LEARNING MADE EASY

Functional Programming

for dummies®
A Wiley Brand

Work faster with functional programming

Understand the pure and impure approach

Perform common tasks in both Python® and Haskell

Functional Programming

by John Paul Mueller

Functional Programming For Dummies®

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, `www.wiley.com`

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit `https://hub.wiley.com/community/support/dummies`.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at `http://booksupport.wiley.com`. For more information about Wiley products, visit `www.wiley.com`.

Library of Congress Control Number: 2018965285

ISBN: 978-1-119-52750-3

ISBN 978-1-119-52751-0 (ebk); ISBN ePDF 978-1-119-52749-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Table of Contents

INTRODUCTION 1
About This Book 1
Foolish Assumptions 3
Icons Used in This Book 3
Beyond the Book 4
Where to Go from Here 5

PART 1: GETTING STARTED WITH FUNCTIONAL PROGRAMMING 7

CHAPTER 1: **Introducing Functional Programming** 9
Defining Functional Programming 10
Understanding its goals 11
Using the pure approach 11
Using the impure approach 12
Considering Other Programming Paradigms 13
Imperative...13
Procedural 13
Object-oriented 14

Declarative 14
Using Functional Programming to Perform Tasks 15
Discovering Languages That Support Functional Programming 16
Considering the pure languages 16
Considering the impure languages 17
Finding Functional Programming Online 17

CHAPTER 2: **Getting and Using Python** . . . .
19 Working with Python in This Book . . . .
20
Creating better code 20
Debugging functionality 20
Defining why notebooks are useful 21
Obtaining Your Copy of Anaconda 21
Obtaining Analytics Anaconda 21
Installing Anaconda on Linux 22
Installing Anaconda on MacOS 23
Installing Anaconda on Windows 24
Understanding the Anaconda package 26
Downloading the Datasets and Example Code 27
Using Jupyter Notebook 28
Defining the code repository 28
Getting and using datasets 33

Creating a Python Application 34
Understanding cells.......................................35 Adding documentation cells . . . . 36 Other cell content 38
Running the Python Application 38
Understanding the Use of Indentation 39
Adding Comments 41
Understanding comments 41
Using comments to leave yourself reminders 43
Using comments to keep code from executing 43
Closing Jupyter Notebook......................................44 Getting Help with the Python Language 45

CHAPTER 3: **Getting and Using Haskell** . . . .
47 Working with Haskell in This Book . . . .
48 Obtaining and Installing Haskell . . . .
48
Installing Haskell on a Linux system 50
Installing Haskell on a Mac system 50
Installing Haskell on a Windows system 52
Testing the Haskell Installation 54
Compiling a Haskell Application 56

Using Haskell Libraries 59
Getting Help with the Haskell Language 60

PART 2: STARTING FUNCTIONAL PROGRAMMING TASKS.. 63 CHAPTER 4:

Defining the Functional Difference 65
Comparing Declarations to Procedures 66
Understanding How Data Works...............................67 Working with immutable data . . . . 68 Considering the role of state . . . . 68 Eliminating side effects . . . . 69 Seeing a Function in Haskell . . . . 69 Using non-curried functions . . . . 69 Using curried functions . . . . 70 Seeing a Function in Python...................................73 Creating and using a Python function . . . . 73 Passing by reference versus by value 74

CHAPTER 5: Understanding the Role of Lambda Calculus . . . . 77 Considering the Origins of Lambda Calculus . . . . 78 Understanding the Rules 80
Working with variables 80
Using application 81
Using abstraction 82

Performing Reduction Operations 85
Considering α-conversion 85
Considering β-reduction 86
Considering η-conversion 88
Creating Lambda Functions in Haskell 89
Creating Lambda Functions in Python 89

CHAPTER 6: Working with Lists and Strings . . . . 91 Defining List Uses . . . . 92 Creating Lists 93
Using Haskell to create Lists 94
Using Python to create lists 95
Evaluating Lists 96
Using Haskell to evaluate Lists 97
Using Python to evaluate lists 99
Performing Common List Manipulations 100
Understanding the list manipulation functions 101
Using Haskell to manipulate lists 101
Using Python to manipulate lists 102
Understanding the Dictionary and Set Alternatives 103
Using dictionaries 103

Using sets 104
Considering the Use of Strings 105
Understanding the uses for strings 105
Performing string-related tasks in Haskell 106
Performing string-related tasks in Python 106

PART 3: MAKING FUNCTIONAL PROGRAMMING PRACTICAL . . . .

