O'REILLY®

Third Edition

Think Python

How to Think Like a Computer Scientist

Allen B. Downey

O'REILLY®

Think Python

Python is an excellent way to get started in programming, and this clear, concise guide walks you through the language a step at a time—beginning with basic programming concepts, then moving on to functions, data structures, and object-oriented programming. This revised third edition reflects the growing role of large language models (LLMs) in programming and includes exercises on effective LLM prompts, testing code, and debugging.

Through exercises in each chapter, you'll try out programming skills as you learn them. Author Allen Downey focuses on fundamental programming concepts that will remain relevant even as the tools evolve. With this popular hands-on guide, you'll learn:

- **The syntax and semantics of the Python language**
- **A clear definition of each programming concept, with emphasis on important vocabulary**
- **How to work with variables, statements, functions, and data structures**
- **Techniques for reading and writing files and databases**
- **Fundamentals of objects, methods, and object-oriented programming**
- **Debugging strategies for syntax, runtime, and semantic errors**
- **How to use LLMs to accelerate your learning—including effective prompts, testing code, and debugging**

Allen B. Downey is a professor emeritus at Olin College of Engineering and the author of several other books on programming and data science.

"An outstanding guide for interested adults to learn programming from scratch through exercises (the only way). Even better, this third edition makes readers comfortable with modern tools of the trade—Jupyter Notebooks and AI coding assistants."

—Luciano Ramalho
Author of *Fluent Python*

"An excellent introduction to Python programming without a single superfluous word or line of code. The third edition is especially exciting because it teaches you how you can use large language models to deepen your knowledge of programming even as a beginner."

—Sam Lau
Coauthor of *Learning Data Science*

PYTHON PROGRAMMING

US $48.99 CAN $61.99
ISBN: 978-1-098-15543-8

linkedin.com/company/oreilly-media
youtube.com/oreillymedia

THIRD EDITION

Think Python

How to Think Like a Computer Scientist

Allen B. Downey

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Think Python
by Allen B. Downey

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Brian Guerin
Development Editor: Jeff Bleiel
Production Editor: Christopher Faucher
Copyeditor: Sonia Saruba
Proofreader: Kim Cofer

Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

August 2012: First Edition
December 2015: Second Edition
June 2024: Third Edition

Revision History for the Third Edition
2024-05-24: First Release
2024-09-20: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098155438* for release details.

978-1-098-15543-8

[LSI]

Table of Contents

Preface

Who Is This Book For?

If you want to learn to program, you have come to the right place. Python is one of the best programming languages for beginners—and it is also one of the most in-demand skills.

You have also come at the right time, because learning to program now is probably easier than ever. With virtual assistants like ChatGPT, you don't have to learn alone. Throughout this book, I'll suggest ways you can use these tools to accelerate your learning.

This book is primarily for people who have never programmed before and people who have some experience in another programming language. If you have substantial experience in Python, you might find the first few chapters too slow.

One of the challenges of learning to program is that you have to learn *two* languages: one is the programming language itself; the other is the vocabulary we use to talk about programs. If you learn only the programming language, you are likely to have problems when you need to interpret an error message, read documentation, talk to another person, or use virtual assistants. If you have done some programming, but you have not also learned this second language, I hope you find this book helpful.

Goals of the Book

Writing this book, I tried to be careful with the vocabulary. I define each term when it first appears. And there is a glossary at the end of each chapter that reviews the terms that were introduced.

I also tried to be concise. The less mental effort it takes to read the book, the more capacity you will have for programming.

But you can't learn to program just by reading a book—you have to practice. For that reason, this book includes exercises at the end of every chapter where you can practice what you have learned.

If you read carefully and work on exercises consistently, you will make progress. But I'll warn you now—learning to program is not easy, and even for experienced programmers it can be frustrating. As we go, I will suggest strategies to help you write correct programs and fix incorrect ones.

Navigating the Book

Each chapter in this book builds on the previous ones, so you should read them in order and take time to work on the exercises before you move on.

The first six chapters introduce basic elements like arithmetic, conditionals, and loops. They also introduce the most important concept in programming, functions, and a powerful way to use them, recursion.

Chapters 7 and 8 introduce strings—which can represent letters, words, and sentences—and algorithms for working with them.

