without any errors, but no output is displayed! Throughout this book, you will see examples that use the interactive window to inspect variables.

Leave feedback on this section »

3.5 Leave Yourself Helpful Notes

A common experience among programmers is reading something they wrote several months ago and wondering "What the heck does this do?" To help avoid these moments, or at least make them a little less painful, you can leave yourself comments in your code.

Comments are lines of text that don't affect the way the script runs. They help to document what's supposed to be happening. Python interprets any line that starts with the `#` character as a comment.

Here is an example of the `hello_world.py` script with some comments added in:

```
# This is my first script


phrase = "Hello, world."
print(phrase)  # This line displays "Hello, world"
```

The first line doesn't do anything, because it starts with a `#`. Python ignores this line completely. Likewise, Python ignores the comment on the last line. The variable `phrase` is still printed, but everything after the `#` is ignored.

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

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

Note

It is often considered bad practice to write a comment that restates what the code following it already says. For example, the following comment would be considered unnecessary:

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

No comment is needed in the above example because the code itself explicitly describes what is being done.

Comments are best used to clarify code that may not be easy to understand. In the following example, the comment is helpful because it may not be clear what is being calculated:

```
# Calculate the area of a rectangle
area = 10 * 5
```

Many programmers strive to write **self-documenting** code, which is explicit enough to understand without inserting comments. This is not always possible, though!

If you have a lot to say, you can also create comments that span multiple lines by using a series of three single quotes (`'''`) or three double quotes (`"""`) without any spaces between them. Once you do that, everything after the `'''` or `"""` becomes a comment until you close the comment with a matching `'''` or `"""`.

Multi-line comments are frequently used at the top of a script to document what the script does, provide information about the author, and describe how to use the script. For example, the `hello_world.py` script could look something like this:

```
"""
This is my first script.
```

```
It prints the phrase "Hello, world."

The comments are longer than the script!
"""

phrase = "Hello, world."
print(phrase)
```

The first three lines are now all one comment since they fall between pairs of `"""`. You can't add a multi-line comment at the end of a line of code like with the `#` version. For example, the following script produces a `SyntaxError`:

```
print("Hello, world")  """This is
an invalid comment"""
```

Besides leaving yourself notes, comments can also be used to "comment out code" while you're testing a script. In other words, adding a `#` at the beginning of a line of code is an easy way to make sure that you don't use that line right now, even though you might want to keep it and use it later.

Leave feedback on this section »

3.6 Summary and Additional Resources

In this chapter you learned how to interact with the Python interpreter using IDLE's interactive window, including how to print something to the console using the `print()` function. You also learned how to create and save a Python script using IDLE's script editor.

In addition to interacting with the Python interpreter, you learned about three important concepts: variables, errors, and comments.

Variables are names that are assigned to objects, such as strings, using the assignment operator `=`. Python has rules about what constitutes a valid variable name, and Python programmers have adopted the convention of writing variable names in snake case rather than CamelCase. This convention is documented in PEP 8, which is Python's official style guide.

Errors are an inevitable part of programming, and in this chapter, you learned about two common errors: syntax errors, which occur when you type something that cannot be understood by the Python interpreter, and run-time errors, which can only be caught once a program is running.

Finally, comments are used to leave notes for yourself and other people who read your code. There are two types of comments: inline comments, which start with `#`, and multi-line comments that begin and end with triple quotes—either `'''` or `"""`.

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/python-basics-3

Additional Resources

To take your knowledge even further, check out the resources below:

- 11 Beginner Tips for Learning Python Programming
- Writing Comments in Python (Guide)
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 4

Strings and Methods

Strings are a fundamental **data type** in Python. In simplified terms, strings are collections of text, and they show up in many contexts. For example, strings can come from user input, data read from a file, or messages sent by other devices talking over a network.

In this chapter you will learn how to work with strings and the fundamentals of using Python methods. Becoming proficient in manipulating strings is a skill with big payoffs, because a lot of the data encountered in the real world is in the form of unstructured text. By the end of this chapter, you will know how to:

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

Let's get started!