109 CHAPTER 7: **Performing Pattern Matching** . . . .
. 111 Looking for Patterns in Data 112
Understanding Regular Expressions . . . . 113 Defining
special characters using escapes . . . . 114 Defining wildcard
characters . . . . 115 Working with anchors . . . .
. . . . 115 Delineating subexpressions using grouping
constructs . . . . 116 Using Pattern Matching in Analysis . . . .
. . . . 117 Working with Pattern Matching in Haskell . . . .
118 Performing simple Posix matches . . . . 118 Matching
a telephone number with Haskell . . . . 120 Working with Pattern
Matching in Python . . . . 121 Performing simple Python
matches . . . . 121 Doing more than matching . . . .
. . . . 123 Matching a telephone number with Python . . . .
. . . . 124

CHAPTER 8: **Using Recursive Functions** . . . .
125 Performing Tasks More than Once 126
Defining the need for repetition . . . . 126 Using
recursion instead of looping . . . . 127 Understanding
Recursion....................................128 Considering basic recursion . . . .
. . . . 129 Performing tasks using lists . . . .
. . . . 131 Upgrading to set and dictionary 132
Considering the use of collections . . . . 134 Using
Recursion on Lists . . . . 135 Working with
Haskell . . . . 135 Working with Python . . . . . .
. . . . 136 Passing Functions Instead of Variables . .
. . . . 137 Understanding when you need a function . . . .
. . . . 138 Passing functions in Haskell . . . .
138 Passing functions in Python . . . . 139 Defining
Common Recursion Errors . . . . 140 Forgetting an
ending . . . . 140 Passing data incorrectly . .
. . . . 141 Defining a correct base instruction . . . . .
. . . . 141

CHAPTER 9: **Advancing with Higher-Order Functions** . . . .
143 Considering Types of Data Manipulation . . . .
144 Performing Slicing and Dicing . . . .
146

Keeping datasets controlled 146
Focusing on specific data 147
Slicing and dicing with Haskell 147
Slicing and dicing with Python 150
Mapping Your Data 151
Understanding the purpose of mapping 151
Performing mapping tasks with Haskell 152
Performing mapping tasks with Python 153
Filtering Data 154
Understanding the purpose of filtering 154
Using Haskell to filter data 155
Using Python to filter data 156
Organizing Data 157
Considering the types of organization 157
Sorting data with Haskell 158
Sorting data with Python 159

CHAPTER 10: **Dealing with Types** . . . .
161 Developing Basic Types 162
Understanding the functional perception of type . . . . 162 Considering
the type signature . . . . 162 Creating types . . . .
. . . . 164

Composing Types 170
Understanding monoids 170
Considering the use of Nothing, Maybe, and Just..............174
Understanding semigroups 176
Parameterizing Types 176
Dealing with Missing Data 178
Handling nulls 178
Performing data replacement 180
Considering statistical measures 180
Creating and Using Type Classes 181

PART 4: INTERACTING IN VARIOUS WAYS . . . .
183 CHAPTER 11: **Performing Basic I/O** . . . .
. . 185 Understanding the Essentials of I/O 186
Understanding I/O side effects . . . . 186 Using
monads for I/O . . . . 188 Interacting with the
user . . . . 188 Working with devices . . . .
. . . . 189 Manipulating I/O Data . . . .
. . . . 191 Using the Jupyter Notebook Magic Functions . . . .
. . . . 192 Receiving and Sending I/O with Haskell . . . .
195 Using monad sequencing 195
Employing monad functions 195

CHAPTER 12: **Handling the Command Line** . . . .

197 Getting Input from the Command Line 198
Automating the command line . . . . 198 Considering the use of prompts . . . . 198 Using the command line effectively . . . . 199 Accessing the Command Line in Haskell . . . . 200 Using the Haskell environment directly . . . . 200 Making sense of the variety of packages . . . . 201 Obtaining CmdArgs 202
Getting a simple command line in Haskell . . . . 204 Accessing the Command Line in Python . . . . 205 Using the Python environment directly . . . . 205 Interacting with Argparse 206

CHAPTER 13: **Dealing with Files** . . . .
207 Understanding How Local Files are Stored . . . .
208 Ensuring Access to Files . . . .
209 Interacting with Files . . . .
209
Creating new files 210
Opening existing files 211