Chapters 9 through 12 introduce Python's core data structures—lists, dictionaries, and tuples—which are powerful tools for writing efficient programs. Chapter 12 presents algorithms for analyzing text and randomly generating new text. Algorithms like these are at the core of large language models (LLMs), so this chapter will give you an idea of how tools like ChatGPT work.

Chapter 13 is about ways to store data in long-term storage—files and databases. As an exercise, you can write a program that searches a filesystem and finds duplicate files.

Chapters 14 through 17 introduce object-oriented programming (OOP), which is a way to organize programs and the data they work with. Many Python libraries are written in object-oriented style, so these chapters will help you understand their design—and define your own objects.

The goal of this book is not to cover the entire Python language. Rather, I focus on a subset of the language that provides the greatest capability with the fewest concepts. Nevertheless, Python has a lot of features you can use to solve common problems efficiently. Chapter 18 presents some of these features.

Finally, Chapter 19 presents my parting thoughts and suggestions for continuing your programming journey.

What's New in the Third Edition?

The biggest changes in this edition were driven by two new technologies—Jupyter notebooks and virtual assistants.

Each chapter of this book is a Jupyter notebook, which is a document that contains both ordinary text and code. For me, that makes it easier to write the code, test it, and keep it consistent with the text. For you, it means you can run the code, modify it, and work on the exercises, all in one place. Instructions for working with the notebooks are in the first chapter.

The other big change is that I've added advice for working with virtual assistants like ChatGPT and using them to accelerate your learning. When the previous edition of this book was published in 2015, the predecessors of these tools were far less useful and most people were unaware of them. Now they are a standard tool for software engineering, and I think they will be a transformational tool for learning to program —and learning a lot of other things, too.

The other changes in the book were motivated by my regrets about the second edition. The first is that I did not emphasize software testing. That was already a regrettable omission in 2015, but with the advent of virtual assistants, automated testing has become even more important. So this edition presents Python's most widely used testing tools, `doctest` and `unittest`, and includes several exercises where you can practice working with them.

My other regret is that the exercises in the second edition were uneven—some were more interesting than others and some were too hard. Moving to Jupyter notebooks helped me develop and test a more engaging and effective sequence of exercises.

In this revision, the sequence of topics is almost the same, but I rearranged a few of the chapters and compressed two short chapters into one. Also, I expanded the coverage of strings to include regular expressions.

A few chapters use turtle graphics. In previous editions, I used Python's `turtle` module, but unfortunately it doesn't work in Jupyter notebooks. So I replaced it with a new turtle module that should be easier to use.

Finally, I rewrote a substantial fraction of the text, clarifying places that needed it and cutting back in places where I was not as concise as I could be.

I am very proud of this new edition—I hope you like it!

Getting Started

For most programming languages, including Python, there are many tools you can use to write and run programs. These tools are called integrated development environments (IDEs). In general, there are two kinds of IDEs:

- Some work with files that contain code, so they provide tools for editing and running these files.
- Others work primarily with notebooks, which are documents that contain text and code.

For beginners, I recommend starting with a notebook development environment like Jupyter. The notebooks for this book are available from an online repository at *https://allendowney.github.io/ThinkPython*. There are two ways to use them:

- You can download the notebooks and run them on your own computer. In that case, you have to install Python and Jupyter, which is not hard, but if you want to learn Python, it can be frustrating to spend a lot of time installing software.
- An alternative is to run the notebooks on Colab, which is a Jupyter environment that runs in a web browser, so you don't have to install anything. Colab is operated by Google, and it is free to use.

If you are just getting started, I strongly recommend you start with Colab.

Resources for Teachers

If you are teaching with this book, here are some resources you might find useful.

