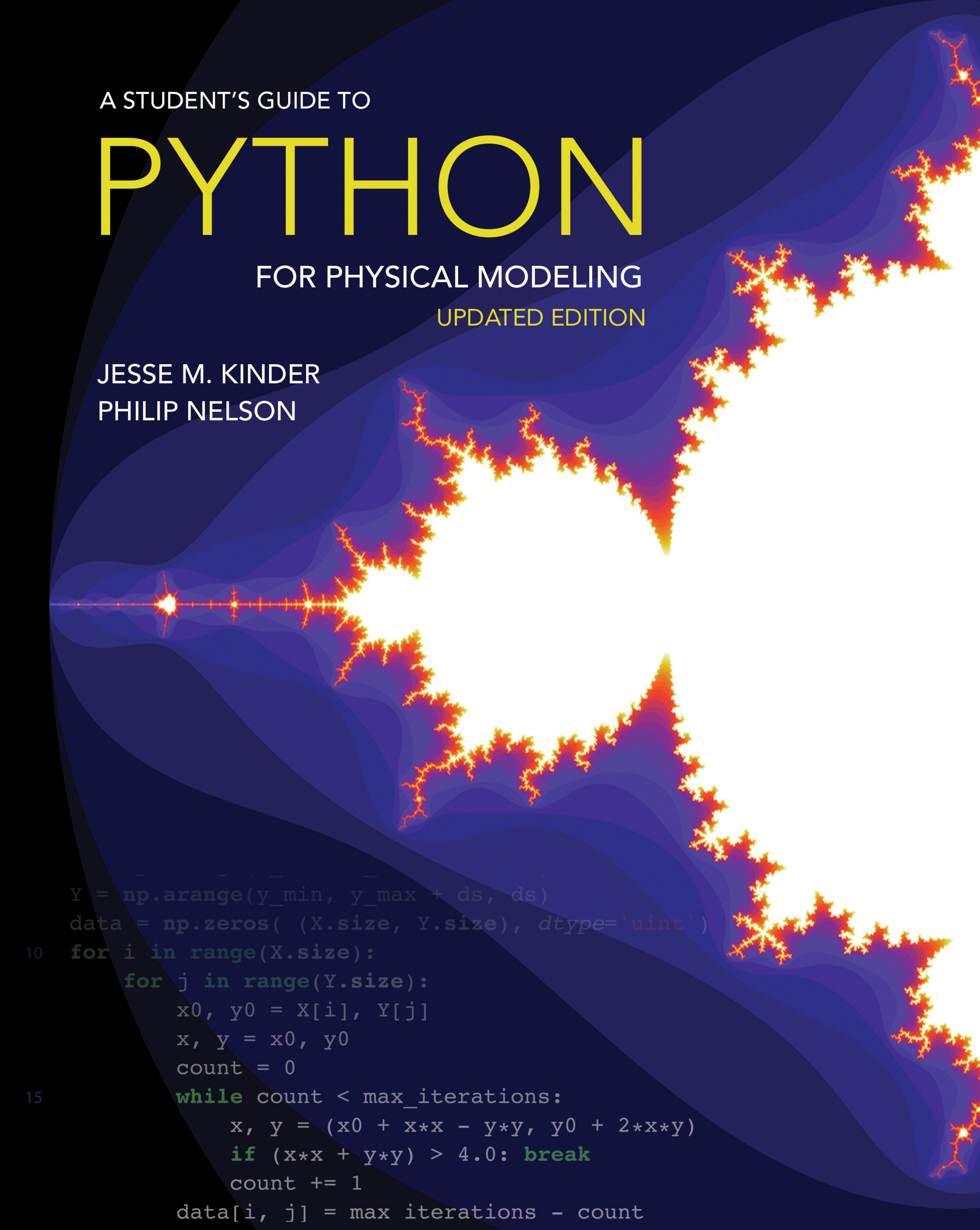
A STUDENT'S GUIDE TO

PYTHON

FOR PHYSICAL MODELING

UPDATED EDITION

JESSE M. KINDER
PHILIP NELSON



A Student's Guide to Python for Physical Modeling

Updated Edition

A Student's Guide to Python for Physical Modeling

Updated Edition

Jesse M. Kinder and Philip Nelson

Princeton University Press
Princeton and Oxford

Published by Princeton University Press, 41 William Street,
Princeton, New Jersey 08540
In the United Kingdom: Princeton University Press, 6 Oxford Street,
Woodstock, Oxfordshire OX20 1TW

press.princeton.edu

Original edition published by Princeton University Press 2015
Updated edition published 2018
Second printing 2019
Library of Congress Control Number: 2017959122

ISBN 978-0-691-18056-4
ISBN (pbk.) 978-0-691-18057-1

British Library Cataloging-in-Publication Data is available.

This book has been composed using the LaTeX typesetting system.

The publisher would like to acknowledge the authors of this volume for providing the camera-ready copy from which this book was printed.

Printed on acid-free paper. ∞

Printed in the United States of America

3 5 7 9 10 8 6 4 2

The front cover shows a Mandelbrot set, an image that you will be able to generate for yourself after you work through this book.

For Oliver Arthur Nelson – PN

Contents

Let's Go

Why teach yourself Python, and why do it this way

Learning to program a computer can change your perspective. At first, it feels like you are struggling along and picking up a couple of neat tricks here and there, but after a while, you start to realize that *you* can make the computer do almost *anything*. You can add in the effects of friction and air resistance that your physics professor is always telling you to ignore, you can make your own predator–prey simulations to study population models, you can create your own fractals, you can look for correlations in the stock market—the list is endless.

In order to communicate with a computer, you must first learn a language it understands. Python is an excellent choice because it is easy to get started and its structure is very natural—at least compared to some other computer languages. Soon, you will find yourself spending most of your time thinking about how to solve a problem, rather than how to explain your calculation to a computer.