Manipulating File Content 212
Considering CRUD 213
Reading data 214
Updating data 215
Completing File-related Tasks 217

CHAPTER 14: **Working with Binary Data** . . . .
219 Comparing Binary to Textual Data . . . .
220 Using Binary Data in Data Analysis . . . .
221 Understanding the Binary Data Format . . . .
222 Working with Binary Data.....................................225 Interacting with Binary Data in Haskell 225
Writing binary data using Haskell 226
Reading binary data using Haskell 227
Interacting with Binary Data in Python 228
Writing binary data using Python 228
Reading binary data using Python 229

CHAPTER 15: **Dealing with Common Datasets** . . . .
231 Understanding the Need for Standard Datasets . . . .
232 Finding the Right Dataset . . . .
233
Locating general dataset information 233
Using library-specific datasets 234
Loading a Dataset 236
Working with toy datasets 237
Creating custom data 238

Fetching common datasets 239
Manipulating Dataset Entries 241
Determining the dataset content 241
Creating a DataFrame 243
Accessing specific records 244

PART 5: PERFORMING SIMPLE ERROR TRAPPING . . . .

247 CHAPTER 16: **Handling Errors in Haskell . . . .**
. 249 Defining a Bug in Haskell 250
Considering recursion 250
Understanding laziness . . . . 251 Using unsafe
functions . . . . 252 Considering
implementation-specific issues . . . . 253 Understanding the
Haskell-Related Errors . . . . 253 Fixing Haskell Errors Quickly .
. . . . 256 Relying on standard debugging . . . .
. . . . 256 Understanding errors versus exceptions . . . .
. . . . 258

CHAPTER 17: **Handling Errors in Python . . . .**
259 Defining a Bug in Python 260
Considering the sources of errors . . . . 260 Considering
version differences . . . . 262 Understanding the
Python-Related Errors . . . . 263 Dealing with late binding
closures . . . . 263 Using a variable . . . .
. . . . 264 Working with third-party libraries . . . .
. . . . 264 Fixing Python Errors Quickly . . . .
265 Understanding the built-in exceptions 265
Obtaining a list of exception arguments . . . . 266 Considering
functional style exception handling 267

PART 6: THE PART OF TENS . . . .

269 CHAPTER 18: **Ten Must-Have Haskell Libraries . . . .**
. 271 binary 271
Hascore . . . . 273 vect . . . . .
. . . . 273 vector . . . .
. . . . 274 aeson . . . .
. . . . 274 attoparsec . . . .
. . . . 275 bytestring . . . .
. . . . 275 stringsearch . . . .
. . . 276 text 276
moo 277

CHAPTER 19: **Ten (Plus) Must-Have Python Packages . . . .**
279 Gensim . . . .

280 PyAudio . . . .
281 PyQtGraph . . . .
282 TkInter . . . .
283 PrettyTable . . . .
283 SQLAlchemy . . . .
284 Toolz . . . .
284 Cloudera Oryx . . . .
285 funcy . . . .
285 SciPy . . . .
286 XGBoost . . . .
287

CHAPTER 20: **Ten Occupation Areas that Use Functional Programming** . . . .
289 Starting with Traditional Development . . . .
289 Going with New Development . . . .
290 Creating Your Own Development . . . .
291 Finding a Forward-Thinking Business . . . .
292 Doing Something Really Interesting . . . .
292 Developing Deep Learning Applications . . . .
293 Writing Low-Level Code . . . .
293 Helping Others in the Health Care Arena . . . .
294 Working as a Data Scientist . . . .
294 Researching the Next Big Thing . . . .
295

INDEX 297

Introduction

The *functional programming paradigm* is a framework that expresses a partic ular set of assumptions, relies on particular ways of thinking through prob lems, and uses particular methodologies to solve those problems. Some people view this paradigm as being akin to performing mental gymnastics. Other people see functional programming as the most logical and easiest method for coding any particular problem ever invented. Where you appear in this rather broad range of perspectives depends partly on your programming background, partly on the manner in which you think through problems, and partly on the problem you're trying to solve.

Functional Programming For Dummies doesn't try to tell you that the functional programming paradigm will solve every problem, but it does help you understand that functional programming can solve a great many problems with fewer errors, less code, and a reduction in development time. Most important, it helps you understand the difference in the thought process that using the functional pro gramming paradigm involves. Of course, the key is knowing when functional pro gramming is the best option, and that's what you take away from this book. Not only do you see how to perform functional programming with both pure (Haskell) and impure (Python) languages, but you also gain insights into when functional programming is the best solution.

