

A STUDENT'S GUIDE TO

# PYTHON

FOR PHYSICAL MODELING

UPDATED EDITION

JESSE M. KINDER

PHILIP NELSON



```
Y = np.arange(y_min, y_max + ds, ds)
data = np.zeros( (X.size, Y.size), dtype='uint')
10 for i in range(X.size):
    for j in range(Y.size):
        x0, y0 = X[i], Y[j]
        x, y = x0, y0
        count = 0
15 while count < max_iterations:
            x, y = (x0 + x*x - y*y, y0 + 2*x*y)
            if (x*x + y*y) > 4.0: break
            count += 1
        data[i, j] = max_iterations - count
```

# 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

Copyright © 2018 by Princeton University Press.  
All rights associated with the computer code in this work are retained by Jesse M. Kinder and Philip Nelson.

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`

All Rights Reserved  
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 L<sup>A</sup>T<sub>E</sub>X 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.

Although the authors have sought to provide accurate and up-to-date information, they assume no responsibility for errors or omissions, nor any liability for damages resulting from the use of the information contained within this document. The authors make no claim that suggestions and code samples described herein will work in future versions of Python or its extended environments.

For Oliver Arthur Nelson – PN



# Contents

Let's Go

xiii

1

## Getting Started with Python

1

- 1.1 Algorithms and algorithmic thinking 1
  - 1.1.1 Algorithmic thinking 1
  - 1.1.2 States 2
  - 1.1.3 What does `a = a + 1` mean? 3
  - 1.1.4 Symbolic versus numerical 4
- 1.2 Launch Python 4
  - 1.2.1 IPython console 5
  - 1.2.2 Error messages 9
  - 1.2.3 Sources of help 9
  - 1.2.4 Good practice: Keep a log 11
- 1.3 Python modules 11
  - 1.3.1 `import` 11
  - 1.3.2 `from ... import` 12
  - 1.3.3 NumPy and PyPlot 12
- 1.4 Python expressions 13
  - 1.4.1 Numbers 13
  - 1.4.2 Arithmetic operations and predefined functions 13
  - 1.4.3 Good practice: Variable names 15
  - 1.4.4 More about functions 15



2

## Organizing Data

17

- 2.1 Objects and their methods 17
- 2.2 Lists, tuples, and arrays 19
  - 2.2.1 Creating a list or tuple 19
  - 2.2.2 NumPy arrays 19
  - 2.2.3 Filling an array with values 21
  - 2.2.4 Concatenation of arrays 22
  - 2.2.5 Accessing array elements 23
  - 2.2.6 Arrays and assignments 24
  - 2.2.7 Slicing 24
  - 2.2.8 Flattening an array 26




2.2.9	Reshaping an array	26
2.2.10	 Lists and arrays as indices	26
2.3	Strings	27
2.3.1	Formatting strings with the <code>format()</code> method	29
2.3.2	 Formatting strings with <code>%</code>	30

## 3

## Structure and Control



31


3.1	Loops	31
3.1.1	<code>for</code> loops	31
3.1.2	<code>while</code> loops	33
3.1.3	Very long loops	33
3.1.4	Infinite loops	33
3.2	Array operations	34
3.2.1	Vectorizing math	34
3.2.2	Matrix math	36
3.2.3	Reducing an array	36
3.3	Scripts	37
3.3.1	The Editor	37
3.3.2	 Other editors	38
3.3.3	First steps to debugging	38
3.3.4	Good practice: Commenting	40
3.3.5	Good practice: Using named parameters	43
3.3.6	Good practice: Units	44
3.4	Contingent behavior: Branching	44
3.4.1	The <code>if</code> statement	45
3.4.2	Testing equality of floats	46
3.5	Nesting	47

## 4

## Data In, Results Out

48



4.1	Importing data	48
4.1.1	Obtaining data	49
4.1.2	Bringing data into Python	49
4.2	Exporting data	52
4.2.1	Scripts	52
4.2.2	Data files	52
4.3	Visualizing data	54
4.3.1	The <code>plot</code> command and its relatives	55
4.3.2	Manipulate and embellish	57
4.3.3	 More about figures and their axes	59
4.3.4	 Error bars	60
4.3.5	3D graphs	60
4.3.6	Multiple plots	61
4.3.7	Subplots	62


- 4.3.8 Saving figures 62
- 4.3.9  Using figures in other applications 63

## 5 | First Computer Lab 64

- 5.1 HIV example 64
  - 5.1.1 Explore the model 64
  - 5.1.2 Fit experimental data 65
- 5.2 Bacterial example 66
  - 5.2.1 Explore the model 66
  - 5.2.2 Fit experimental data 66