Leave feedback on this section »

4.1 String Fundamentals

In this section, you will learn more about what a string is, as well as some of the fundamental operations you can perform on strings. Before we dive in, though, let's see why strings are so important. In loose terms, a string is a collection of text. Working with text is probably not the most glamorous part of programming. However, most programmers, regardless of their specialty, deal with text on a daily basis.

Web developers work with text that gets displayed on a web page, input from web forms, and read from databases. In business automation, programmers read and extract text from PDFs, spreadsheets, and other documents. Data scientists process text to extract data and perform things like sentiment analysis, which can help identify and classify opinions in a body of text.

The focus of this chapter is to provide you with an overview of how strings work and a variety of ways to manipulate and extract information from them. Let's start off by taking a closer look at what a string is.

What Is a String?

You can create a string by surrounding some text with quotation marks. You can use single quotes or double quotes, as long as the leading and trailing quotation marks are the same type. The single, or double, quotes that surround a string are called the string's **delimiters** because they tell Python where one string begins and ends.

Whenever you write out a string, the result is called a **string literal**. The name indicates that the string is written out literally the way it looks. Contrast this to a string that may come into your program in the form of user input. Such a string is not a string literal because it is not explicitly written out in the program's code.

Here are some examples of string literals:

```
string1 = 'Hello, world.'
string2 = "We're #1!"
string3 = "1234"
string4 = 'I said, "Put it over by the llama."'
```

`string1` and `string4` have single quotes as delimiters, while `string2` and `string3` use double quotes. Notice that `string2` and `string4` show how both kinds of quotes can be used in the same string. The string `"We're #1"` uses the single quote `'` as an apostrophe, and in the string `'I said, "Put it over by the llama."'`, double quotes are used inside of the string.

After Python reads the first delimiter, all of the characters after it are considered a part of the string until a second matching delimiter is read. 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 of a string that is delimited by double quotes, you will run into some problems:

```
>>> 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 that the string ends after the second `"` and doesn't know how to interpret the rest of the line.

Note

A common pet peeve among programmers is the use of mixed quotes for delimiting strings. When you work on a project, it is best practice to pick either single quotes or double quotes as the delimiter of choice and stick with it for the entire project.

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

Strings can contain any kind of character. For example, `string2` in the above example contains a pound sign, `#`, and `string3` contains numbers. The string `"×Pýŧħøŋ×"` is also a valid Python string!

String literals can become quite long. The PEP8 style-guide recommends that any single line of Python code should contain no more than 79 characters—including spaces. To deal with string literals that contain more than 79 characters, you can break up the string across multiple lines..

Multi-Line Strings

Suppose you need to fit the following text in 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 won't fit in PEP8's 79-character line limit, which is a good recommendation for keeping your code easy to read. So what do you do?

There are a couple of ways to tackle this. The first way is to break up the string across multiple lines and put a backslash, `\`, at the end of all but the last line. Here's an example:

```python
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, however, you can keep writing the same string on the next line. When you print a string literal that is broken up on multiple lines this way, the output is still on one line.

Another approach is to wrap your string literal with triple quotes, either `"""` or `'''`. Here is how you could write the same paragraph above using this approach:

```python
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."""
```

When you write a string literal with triple quotes, the whitespace in your string is preserved. So, if you `print(paragraph)`, the string will be broken up on multiple lines just like it is in the string literal. This may or may not be what you want, so you will need to think about what the output should be before you choose how to write a multi-line string.

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

```
>>> print("""This is a
...     string that spans across multiple lines
...         that also preserves whitespace.""")
```

The output looks like this:

```
This is a
    string that spans across multiple lines
        that also preserves whitespace.
```

Review Exercises

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 one-line string that you have written out on multiple lines.

Leave feedback on this section »

4.2 Basic String Operations

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

In this section, you'll learn three basic string operations: how to join multiple strings together into a single string, how to determine the length of the string, and how to access individual parts of a string.