About This Book

Functional Programming For Dummies begins by describing what a paradigm is and how the functional programming paradigm differs. Many developers today don't really understand that different paradigms can truly change the manner in which you view a problem domain, thereby making some problem domains consider ably easier to deal with. As part of considering the functional programming par adigm, you install two languages: Haskell (a pure functional language) and Python (an impure functional language). Of course, part of this process is to see how pure and impure languages differ and determine the advantages and disad vantages of each.

Part of working in the functional programming environment is to understand and use lambda calculus, which is part of the basis on which functional programming it built. Imagine that you're in a room with some of the luminaries of computer science and they're trying to decide how best to solve problems in computer science at a time when the term *computer science* doesn't even exist. For that mat ter, no one has even defined what it means to compute. Even though functional programming might seem new to many people, it's based on real science created by the best minds the world has ever seen to address particularly difficult prob lems. This science uses lambda calculus as a basis, so an explanation of this par ticularly difficult topic is essential.

After you understand the basis of the functional programming paradigm and have installed tools that you can use to see it work, it's time to create some example code. This book starts with some relatively simple examples that you might find in other books that use other programming paradigms so that you compare them and see how functional programming actually differs. You then move on to other sorts of programming problems that begin to emphasize the benefits of functional programming in a stronger way. To make absorbing the concepts of functional programming even easier, this book uses the following conventions:

- Text that you're meant to type just as it appears in the book is **bold**. The exception is when you're working through a step list: Because each step is bold, the text to type is not bold.
- Because functional programming will likely seem strange to many of you, I've made a special effort to define terms, even some of those that you might already know, because they may have a different meaning in the functional realm. You see the terms in italics, followed by their definition.
- When you see words in *italics* as part of a typing sequence, you need to replace that value with something that works for you. For example, if you see "Type ***Your Name*** and press Enter," you need to replace *Your Name* with your actual name.
- Web addresses and programming code appear in `monofont`. If you're reading a digital version of this book on a device connected to the Internet, note that you can click the web address to visit that website, like this: `www.dummies.com`.
- When you need to type command sequences, you see them separated by a special arrow, like this: File ➪ New File. In this case, you go to the File menu first and then select the New File entry on that menu. The result is that you see a new file created.

Foolish Assumptions

You might find it difficult to believe that I've assumed anything about you — after all, I haven't even met you yet! Although most assumptions are indeed foolish, I made these assumptions to provide a starting point for the book.

You need to be familiar with the platform that you want to use because the book doesn't provide any guidance in this regard. To give you maximum information about the functional programming paradigm, this book doesn't discuss any platform-specific issues. You need to know how to install applications, use applications, and generally work with your chosen platform before you begin working with this book. Chapter 2 does show how to install Python, and Chapter 3 shows how to install Haskell. Part 2 of the book gives you the essential introduc tion to functional programming, and you really need to read it thoroughly to obtain the maximum benefit from this book.

This book also assumes that you can find things on the Internet. Sprinkled throughout are numerous references to online material that will enhance your learning experience. However, these added sources are useful only if you actually find and use them.

Icons Used in This Book

As you read this book, you see icons in the margins that indicate material of inter est (or not, as the case may be). This section briefly describes each icon in this book.

Tips are nice because they help you save time or perform some task without a lot of extra work. The tips in this book are time-saving techniques or pointers to resources that you should try in order to get the maximum benefit from Python, Haskell, or the functional programming paradigm.

I don't want to sound like an angry parent or some kind of maniac, but you should avoid doing anything marked with a Warning icon. Otherwise, you could find that your program serves only to confuse users, who will then refuse to work with it.

Whenever you see this icon, think advanced tip or technique. You might find these tidbits of useful information just too boring for words, or they could contain the solution that you need to get a program running. Skip these bits of information whenever you like.

If you don't get anything else out of a particular chapter or section, remember the material marked by this icon. This text usually contains an essential process or a bit of information that you must know to write Python, Haskell, or functional programming applications successfully.