- You can find notebooks with solutions to the exercises, along with links to the additional resources listed here, at *https://allendowney.github.io/ThinkPython*.
- Quizzes for each chapter, and a summative quiz for the whole book, are available in the O'Reilly Learning Platform version of this book.
- *Teaching and Learning with Jupyter* is an online book with suggestions for using Jupyter effectively in the classroom. You can read the book at *https://jupyter4edu.github.io/jupyter-edu-book*.
- One of the best ways to use notebooks is live coding, where an instructor writes code and students follow along in their own notebooks. To learn about live coding—and get other great advice about teaching programming—I recommend the instructor training provided by The Carpentries, at *https://carpentries.github.io/instructor-training*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

Bold
: Indicates the first introduction of new technical term, which also has a corresponding glossary entry.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://allendowney.github.io/ThinkPython*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Think Python* by Allen B. Downey (O'Reilly). Copyright 2024 Allen B. Downey, 978-1-098-15543-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/think-python-3e*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Acknowledgments

Many thanks to Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language. Thanks also to Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist* (Green Tea Press).

Thanks to the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible, and thanks to the Creative Commons for the license I am using now.

Thanks to the developers and maintainers of the Python language and the libraries I used, including the turtle graphics module; the tools I used to develop the book, including Jupyter and JupyterBook; and the services I used, including ChatGPT, Copilot, Colab, and GitHub.

Thanks to the editors at Lulu who worked on *How to Think Like a Computer Scientist* and the editors at O'Reilly Media who worked on *Think Python*.

Special thanks to the technical reviewers for the second edition, Melissa Lewis and Luciano Ramalho, and for the third edition, Sam Lau and Luciano Ramalho (again!). I am also grateful to Luciano for developing the turtle graphics module I use in several chapters, called `jupyturtle`.

Thanks to all the students who worked with earlier versions of this book and all the contributors who sent in corrections and suggestions. More than one hundred sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

If you have a suggestion or correction, please email *feedback@thinkpython.com*. If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

CHAPTER 1

Programming as a Way of Thinking

The first goal of this book is to teach you how to program in Python. But learning to program means learning a new way to think, so the second goal of this book is to help you think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas—specifically computations. Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

We will start with the most basic elements of programming and work our way up. In this chapter, we'll see how Python represents numbers, letters, and words. And you'll learn to perform arithmetic operations.

You will also start to learn the vocabulary of programming, including terms like operator, expression, value, and type. This vocabulary is important—you will need it to understand the rest of the book, to communicate with other programmers, and to use and understand virtual assistants.

Arithmetic Operators

An **arithmetic operator** is a symbol that represents an arithmetic computation. For example, the plus sign, `+`, performs addition:

```
30 + 12
```

```
42
```

The minus sign, `-`, is the operator that performs subtraction:

```
43 - 1
```

```
42
```

The asterisk, `*`, performs multiplication:

```
6 * 7
```

```
42
```

And the forward slash, `/`, performs division:

```
84 / 2
```

```
42.0
```

Notice that the result of the division is `42.0` rather than `42`. That's because there are two types of numbers in Python:

- **integers**, which represent whole numbers, and
- **floating-point numbers**, which represent numbers with a decimal point.

If you add, subtract, or multiply two integers, the result is an integer. But if you divide two integers, the result is a floating-point number. Python provides another operator, `//`, that performs **integer division**. The result of integer division is always an integer:

```
84 // 2
```

```
42
```

Integer division is also called "floor division" because it always rounds down (toward the "floor"):

```
85 // 2
```

```
42
```

Finally, the operator ** performs exponentiation; that is, it raises a number to a power:

```
7 ** 2
```

```
49
```

In some other languages, the caret, ^, is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result might be unexpected:

```
7 ^ 2
```

```
5
```

I won't cover bitwise operators in this book, but you can read about them at *http://wiki.python.org/moin/BitwiseOperators*.

Expressions

A collection of operators and numbers is called an **expression**. An expression can contain any number of operators and numbers. For example, here's an expression that contains two operators:

```
6 + 6 ** 2
```

```
42
```

Notice that exponentiation happens before addition. Python follows the order of operations you might have learned in a math class: exponentiation happens before multiplication and division, which happen before addition and subtraction.

In the following example, multiplication happens before addition:

```
12 + 5 * 6
```

```
42
```

If you want the addition to happen first, you can use parentheses:

```
(12 + 5) * 6
```

```
102
```

Every expression has a **value**. For example, the expression `6 * 7` has the value `42`.