String Concatenation

Two strings can be combined, or **concatenated**, using the `+` operator:

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

In the example above, string concatenation occurs on the third line where `string1` and `string2` are concatenated using `+` and the result is assigned to the variable `magic_string`. Notice that the two strings are joined together without any whitespace between them.

One common application of string concatenation is to join two related strings, such as a first and last name into a full name:

```
>>> first_name = "Jean-Luc"
>>> last_name = "Picard"
>>> full_name = first_name + " " + last_name
>>> full_name
'Jean-Luc Picard"
```

In this example, a space is added between the `first_name` and `last_name` strings by first concatenating `first_name` with `" "`, and then concatenating the result with `last_name`.

When you want to combine many strings at once inside of a `print()` function, you can also use commas to separate them. This automatically adds spaces between the strings, like so:

```
>>> print("abra", "ca", "dabra")
abra ca dabra
```

The commas have to go outside of the quotation marks, since otherwise, the commas would become part of the strings themselves.

Note

Technically speaking, when you `print()` multiple strings separated by a comma, you aren't performing string concatenation because you aren't using the `+` operator. This technique can be useful, however, when you need to display several strings on the same line and you don't want to concatenate them into a new string.

Determine the Length of a String

The length of a string is the number of characters contained in the string, including spaces. For example, the string `"abc"` has length 3, and the string `"make it so"` has length 10.

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

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

You can use `len()` to get the length of strings that have been assigned to a variable:

```
>>> my_string = "abc"
>>> string_length = len(my_string)
>>> string_length
3
```

First, the string `"abc"` is assigned to the variable `my_string`. Then `len()` is used to get the length of `my_string` and assign this value to the `string_length` variable. Finally, the value of `string_length`, which is 3, is displayed.

Notice how `my_string` is placed between the two parentheses of the `len()` function. This tells `len()` to perform its operation with the `my_-`

`string` variable, the same way that putting a string in between the parentheses of `print()` tells Python to print the string.

Access Characters in a String

A string is a sequence of characters. You can access individual characters in a string by tacking on square brackets (`[` and `]`) after the string and putting a number *n* in between the brackets to get the *nth* character in the string.

Warning

Be careful when you're using:

- parentheses: `( )`
- square brackets: `[ ]`
- curly braces: `{ }`

These all mean different things in Python, so you can never switch one for another. You'll see more examples of when each one is used as you progress through this book. For now, just be aware they're all used differently.

Type the following into IDLE's interactive window:

```
>>> flavor = "apple pie"
>>> flavor[3]
'l'
```

Typing `flavor[3]` returns the third character of the string `"apple pie"`, which is... `"l"`? Wait, isn't `'p'` the third character of `"apple pie"`?

In Python—and most other programming languages—counting always starts at 0. So in this case, `'a'` is the "zeroth" character of the string `"apple pie"`. Thus `'p'` is the first character, `'p'` is the second, and `'l'` is the third.

To display the "first" character, you need to print the 0th character:

```
>>> flavor[0]
'a'
```

The number assigned to each character's position is called the character's **index** or **subscript number**. The following table shows each character of the string `"apple pie"` and its associated index:

Character:	a	p	p	l	e		p	i	e
Index/Subscript:	0	1	2	3	4	5	6	7	8

Warning

Forgetting that counting starts with zero and trying to refer to the first character in a string with the index 1 results in what is commonly known as an **off-by-one error**.

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

What do you think happens if you try to access a character by an index that is greater than the index of the last character in the string? Try this out:

```
>>> flavor[13]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Python throws an `IndexError` telling you that the index is "out of range." The largest index that you can access in a string is one less than the length of the string. Since `"apple pie"` has length 9, the largest index you can access is 8.

Let's try something even crazier. What do you think happens if you try to access a negative index? Give this a try:

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

The index -1 is associated to the last character in the string, which for `"apple pie"` is the letter `"e"`. The negative indices for the characters in the string `"apple pie"` are shown in the table below:

Character:	a	p	p	l	e		p	i	e
Negative index:	-9	-8	-7	-6	-5	-4	-3	-2	-1

Suppose you need the string containing just the first three letters of the string `flavor = "apple 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
'app'
```

You can probably imagine how clumsy this would be for extracting lots of information from a long string. It is usually easier to get portions of a string using slices.