Beyond the Book

This book isn't the end of your functional programming experience — it's really just the beginning. I provide online content to make this book more flexible and better able to meet your needs. That way, as I receive email from you, I can do things like address questions and tell you how updates to Python, its associated packages, Haskell, it's associated libraries, or changes to functional programming techniques that affect book content. In fact, you gain access to all these cool additions:

- **Cheat sheet:** You remember using crib notes in school to make a better mark on a test, don't you? You do? Well, a cheat sheet is sort of like that. It provides you with some special notes about tasks that you can do with Python or Haskell that not every other developer knows. In addition, you find some quick notes about functional programming paradigm differences. You can find the cheat sheet for this book by going to `www.dummies.com` and searching this book's title. Scroll down the page until you find a link to the Cheat Sheet.
- **Updates:** Sometimes changes happen. For example, I might not have seen an upcoming change when I looked into my crystal ball during the writing of this book. In the past, that simply meant the book would become outdated and less useful, but you can now find updates to the book by searching this book's title at `www.dummies.com`.

  In addition to these updates, check out the blog posts with answers to reader questions and demonstrations of useful book-related techniques at `http://blog.johnmuellerbooks.com/`.
- **Companion files:** Hey! Who really wants to type all the code in the book? Most readers would prefer to spend their time actually working through coding examples, rather than typing. Fortunately for you, the source code is available for download, so all you need to do is read the book to learn functional program ming techniques. Each of the book examples even tells you precisely which example project to use. You can find these files at `www.dummies.com`. Click More about This Book and, on the page that appears, scroll down the page to the set of tabs. Click the Downloads tab to find the downloadable example files.

Where to Go from Here

It's time to start your functional programming paradigm adventure! If you're a complete functional programming novice, you should start with Chapter 1 and progress through the book at a pace that allows you to absorb as much of the material as possible.

If you're a novice who's in an absolute rush to get going with functional program ming techniques as quickly as possible, you can skip to Chapter 2, followed by Chapter 3, with the understanding that you may find some topics a bit confusing later. You must install both Python and Haskell to have any hope of getting some thing useful out of this book, so unless you have both languages installed, skip ping these two chapters will likely mean considerable problems later.

Readers who have some exposure to functional programming and already have both Python and Haskell installed can skip to Part 2 of the book. Even with some functional programming experience, Chapter 5 is a must-read chapter because it provides the basis for all other discussions in the book. The best idea is to at least skim all of Part 2.

If you're absolutely certain that you understand both functional programming paradigm basics and how lambda calculus fits into the picture, you can skip to Part 3 with the understanding that you may not see the relevance of some exam ples. The examples build on each other so that you gain a full appreciation of what makes the functional programming paradigm different, so try not to skip any of the examples, even if they seem somewhat simplistic.

Gettin

g Started

with Functional Programming

Discover the functional programming

paradigm. Understand how functional

programming differs. Obtain and install

Python.

Obtain and install Haskell.

» Exploring functional programming »

Programming in the functional way

» Finding a language that suits your needs

» Locating functional programming resources

Chapter 1

Introducing Functional Programming

This book isn't about a specific programming language; it's about a pro gramming paradigm. A *paradigm* is a framework that expresses a particular set of assumptions, relies on particular ways of thinking through problems, and uses particular methodologies to solve those problems. Consequently, this programming book is different because it doesn't tell you which language to use; instead, it focuses on the problems you need to solve. The first part of this chapter discusses how the functional programming paradigm accomplishes this task, and the second part points out how functional programming differs from other para digms you may have used.

The math orientation of functional programming means that you might not create an application using it; you might instead solve straightforward math problems or devise *what if* scenarios to test. Because functional programming is unique in its approach to solving problems, you might wonder how it actually accomplishes its goals. The third section of this chapter provides a brief overview of how you use the functional programming paradigm to perform various kinds of tasks (includ ing traditional development), and the fourth section tells how some languages follow a pure path to this goal and others follow an impure path. That's not to say that those following the pure path are any more perfect than those following the impure path; they're simply different.

Finally, this chapter also discusses a few online resources that you see mentioned in other areas of the book. The functional programming paradigm is popular for solving certain kinds of problems. These resources help you discover the specifics of how people are using functional programming and why they feel that it's such an important method of working through problems. More important, you'll dis cover that many of the people who rely on the functional programming paradigm aren't actually developers. So, if you aren't a developer, you may find that you're already in good company by choosing this paradigm to meet your needs.

Defining Functional Programming

Functional programming has somewhat different goals and approaches than other paradigms use. Goals define what the functional programming paradigm is trying to do in forging the approaches used by languages that support it. However, the goals don't specify a particular implementation; doing that is within the pur view of the individual languages.

