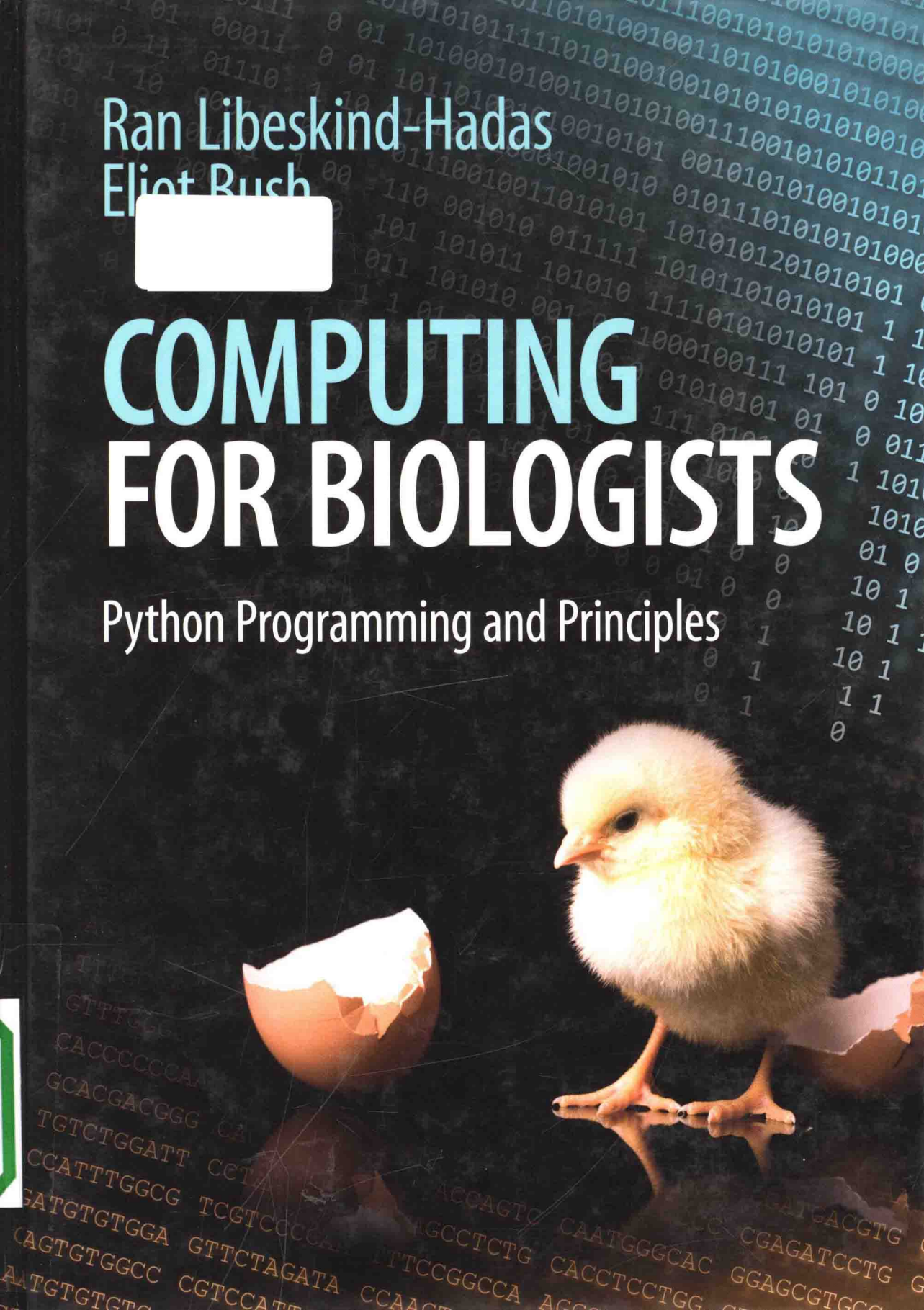
Ran Libeskind-Hadas

Eliot Bush

COMPUTING FOR BIOLOGISTS

Python Programming and Principles



for Biologists

Python Programming and Principles

Ran Libeskind-Hadas

Department of Computer Science,
Harvey Mudd College

Eliot Bush

Department of Biology,
Harvey Mudd College

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107042827

First published 2014

Printed in the United States of America by Sheridan Books, Inc.