String Slices

You can extract a portion of a string, called a **substring**, with an extended version of the subscript operation. To do this, insert a colon between two subscript numbers inside of square brackets, like this:

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

The expression `flavor[0:3]` returns the first three characters of the string assigned to the variable `flavor`, starting at the 0th character and going up to, but not including, the 3rd character. The `[0:3]` part of `flavor[0:3]` is called a **slice** since it returns a "slice" of the string referenced by the variable `flavor`. In this case, it returns a slice of `"apple pie"`. Yum!

The number before the colon in a slice is always the index of the first character to include, while the number after the colon is the index of the first character that isn't included. If you use the colon in the brackets but omit one of the numbers in a range, Python assumes that you want to go all the way to the end of the string in that direction:

```
>>> flavor = "apple pie"
>>> flavor[:5]
'apple'
>>> flavor[5:]
' pie'
>>> flavor[:]
'apple pie'
```

Slices also support negative indices:

```
>>> flavor[:-5]
'appl'
>>> flavor[-5:]
'e pie'
>>> flavor[-8:-5]
'ppl'
```

If you try to access an index in a slice that doesn't exist, Python won't throw an `IndexError` like it does if you try to get a single character:

```
>>> flavor[:14]
'apple 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 length 9, so you might expect Python to throw an error. Instead, any non-existent indices are ignored and the entire string `"apple pie"` is returned.

The second line of the example shows what happens when you try to get a slice where the entire range is out of bounds. `flavor[13:15]` attempts to get the thirteenth and fourteenth characters, which don't exist. Instead of throwing an error, the **empty string** `""` is returned.

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:

```
>>> my_string = "goal"
>>> my_string[0] = "f"  # This won't work!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

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

Note

The term `str` is Python's internal name for the string data type.

If you want to alter a string, 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":

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

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

Review Exercises

1. Create a string and print its length using the len() function.
2. Create two strings, concatenate them, and print the resulting string.
3. Create two string variables, then print one of them after the other (with a space added in between) using a comma in your print statement.
4. Repeat exercise 3, but instead of using commas in print(), use concatenation to add a space between the two strings.
5. Print the string "zing" by using subscripts and index numbers on the string "bazinga" to specify the correct range of characters.

Leave feedback on this section »

4.3 Use String Methods

The Python programming language is an example of an **object-oriented language**, which means that data is stored in **objects**. A string is an example of an object. Don't worry too much about what it means to be an object. What is important for right now is that objects have both data and functions—called **methods**—that are used to work with the data.

In this section, you will learn about some of the methods that can be used to manipulate strings. In particular, you will learn how to convert a string to upper or lower case, how to remove whitespace from string, and how to determine if a string contains only numbers.

Converting String Case

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

```
>>> "Arthur Dent".lower()
'arthur dent'
```

The `.` tells Python that what follows is the name of a method—the `lower` method in this case. The parentheses `()` after the method's name tell Python to execute the method, just like parentheses are used to execute the `print()` and `len()` functions.

String methods don't just work on string literals. You can use the `.lower()` method on a variable name that is assigned to a string, as well:

```
>>> name = "Arthur Dent"
>>> name.lower()
'arthur dent'
```

The counterpart to the `.lower()` method is the `.upper()` method, which converts every character in a string to upper case:

```
>>> loud_voice = "Can you hear me yet?"
>>> loud_voice.upper()
'CAN YOU HEAR ME YET?'
```

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

The `len()` function doesn't belong to an object. If you want to determine the length of the `loud_voice` string, you call the `len()` function directly, like this:

```
>>> len(loud_voice)
20
```

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

Note

Throughout this book, you will see numerous examples of object methods. When referring to methods in a paragraph of text, they will be prefaced with a `.` (dot), such as `.lower()` and `.upper()`. to indicate that they are methods of an object and not general purpose functions.