## 6 | More Python Constructions 68

- 6.1 Writing your own functions 68
  - 6.1.1 Defining functions in Python 69
  - 6.1.2 Updating functions 71
  - 6.1.3 Arguments, keywords, and defaults 71
  - 6.1.4 Return values 72
  - 6.1.5 Functional programming 73
- 6.2 Random numbers and simulation 74
  - 6.2.1 Simulating coin flips 74
  - 6.2.2 Generating trajectories 75
- 6.3 Histograms and bar graphs 76
  - 6.3.1 Creating histograms 76
  - 6.3.2 Finer control 77
- 6.4 Contour plots and surfaces 77
  - 6.4.1 Generating a grid of points 78
  - 6.4.2 Contour plots 78
  - 6.4.3 Surface plots 79
- 6.5 Numerical solution of nonlinear equations 79
  - 6.5.1 General real functions 80
  - 6.5.2 Complex roots of polynomials 81
- 6.6 Solving systems of linear equations 81
- 6.7 Numerical integration 82
  - 6.7.1 Integrating a predefined function 82
  - 6.7.2 Integrating your own function 83
  - 6.7.3 Oscillatory integrands 84
  - 6.7.4  Parameter dependence 84
- 6.8 Numerical solution of differential equations 84
  - 6.8.1 Reformulating the problem 85
  - 6.8.2 Solving an ODE 86
  - 6.8.3  Parameter dependence 87
- 6.9 Vector fields and streamlines 88
  - 6.9.1 Vector fields 88
  - 6.9.2 Streamlines 89

<b>7</b>	<b>Second Computer Lab</b>	<b>91</b>
7.1	Generating and plotting trajectories	91
7.2	Plotting the displacement distribution	91
7.3	Rare events	93
7.3.1	The Poisson distribution	93
7.3.2	Waiting times	94
<b>8</b>	<b>Still More Techniques</b>	<b>96</b>
8.1	Image processing	96
8.1.1	Images as NumPy arrays	96
8.1.2	Saving and displaying images	97
8.1.3	Manipulating images	97
8.2	Displaying Data as an Image	98
8.3	Animation	99
8.3.1	Creating animations	99
8.3.2	Saving animations	100
	HTML movies	100
	 Using an encoder	102
8.4	Analytic calculations	103
8.4.1	The SymPy library	103
8.4.2	Wolfram Alpha	104
<b>9</b>	<b>Third Computer Lab</b>	<b>106</b>
9.1	Convolution	106
9.1.1	Python tools for image processing	107
9.1.2	Averaging	108
9.1.3	Smoothing with a Gaussian	108
9.2	Denoising an image	109
9.3	Emphasizing features	109
	<b>Get Going</b>	<b>111</b>
<b>A</b>	<b>Installing Python</b>	<b>113</b>
A.1	Install Python and Spyder	113
A.1.1	Graphical installation	114
A.1.2	Command line installation	115
A.2	Setting up Spyder	116
A.2.1	Working directory	116
A.2.2	Interactive graphics	117
A.2.3	Script template	117
A.2.4	Restart	118
A.3	Keeping up to date	118
A.4	Installing FFmpeg	118

## A.5 Installing ImageMagick 119

**B | Jupyter Notebooks 120**

- B.1 Getting Started 120
  - B.1.1 Launch Jupyter Notebooks 120
  - B.1.2 Open a notebook 121
  - B.1.3 Multiple notebooks 121
  - B.1.4 Quitting Jupyter 122
  - B.1.5 [\[72\]](#)Setting the default directory 122
- B.2 Cells 123
  - B.2.1 Code cells 123
  - B.2.2 Graphics 124
  - B.2.3 Markdown cells 124
  - B.2.4 Edit mode and command mode 125
- B.3 Sharing 125
- B.4 More details 125
- B.5 Pros and Cons 125

**C | Errors and Error Messages 127**

- C.1 Python errors in general 127
- C.2 Some common errors 128

**D | Python 2 versus Python 3 131**

- D.1 Division 131
- D.2 Print command 131
- D.3 User input 132
- D.4 More assistance 133

**E | Under the Hood 134**

- E.1 Assignment statements 134
- E.2 Memory management 135
- E.3 Functions 135
- E.4 Scope 137
  - E.4.1 Name collisions 138
  - E.4.2 Variables passed as arguments 139
- E.5 Summary 140

**F | Answers to “Your Turn” Questions 141**

## Acknowledgments 145

## References 147

## Index 149



# 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](http://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: . These are more advanced and can be skipped on a first reading.

And now ... Let's go.



## 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.<sup>1</sup> 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, ...)

---

<sup>1</sup> Machine code and assembly language are low-level programming languages.