Arithmetic Functions

In addition to the arithmetic operators, Python provides a few **functions** that work with numbers. For example, the `round` function takes a floating-point number and rounds it off to the nearest whole number:

```
round(42.4)
```

```
42
```

```
round(42.6)
```

```
43
```

The `abs` function computes the absolute value of a number. For a positive number, the absolute value is the number itself:

```
abs(42)
```

```
42
```

For a negative number, the absolute value is positive:

```
abs(-42)
```

```
42
```

When we use a function like this, we say we're **calling** the function. An expression that calls a function is a **function call**.

When you call a function, the parentheses are required. If you leave them out, you get an error message:

```
abs 42
```

```
  Cell In[18], line 1
    abs 42
        ^
SyntaxError: invalid syntax
```

You can ignore the first line of this message; it doesn't contain any information we need to understand right now. The second line is the code that contains the error, with a caret (^) beneath it to indicate where the error was discovered.

The last line indicates that this is a **syntax error**, which means that there is something wrong with the structure of the expression. In this example, the problem is that a function call requires parentheses.

Let's see what happens if you leave out the parentheses *and* the value:

```
abs
```

```
<function abs(x, /)>
```

A function name all by itself is a legal expression that has a value. When it's displayed, the value indicates that `abs` is a function, and it includes some additional information I'll explain later.

Strings

In addition to numbers, Python can also represent sequences of letters, which are called **strings** because the letters are strung together like beads on a necklace. To write a string, we can put a sequence of letters inside straight quotation marks:

```
'Hello'
```

```
'Hello'
```

It is also legal to use double quotation marks:

```
"world"
```

```
'world'
```

Double quotes make it easy to write a string that contains an apostrophe, which is the same symbol as a straight quote:

```
"it's a small "
```

```
"it's a small "
```

Strings can also contain spaces, punctuation, and digits:

```
'Well, '
```

```
'Well, '
```

The + operator works with strings; it joins two strings into a single string, which is called **concatenation**:

```
'Well, ' + "it's a small " + 'world.'
```

```
"Well, it's a small world."
```

The * operator also works with strings; it makes multiple copies of a string and concatenates them:

```
'Spam, ' * 4
```

```
'Spam, Spam, Spam, Spam, '
```

The other arithmetic operators don't work with strings.

Python provides a function called `len` that computes the length of a string:

```
len('Spam')
```

```
4
```

Notice that `len` counts the letters between the quotes, but not the quotes.

When you create a string, be sure to use straight quotes. The backquote, also known as a backtick, causes a syntax error:

```
        `Hello`
```

```
  Cell In[49], line 1
    `Hello`
    ^
SyntaxError: invalid syntax
```

Smart quotes, also known as curly quotes, are also illegal.

Values and Types

So far we've seen three kinds of values:

- `2` is an integer,
- `42.0` is a floating-point number, and
- `'Hello'` is a string.

A kind of value is called a **type**. Every value has a type—or we sometimes say it "belongs to" a type.

Python provides a function called `type` that tells you the type of any value. The type of an integer is `int`:

```
type(2)
```

```
int
```

The type of a floating-point number is `float`:

```
type(42.0)
```

```
float
```

And the type of a string is `str`:

```
type('Hello, World!')
```

```
str
```

The types `int`, `float`, and `str` can be used as functions. For example, `int` can take a floating-point number and convert it to an integer (always rounding down):

```
int(42.9)
```

```
42
```

And `float` can convert an integer to a floating-point value:

```
float(42)
```

```
42.0
```

Now, here's something that can be confusing. What do you get if you put a sequence of digits in quotes?

```
'126'
```

```
'126'
```

It looks like a number, but it is actually a string:

```
type('126')
```

```
str
```

If you try to use it like a number, you might get an error:

```
'126' / 3
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

This example generates a `TypeError`, which means that the values in the expression, which are called **operands**, have the wrong type. The error message indicates that the `/` operator does not support the types of these values, which are `str` and `int`.

If you have a string that contains digits, you can use `int` to convert it to an integer:

```
int('126') / 3
```

```
42.0
```

If you have a string that contains digits and a decimal point, you can use `float` to convert it to a floating-point number:

```
float('12.6')
```

```
12.6
```

When you write a large integer, you might be tempted to use commas between groups of digits, as in `1,000,000`. This is a legal expression in Python, but the result is not an integer:

```
1,000,000
```

```
(1, 0, 0)
```

Python interprets `1,000,000` as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

You can use underscores to make large numbers easier to read:

```
1_000_000
```

```
1000000
```

Formal and Natural Languages

Natural languages are the languages people speak, like English, Spanish, and French. They were not designed by people; they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Similarly, programming languages are formal languages that have been designed to express computations.

Although formal and natural languages have some features in common there are important differences:

Ambiguity
: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any program has exactly one meaning, regardless of context.

Redundancy
: In order to make up for ambiguity and reduce misunderstandings, natural languages use redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

Literalness
: Natural languages are full of idiom and metaphor. Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, sad, or embarrassed.

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and

precision, and particular weaknesses, like lack of empathy and an inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

Glossary

arithmetic operator: A symbol, like `+` and `*`, that denotes an arithmetic operation like addition or multiplication.

integer: A type that represents whole numbers.

floating-point: A type that represents numbers with fractional parts.

integer division: An operator, `//`, that divides two numbers and rounds down to an integer.

expression: A combination of variables, values, and operators.

value: An integer, floating-point number, or string—or one of other kinds of values we will see later.

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call: An expression—or part of an expression—that runs a function. It consists of the function name followed by an argument list in parentheses.

syntax error: An error in a program that makes it impossible to parse—and therefore impossible to run.

string: A type that represents sequences of characters.

concatenation: Joining two strings end to end.

type: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

operand: One of the values on which an operator operates.

natural language: Any of the languages that people speak that evolved naturally.

formal language: Any of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs. All programming languages are formal languages.

bug: An error in a program.

debugging: The process of finding and correcting errors.

Exercises

Ask a Virtual Assistant

As you work through this book, there are several ways you can use a virtual assistant or chatbot to help you learn:

- If you want to learn more about a topic in the chapter, or anything is unclear, you can ask for an explanation.
- If you are having a hard time with any of the exercises, you can ask for help.

In each chapter, I'll suggest exercises you can do with a virtual assistant, but I encourage you to try things on your own and see what works for you.

Here are some topics you could ask a virtual assistant about:

- Earlier I mentioned bitwise operators but I didn't explain why the value of `7 ^ 2` is 5. Try asking "What are the bitwise operators in Python?" or "What is the value of `7 XOR 2`?"
- I also mentioned the order of operations. For more details, ask "What is the order of operations in Python?"
- The `round` function, which we used to round a floating-point number to the nearest whole number, can take a second argument. Try asking "What are the arguments of the round function?" or "How do I round pi off to three decimal places?"
- There's one more arithmetic operator I didn't mention; try asking "What is the modulus operator in Python?"

Most virtual assistants know about Python, so they answer questions like this pretty reliably. But remember that these tools make mistakes. If you get code from a chatbot, test it!

Exercise

You might wonder what `round` does if a number ends in `0.5`. The answer is that it sometimes rounds up and sometimes rounds down. Try these examples and see if you can figure out what rule it follows:

```
round(42.5)
```

```
42
```

```
round(43.5)
```

```
44
```

If you are curious, ask a virtual assistant, "If a number ends in 0.5, does Python round up or down?"

Exercise

When you learn about a new feature, you should try it out and make mistakes on purpose. That way, you learn the error messages, and when you see them again, you will know what they mean. It is better to make mistakes now and deliberately than later and accidentally.

1. You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2++2`?
2. What happens if you have two values with no operator between them, like `4 2`?
3. If you call a function like `round(42.5)`, what happens if you leave out one or both parentheses?

Exercise

Recall that every expression has a value, every value has a type, and we can use the `type` function to find the type of any value.

What is the type of the value of the following expressions? Make your best guess for each one, and then use `type` to find out.

- `765`
- `2.718`
- `'2 pi'`
- `abs(-7)`
- `abs(-7.0)`
- `abs`
- `int`
- `type`

Exercise

The following questions give you a chance to practice writing arithmetic expressions:

1. How many seconds are there in 42 minutes 42 seconds?
2. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.
3. If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace in seconds per mile?
4. What is your average pace in minutes and seconds per mile?
5. What is your average speed in miles per hour?

If you already know about variables, you can use them for this exercise. If you don't, you can do the exercise without them—and then we'll see them in the next chapter.