Sometimes, the method's parent object is included in the reference to make it clear which object the method belongs to—for example, `loud_voice.upper()`. However, this book uses the shorter `.upper()` style of referring to a method whenever the context is clear.

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, where extra whitespace characters may have been introduced by accident.

There are three string methods that you can use to remove whitespace from a string: `.rstrip()`, `.lstrip()`, and `.strip()`. The first of these, `.rstrip()`, removes whitespace from the right side of a string:

```
>>> name = "Arthur Dent     "
>>> name
'Arthur Dent     '
>>> name.rstrip()
'Arthur Dent'
```

In this example, the string `"Arthur Dent     "` has five trailing spaces. Python doesn't remove any trailing spaces in a string automatically when the string is assigned to a variable. The `.rstrip()` method removes trailing spaces from the right-hand side of the string and returns a new string `"Arthur Dent"`, which no longer has the spaces at the end.

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

```
>>> name = "     Arthur Dent"
>>> name
'     Arthur Dent'
>>> name.lstrip()
'Arthur Dent'
```

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

```
>>> name = "     Arthur Dent     "
>>> name
'     Arthur Dent     '
>>> name.strip()
'Arthur Dent'
```

Note

None of the `.rstrip()`, `.lstrip()`, and `.strip()` methods remove whitespace from the middle of the string, which you can see in the previous examples because the space between “Arthur” and “Dent” is preserved each time.

Determine if a String Starts or Ends With a String

A common problem when working with strings is 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 must tell `.startswith()` what 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"`, 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
```

The `.endswith()` method is used 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 will 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 have 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 = "Elena"
>>> name.upper()
'ELENA'
>>> name
'Elena'
```

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

```
>>> name = "Elena"
>>> name = name.upper()
>>> name
'ELENA'
```

`name.upper()` returns a new string `"ELENA"`, which is re-assigned to the `name` variable. This **overrides** the original string `"Elena"` assigned to `"name"`.

Use IDLE to Discover Additional String Methods

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

```
>>> my_string = "kerfuffle"
```

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

```
>>> my_string.
```

Now hit `Ctrl+Space`. IDLE displays a list of every string method that you can scroll through with the arrow keys.

A related shortcut in IDLE is the ability to fill in text automatically without having to type in long names by hitting `Tab`. For instance, if you only type in `my_string.u` and then hit the `Tab` key, IDLE automatically fills in `my_string.upper` because there is only one method belonging to `my_string` that begins with a "u."

This even works with variable names. Try typing in just the first few letters of `my_string` and, assuming you don't have any other names already defined that share those first letters, IDLE completes the name `my_string` for you when you hit the TAB key.

Review Exercises

1. Write a script that converts the following strings to lowercase: `"Animals"`, `"Badger"`, `"Honey Bee"`, `"Honeybadger"`. Print each lowercase string.
2. Repeat Exercise 1, but convert each string to uppercase instead of lowercase.
3. Write a script that removes whitespace from the following strings:

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

   Print out the strings with the whitespace removed.
4. Write a script 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 script that uses string methods to alter each string so that `.startswith("be")` returns `True` for all of them.

Leave feedback on this section »

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 will learn how to get some input from a user with the `input()` function. You'll write a program that asks a user to input some text and then display that text back to them in uppercase.

To use the `input()` function, you must specify a **prompt**. The prompt is just a string that you put in between the parentheses of `input()`. It can be anything you want: a word, a symbol, a phrase—anything that is a valid Python string.

The `input()` function displays the prompt and waits for the user to type something on their keyboard. 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, save and run the following script:

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

When you run this script, you'll see `Hey, what's up?` displayed 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 you type a response and press `Enter`, your 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 script takes user input and "shouts" it back by converting the input to uppercase with `.upper()` and printing the result:

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

Review Exercises

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

Leave feedback on this section »

4.5 Challenge: Pick Apart Your User's Input

Write a script named `first_letter.py` that first prompts the user for input by using the string: `Tell me your password:` The script should then determine the first letter of the user's input, convert that letter to upper-case, and display it back.