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data
Libeskind-Hadas, Ran.
Computing for biologists / Ran Libeskind-Hadas, Department of Computer Science, Harvey Mudd College, Eliot Bush, Department of Biology, Harvey Mudd College.
pages cm
Includes index.
ISBN 978-1-107-04282-7 (Hardback) – ISBN 978-1-107-64218-8 (Paperback)
1. Biology–Data processing. 2. Python (Computer program language) 3. Computer programming.
I. Bush, Eliot Christen. II. Title.
QH324.2.L53 2014
570.285–dc23 2014014322

ISBN 978-1-107-04282-7 Hardback
ISBN 978-1-107-64218-8 Paperback

Additional resources for this publication at www.cambridge.org/c4b

Computing for Biologists

Python Programming and Principles

Computing is revolutionizing the practice of biology. This book, which assumes no prior computing experience, provides students with the tools to write their own Python programs and to understand fundamental concepts in computational biology and bioinformatics.

Each major part of the book begins with a compelling biological question, followed by the algorithmic ideas and programming tools necessary to explore it: the origins of pathogenicity are examined using gene finding, the evolutionary history of sex determination systems is studied using sequence alignment, and the origin of modern humans is addressed using phylogenetic methods. In addition to providing general programming skills, this book explores the design of efficient algorithms, simulation, NP-hardness, and the maximum likelihood method, among other key concepts and methods.

Easy-to-read and designed to equip students with the skills to write programs for solving a range of biological problems, the book is accompanied by numerous programming exercises, available at www.cs.hmc.edu/CFB.

Ran Libeskind-Hadas is the R. Michael Shanahan Professor of Computer Science at Harvey Mudd College, USA, working in the areas of algorithms and computational biology. He is a recipient of both the Iris and Howard Critchell Professorship and the Joseph B. Platt Professorship for teaching, as well as the Distinguished Alumni Educator Award from the University of Illinois Urbana-Champaign Department of Computer Science.

Eliot Bush is Associate Professor of Biology at Harvey Mudd College, USA. His main research interest is the study of evolution. Among other things he has modeled the evolution of metabolism, characterized DNA methylation patterns in insects, developed algorithms for studying substitution bias in DNA, and analyzed a 30 million-year-old primate fossil. His teaching interests focus on incorporating computers and programming assignments into biology coursework.

Computing

PREFACE

What's this book about?

The computer is the most powerful general-purpose tool available to biologists.

In part, this is due to the continuing rapid growth of biological data. For example, at the time of writing, the GenBank database had over 100 *million* genetic sequences with over 100 *billion* DNA characters. Among the contents of that database are genes from many organisms, annotated with what's known about their function.

Imagine that you're studying a bacterium and wish to understand what causes it to be infectious. One promising approach is to identify genes in the bacterium and compare these to known genes in GenBank. If you're able to find similar genes whose function is known, it will tell you a great deal about the role of the genes in your bacterium. This approach represents a computational challenge, and is, in fact, the topic of Part I of this book.

But searching enormous databases is not the only reason that computers are so useful to biologists. Many biological problems have a large number of different possible solutions and only a computer – programmed with carefully designed computational recipes or "algorithms" – has any chance of finding the right one. For example, biological molecules such as proteins and RNA fold into complex shapes that strongly impact their function. Computational techniques have been developed to predict how these molecules fold. Such techniques help us understand how proteins and RNA work and can even help us design new molecules to treat disease.

Simply put, computing is revolutionizing the practice of biology.

In order to fully appreciate and exploit the power of computation, biologists must be trained to "think" computationally. In practical terms, this means understanding fundamental computing concepts that recur in many applications *and* being able to write programs.

Why? Consider, for example, that there are well over 400 different software packages for phylogenetics (the study of the evolutionary relationships among organisms). An increasing level of computational sophistication is needed to select the appropriate software for a given application, use it correctly, and understand its

abilities and limitations. This is true not just of phylogenetics, but also for many other areas of biology.

But using existing software will not be enough. The number and variety of computational problems that arise in biology are rapidly outpacing the functionality of software tools. Eventually, most biologists are likely to encounter problems that cannot be solved with existing software. Therefore, it is imperative for biologists to have the ability to write their own programs.

This book seeks to provide biology students with both an exposure to major computational ideas *and* practical programming skills. It requires no specific biology or computer science background. It is designed as a first-year college-level course and has been taught to that audience at Harvey Mudd College since 2009. The authors hereby acknowledge the generous grant support of HHMI for the development of that course.