The main difference between the functional programming paradigm and other paradigms is that functional programs use math functions rather than statements to express ideas. This difference means that rather than write a precise set of steps to solve a problem, you use math functions, and you don't worry about how the language performs the task. In some respects, this makes languages that support the functional programming paradigm similar to applications such as MATLAB. Of course, with MATLAB, you get a user interface, which reduces the learning curve. However, you pay for the convenience of the user interface with a loss of power and flexibility, which functional languages do offer. Using this approach to defining a problem relies on the *declarative programming* style, which you see used with other paradigms and languages, such as Structured Query Language (SQL) for database management.

In contrast to other paradigms, the functional programming paradigm doesn't maintain state. The use of *state* enables you to track values between function calls. Other paradigms use state to produce variant results based on environment, such as determining the number of existing objects and doing something different when the number of objects is zero. As a result, calling a functional program func tion always produces the same result given a particular set of inputs, thereby making functional programs more predictable than those that support state.

Because functional programs don't maintain state, the data they work with is also *immutable,* which means that you can't change it. To change a variable's value, you must create a new variable. Again, this makes functional programs more

predictable than other approaches and could make functional programs easier to run on multiple processors. The following sections provide additional information on how the functional programming paradigm differs.

Understanding its goals

Imperative programming, the kind of programming that most developers have done until now, is akin to an assembly line, where data moves through a series of steps in a specific order to produce a particular result. The process is fixed and rigid, and the person implementing the process must build a new assembly line every time an application requires a new result. Object-oriented programming (OOP) simply modularizes and hides the steps, but the underlying paradigm is the same. Even with modularization, OOP often doesn't allow rearrangement of the object code in unanticipated ways because of the underlying interdependencies of the code.

Functional programming gets rid of the interdependencies by replacing proce dures with pure functions, which requires the use of immutable state. Conse quently, the assembly line no longer exists; an application can manipulate data using the same methodologies used in pure math. The seeming restriction of immutable state provides the means to allow anyone who understands the math of a situation to also create an application to perform the math.

Using pure functions creates a flexible environment in which code order depends on the underlying math. That math models a real-world environment, and as our understanding of that environment changes and evolves, the math model and functional code can change with it — without the usual problems of brittleness that cause imperative code to fail. Modifying functional code is faster and less error prone because the person implementing the change must understand only the math and doesn't need to know how the underlying code works. In addition, learning how to create functional code can be faster as long as the person under stands the math model and its relationship to the real world.

Functional programming also embraces a number of unique coding approaches, such as the capability to pass a function to another function as input. This capa bility enables you to change application behavior in a predictable manner that isn't possible using other programming paradigms. As the book progresses, you encounter other such benefits of using functional programming.

Using the pure approach

Programming languages that use the pure approach to the functional program ming paradigm rely on lambda calculus principles, for the most part. In addition, a pure-approach language allows the use of functional programming techniques

only, so that the result is always a functional program. The pure-approach language used in this book is Haskell because it provides the purest implementation, according to articles such as the one found on Quora at `https://www.quora.com/`

What-are-the-most-popular-and-powerful-functional-programming languages. Haskell is also a relatively popular language, according to the TIOBE index (https://www.tiobe.com/tiobe-index/). Other pure-approach languages include Lisp, Racket, Erlang, and OCaml.

As with many elements of programming, opinions run strongly regarding whether a particular programming language qualifies for pure status. For example, many people would consider JavaScript a pure language, even though it's untyped. Oth ers feel that domain-specific declarative languages such as SQL and Lex/Yacc qualify for pure status even though they aren't general programming languages. Simply having functional programming elements doesn't qualify a language as adhering to the pure approach.

Using the impure approach

Many developers have come to see the benefits of functional programming. How ever, they also don't want to give up the benefits of their existing language, so they use a language that mixes functional features with one of the other program ming paradigms (as described in the "Considering Other Programming Para digms" section that follows). For example, you can find functional programming features in languages such as C++, C#, and Java. When working with an impure language, you need to exercise care because your code won't work in a purely functional manner, and the features that you might think will work in one way actually work in another. For example, you can't pass a function to another func tion in some languages.