For example, if the user input is `"no"` then the program should respond like this:

```
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, they just hit `Enter` instead of typing something in. You'll learn about a couple of ways you can deal with this situation in an upcoming chapter.

Leave feedback on this section »

4.6 Working With Strings and Numbers

When you get user input using the `input()` function, the result is always a string. There are many other times when 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 will learn how to deal with strings of numbers. You will see how arithmetic operations work on strings, and how they often lead to surprising results. You will also learn how to convert between strings and number types.

Strings and Arithmetic Operators

You've seen that string objects can hold many types of characters, including numbers. However, don't confuse string "numbers" with actual numbers. For instance, try this bit of code out in IDLE's interactive window:

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

The `+` operator, when used with two strings, concatenates strings together. So the result of `"2" + "2"` is `"22"`, not `"4"`.

Strings can be "multiplied" by an integer. Type the following into the interactive window:

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

The expression `num * 3` concatenates the string referenced by `"12"` with itself 3 times and returns the string `"121212"`. To compare this operation to arithmetic with numbers, notice that `"12" * 3 = "12" + "12" + "12"`. In other words, multiplying a string by an integer `n` concatenates that string with itself `n` times.

The number on the right-hand side of the expression `num * 3` can be moved 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 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 throws a `TypeError` and tells you that you can't multiply a sequence by a non-integer. When the `*` operator is used with a string on either the left or the right side, it always expects an integer on the other side.

Note

A **sequence** is any Python object that supports accessing elements by subscript. Strings are sequences. You will learn about other sequence types in Chapter 9.

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

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

Again, Python throws a `TypeError` because the `+` operator expects both things on either side of it to be of the same type. If any one of the objects on either side of `+` is a string, Python tries to perform string concatenation. Addition will only be performed 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` errors you saw in the previous section highlight a common problem encountered when working with user input: type mismatches when trying to use the input in an operation that requires a number and not a string.

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

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

When you enter a number, such as 2, you expect the output to be 4, but in this case, you get 22. Remember, `input()` always returns a string, so if you input 2, then `num` is assigned to the string `"2"`, not the integer `2`. Therefore, the expression `num * 2` returns the string `"2"` concatenated with itself, which is `"22"`.

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

`int()` stands for "integer" and converts objects into whole numbers, while `float()` stands for "floating-point number" and converts objects into numbers with decimal points. Here's what using them 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:

```
>>> 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 the loss of precision.

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

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

The issue lies in the line `doubled_num = num * 2` because `num` references a string and `2` is an integer. You can fix the problem by wrapping `num` with 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 script 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 pre-existing variables that are assigned to numeric values.

As you've already seen, the following 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."
```

You're not limited to numbers when using `str()`. You can pass it all sorts of objects to get their string representations:

```
>>> str(print)
'<built-in function print>'

>>> str(int)
"<class 'int'>"

>>> str(float)
"<class 'float'>"
```

These examples may not seem very useful, but they illustrate how flexible `str()` is.

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

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 by using the `str()` function.
4. Write a script that gets two numbers from the user using the `input()` function twice, multiplies the numbers together, and displays the result. If the user enters 2 and 4, your output should look like:

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

Leave feedback on this section »

This is an Early Access version of "Python Basics: A Practical Introduction to Python 3"

With your help we can make this book even better:

At the end of each section of the book you'll find a "magical" feedback link. Clicking the link takes you to an **online feedback form** where you can share your thoughts with us.

We welcome any and all feedback or suggestions for improvement you may have.

Please feel free to be as terse or detailed as you see fit. All feedback is stored anonymously, but you can choose to leave your name and contact information so we can follow up or mention you on our "Thank You" page.

We use a different feedback link for each section, so we'll always know which part of the book your notes refer to.

Thank you for helping us make this book an even more valuable learning resource for the Python community.

— Dan Bader, Editor-in-Chief at Real Python