In contrast to a typical introductory bioinformatics book, this book emphasizes programming over the use of existing software. By the time you've completed this book, and the online homework problems, you should feel comfortable writing programs for a wide array of applications in biology and beyond. You'll also have an understanding of computational ideas like "heuristics," "memoization" and "dynamic programming," "NP-completeness," and others, that will allow you to understand and compare the technical aspects of existing bioinformatics tools.

This book begins with Chapter 0, which offers a first introduction to the Python programming language. The rest of the book is organized into four parts, each comprising several chapters. Each part begins with a "large" biological question and the chapters in that part provide the computational and programming tools to answer that question.

At the end of each chapter – and sometimes even within a chapter – you'll see a question icon pointing you to one or more recommended problems. You'll find those problems at the url:

`www.cs.hmc.edu/CFB`

Let's get started!

CONTENTS

Meet Python

0

One way that you can spot a computer scientist is that they begin counting from 0 rather than from 1. So this is Chapter 0. But it's also Chapter 0 to signify that it's a warm-up chapter to get you on the path to feeling comfortable with Python, the programming language that we'll be using in this book. Every subsequent chapter will begin with an application in biology followed by the computer science ideas that we'll need to solve that problem.

Python is a programming language that, according to its designers, aims to combine "remarkable power with very clear syntax." Indeed, Python programs tend to be relatively short and easy to read. Perhaps for this reason, Python is growing rapidly in popularity among computer scientists, biologists, and others.

The best way to learn to program is to experiment! Therefore, **we strongly urge you to pause frequently as you read this book and try some of the things that we're doing here (and experiment with variations) in Python.** It will make the reading more fun and meaningful.

This chapter includes a number of short exercises that we encourage you to try. Subsequent chapters have links to end-of-chapter programming problems.

The link below offers instructions on how to install and run Python on your computer.

`www.cs.hmc.edu/CFB/Setup`

If you're reading this book as a part of a course, your instructor may have some additional or different instructions.

0.1 Getting Started

When you start up Python, you'll see a "prompt" that looks like this:

```
>>>
```

You can type commands at the prompt and then press the "Return" (or "Enter") key and Python will interpret what we've typed.

For example, below we've typed `3 + 5` (throughout this book, the user's input is always shown in black and Python's response is always shown in blue).

```
>>> 3 + 5
8
```

Next, we can do fancier things like this:

```
>>> (3 + 5) * 2 - 1
15
```

Notice that parentheses were used here to control the order of operations. Normally, multiplication and division have higher *precedence* than addition and subtraction, meaning that Python does multiplications and divisions first and addition and subtractions afterwards. So without parentheses we would have gotten...

```
>>> 3 + 5 * 2 - 1
12
```

You can always use parentheses to specify the order of operations that you desire.

Here are a few more examples of arithmetic in Python:

```
>>> 6 / 2
3
>>> 2 ** 5
32
>>> 10 ** 3
1000
>>> 52 % 10
2
```

You may have inferred what `/`, `**`, and `%` do. Arithmetic symbols like `+`, `-`, `/`, `*`, `**`, and `%` are called *operators*. The `%` operator is pronounced "mod" and it gives the remainder that results when the first number is divided by the second one.

0.2 Big Numbers

Python is not intimidated by big numbers. For example:

```
>>> 2 ** 100
1267650600228229401496703205376L
```

The "L" at the end of that number stands for "Long." It's just Python's way of telling you that the number is big. Look at this:

```
>>> (2 ** 100) % 3
1L
```

The number 1 is *not* long, but `2 ** 100` was long; once Python sees a mathematical expression with a long number in it, it stays in the long number state of mind. You don't have to worry about that – we just note this so you won't say "Huh!?" when you see the "L."

0.3 Strange Division

If you're using Python version 3, division works as you'd expect.

```
>>> 5 / 2
2.5
```

If you're using Python version 2 (e.g., version 2.7), the way it works may surprise you. Take a look at this:

```
>>> 5 / 2
2
```

We've consulted with expert mathematicians and they've verified that 5 divided by 2 is not 2! What's wrong with Python!? The answer is that Python (version 2) is doing "integer division". It assumes that since 5 and 2 are integers, you are only interested in integers. So when it divides 5 by 2, it rounds down to the nearest integer, which is 2. If you wanted Python to do "decimal division", you could do this:

```
>>> 5.0 / 2.0
2.5
```