Whatever your motivation for learning Python, you may wonder whether it's really necessary to wade through everything in this book. Bear with us. We are working scientists, and we have used our experience to prepare you to start exploring and learning on your own as efficiently as possible. Spend a few hours trying everything we recommend, in the order we recommend it. This will save time in the long run. We have eliminated everything you don't need at the outset. What remains is a set of basic knowledge and skills that you will almost certainly find useful someday.

How to use this tutorial

Here are a few ideas about how you might teach yourself Python using this book.

- Many code samples that appear in this document, as well as errata, updates, data sets, and more are available via `press.princeton.edu/titles/11349.html`.
- After the first few pages, you'll need to work in front of a computer that is running Python. (Don't worry—we'll tell you how to set it up.) On that same computer, you'll probably want to have open a text document named `code_samples.txt`, which is available via the website above.
- Next to the computer you may have a hard copy of this book, or the eBook on a tablet or other device. Alternatively, the eBook can be open on the same computer that runs Python.
- This book will frequently ask you to try things. Some of those things involve snippets of code given in the text. You can copy and paste code from `code_samples.txt` into your Python session, see what happens, then modify and play with it.
- You can also access the snippets interactively. A page with links to individual code samples is available via the website above. In the eBook, you can also click on the words `[get code]` at the top of a code sample to visit the web page. Either way, you can copy and paste code from the web page into Python.
- A few sections and footnotes are flagged with this "Track 2" symbol: $\boxed{T_2}$. These are more advanced and can be skipped on a first reading.

And now . . . Let's go.

CHAPTER 1

Getting Started with Python

The Analytical Engine weaves algebraical patterns, just as the Jacquard loom weaves flowers and leaves.
—Ada, Countess of Lovelace, 1815–1853

1.1 ALGORITHMS AND ALGORITHMIC THINKING

The goal of this tutorial is to get you started in computational science using the computer language Python. Python is open-source software. You can download, install, and use it anywhere. Many good introductions exist, and more are written every year. *This* one is distinguished mainly by the fact that it focuses on skills useful for solving problems in physical modeling.

Modeling a physical system can be a complicated task. Let's take a look at how we can use the powerful processors inside your computer to help.

1.1.1 Algorithmic thinking

Suppose that you need to instruct a friend how to back your car out of your driveway. Your friend has never driven a car, but it's an emergency, and your only communication channel is a phone conversation before the operation begins.

You need to break the required task down into small, explicit steps that your friend understands and can execute in sequence. For example, you might provide your friend the following set of instructions:

```
1 Put the key in the ignition.
2 Turn the key until the car starts, then let go.
3 Push the button on the shift lever and move it to "Reverse."
4 ...
```

Unfortunately, for many cars this "code" won't work, even if your friend understands each instruction: It contains a **bug**. Before step 3, many cars require that the driver

```
Press down the left pedal.
```

Also, the shifter may be marked `R` instead of `Reverse`. It is difficult at first to get used to the high degree of precision required when composing instructions like these.

Because you are giving the instructions in advance (your friend has no mobile phone), it's also wise to allow for contingencies:

```
If a crunching sound is heard, press down on the left pedal ...
```

Breaking the steps of a long operation down into small, explicit substeps and anticipating contingencies are the beginning of *algorithmic thinking*.

If your friend has had a lot of experience watching people drive cars, then the instructions above may be sufficient. But a friend from Mars—or a robot—would need much more detail. For example, the first two steps may need to be expanded to something like

```
Grab the wide end of the key.
Insert the pointed end of the key into the slot on the lower right side
        of the steering column.
Rotate the key about its long axis in the clockwise direction
        (when viewed from the wide end toward the pointed end).
...
```

These two sets of instructions illustrate the difference between low-level and high-level languages for communicating with a computer. A *low-level* computer program is similar to the second set of explicit instructions, written in a language that a machine can understand.[1] A *high-level* system understands many common tasks, and therefore can be programmed in a more condensed style, like the first set of instructions above. Python is a high-level language. It includes commands for common operations in mathematical calculations, processing text, and manipulating files. In addition, Python can access many *standard libraries*, which are collections of programs that perform advanced functions such as data visualization and image processing.

Python also comes with a *command line interpreter*—a program that executes Python commands as you type them. Thus, with Python, you can save instructions in a file and run them later, or you can type commands and execute them immediately. In contrast, many other programming languages used in scientific computing, like C, C++, or FORTRAN, require you to *compile* your programs before you can *execute* them. A separate program called a compiler translates your code into a low-level language. You then run the resulting compiled program to execute (carry out) your algorithm. With Python, it is comparatively easy to quickly write, run, and debug programs. (It still takes patience and practice, though.)

A command line interpreter combined with standard libraries and programs you write yourself provides a convenient and powerful scientific computing platform.

1.1.2 States

You have probably studied multistep mathematical proofs, perhaps long ago in geometry class. The goal of such a narrative is to establish the truth of a desired conclusion by sequentially appealing to given information and a formal system. Thus, each statement's truth, although not evident in isolation, is supposed to be straightforward in light of the preceding statements. The reader's "state" (list of propositions known to be true) changes while reading through the proof. At the end of the proof, there is an unbroken chain of logical deductions that lead from the axioms and assumptions to the result.

An **algorithm** has a different goal. It is a chain of instructions, each of which describes a simple operation, that accomplishes a complex task. The chain may involve a lot of repetition, so you won't want to supervise the execution of every step. Instead, you specify all the steps in advance, then stand back while your electronic assistant performs them rapidly. There may also be contingencies that cannot be known in advance. (`If a crunching sound is heard, ...`)

[1] Machine code and assembly language are low-level programming languages.