Since we typed 5.0 and 2.0, Python realized that we are interested in numbers with decimal points, not just integers, so it gave us the answer with a decimal point. In fact, if just ONE of the 5 or 2 has a decimal point after it, Python will get the message that we are thinking about decimal numbers. So we could do any of these:

```
>>> 5.0 / 2
2.5
>>> 5 / 2.0
2.5
>>> 5. / 2
2.5
>>> 5 / 2.
2.5
```

0.4 Naming Things

Python lets you give names to values. This is very useful because it allows you to compute a value, give it a name, and then use it by name in the future. Here's an example:

```
>>> myNumber = 42
>>> myNumber * (10 ** 2)
4200
```

In the first line, we defined `myNumber` to be 42. In the second line, we used that value as part of a computation. The name `myNumber` is called a *variable* in computer science. It's simply a name that we've made up and we can assign values to it.

Notice that the "=" sign is used to make an assignment. On the left of the equal sign is the name of the variable. On the right of the equal sign is an expression that Python evaluates, assigning the resulting value to the variable. For example, we can do something like this:

```
>>> pi = 3.1415926
>>> area = pi * (10 ** 2)
>>> area
314.15926
```

In this case, we define the value of `pi` in the first line. In the second line, the expression `pi * (10 ** 2)` is evaluated (its value is 314.15926) and that value is assigned to another variable called `area`. Finally, when we type `area`, Python displays the value. Notice, also, that the parentheses aren't actually necessary here. However, we used them just to help remind us which operations will be done first. It's often a good idea to do things like this to make your code more readable to other humans.

One last thing before we move on. In mathematics, the expression `pi = 3.1415926` is equivalent to the expression `3.1415926 = pi`. In Python, and in almost all programming languages, *these are not the same*! In fact, while Python understands `pi = 3.1415926` it will complain loudly if you type `3.1415926 = pi`. (Try it in Python to see what it looks like when Python complains.) Didn't Python go to elementary school? The answer is that programming languages view the = symbol in a special way. They assume that what's on the left-hand side is the name of a variable (`pi` in our example). On the right-hand side of = is an expression that can be evaluated. That expression is evaluated and its value is then associated with the variable.

So, when Python sees...

```
>>> pi = 3.1415926
```

... it evaluates the expression on the right. That's easy, there's nothing to evaluate – it's 3.1415926. Python then assigns that value to the variable `pi` on the left. Now, when Python sees...

```
>>> area = pi * (10 ** 2)
```

... it evaluates the expression on the right. That's not too hard either; we've already defined `pi` to be 3.1415926 and Python does the math and evaluates the expression on the right to be 314.15926. It then associates that value with the variable named `area`.

Sometimes, you'll see (and write) things like this:

```
>>> counter = 0
>>> counter = counter + 1
```

In the first line, we've set a variable named `counter` to 0. The second line makes no sense from a mathematical perspective, but it makes complete sense based on

what we now know about Python. Python begins by evaluating the expression on the right side of the = sign. Since `counter` is currently 0, that expression `counter + 1` evaluates to 1. Only once this evaluation is done, Python sets the variable on the left side of the = sign to be this value. So now `counter` is set to 1. This is how we'll often count the number of events of some sort – like the number of occurrences of a particular nucleotide in a DNA sequence or the number of species with a particular property.

0.5 What's in a Name?

Variable names are pretty much up to you. We didn't have to use the names `myNumber`, `pi`, and `area` in our examples above. We could have called them `Joe`, `Sally`, and `Melissa` if we preferred. Variable names must begin with a letter and there are certain symbols that aren't permitted in a variable name. For example, naming a variable `Joe` or `Joe42` is fine, but naming it `Joe+Sally` is not permitted. You can probably imagine why. If Python sees `Joe+Sally` it will think that you are trying to add the values of two variables `Joe` and `Sally`. Similarly, there are a few built-in Python "special words" that can't be used as a variable name. If you try to use them, Python will give you an *error message*. (An error message is the technical term for a complaint.)

Finally, it's a good practice to use descriptive variable names to help the reader understand your program. For example, if a variable is going to store the area of a circle, calling that variable `area` is a much better choice than calling it something like `z` or `y42b` or `harriet`.

Variables provide a convenient way to store values and the names that we give to variables should be simple and descriptive.

0.6 From Numbers to Strings...

That was all good, but numbers are not the only useful kind of data. Python allows us to define strings as well. A *string is* any sequence of symbols within either single or double quotation marks. Here are some examples:

```
>>> name1 = "Ben"
>>> name2 = 'Jerry'
```