At least one language, Python, is designed from the outset to support multiple programming paradigms (see https://blog.newrelic.com/2015/04/01/python-programming-styles/ for details). In fact, some online courses make a point of teaching this particular aspect of Python as a special benefit (see https://www.coursehero.com/file/p1hkiub/Python-supports-multiple-programming paradigms-including-object-oriented/). The use of multiple programming paradigms makes Python quite flexible but also leads to complaints and apologists (see
http://archive.oreilly.com/pub/post/pythons_weak_functional_progra.html as an example). The reasons that this book relies on Python to demonstrate the impure approach to functional programming is that it's both popular and flexible, plus it's easy to learn.

Considering Other

Programming Paradigms

You might think that only a few programming paradigms exist besides the func tional programming paradigm explored in this book, but the world of develop ment is literally packed with them. That's because no two people truly think completely alike. Each paradigm represents a different approach to the puzzle of conveying a solution to problems by using a particular methodology while making assumptions about things like developer expertise and execution environment. In fact, you can find entire sites that discuss the issue, such as the one at `http://cs.lmu.edu/~ray/notes/paradigms/`. Oddly enough, some languages (such as Python) mix and match compatible paradigms to create an entirely new way to perform tasks based on what has happened in the past.

The following sections discuss just four of these other paradigms. These para digms are neither better nor worse than any other paradigm, but they represent common schools of thought. Many languages in the world today use just these four paradigms, so your chances of encountering them are quite high.

Imperative

Imperative programming takes a step-by-step approach to performing a task. The developer provides commands that describe precisely how to perform the task from beginning to end. During the process of executing the commands, the code also modifies application state, which includes the application data. The code runs from beginning to end. An imperative application closely mimics the computer hardware, which executes machine code. *Machine code* is the lowest set of instruc tions that you can create and is mimicked in early languages, such as assembler.

Procedural

Procedural programming implements imperative programming, but adds func tionality such as code blocks and procedures for breaking up the code. The com piler or interpreter still ends up producing machine code that runs step by step, but the use of procedures makes it easier for a developer to follow the code and understand how it works. Many procedural languages provide a disassembly mode in which you can see the correspondence between the higher-level language and the underlying assembler. Examples of languages that implement the procedural paradigm are C and Pascal.

Early languages, such as Basic, used the imperative model because developers creating the languages worked closely with the computer hardware. However, Basic users often faced a problem called *spaghetti code,* which made large applica tions appear to be one monolithic piece. Unless you were the application's devel oper, following the application's logic was often hard. Consequently, languages that follow the procedural paradigm are a step up from languages that follow the imperative paradigm alone.

Object-oriented

The procedural paradigm does make reading code easier. However, the relation ship between the code and the underlying hardware still makes it hard to relate what the code is doing to the real world. The object-oriented paradigm uses the concept of objects to hide the code, but more important, to make modeling the real world easier. A developer creates code objects that mimic the real-world objects they emulate. These objects include properties, methods, and events to allow the object to behave in a particular manner. Examples of languages that implement the object-oriented paradigm are C++ and Java.

Languages that implement the object-oriented paradigms also implement both the procedural and imperative paradigms. The fact that objects hide the use of these other paradigms doesn't mean that a developer hasn't written code to create the object using these older paradigms. Consequently, the object-oriented paradigm still relies on code that modifies application state, but could also allow for modifying variable data.

Declarative

Functional programming actually implements the declarative programming par adigm, but the two paradigms are separate. Other paradigms, such as logic pro gramming, implemented by the Prolog language, also support the declarative programming paradigm. The short view of declarative programming is that it does the following:

» Describes what the code should do, rather than how to do it

» Defines functions that are referentially transparent (without side

effects) » Provides a clear correspondence to mathematical logic

to Perform Tasks

It's essential to remember that functional programming is a paradigm, which means that it doesn't have an implementation. The basis of functional program ming is lambda calculus (`https://brilliant.org/wiki/lambda-calculus/`), which is actually a math abstraction. Consequently, when you want to perform tasks by using the functional programming paradigm, you're really looking for a programming language that implements functional programming in a manner that meets your needs. (The next section, "Discovering Languages that Support Func tional Programming," describes the available languages in more detail.) In fact, you may even be performing functional programming tasks in your current language without realizing it. Every time you create and use a lambda function, you're likely using functional programming techniques (in an impure way, at least).

In addition to using lambda functions, languages that implement the functional programming paradigm have some other features in common. Here is a quick overview of these features:

- **First-class and higher-order functions:** First-class and higher-order func tions both allow you to provide a function as an input, as you would when using a higher-order function in calculus.
- **Pure functions:** A pure function has no side effects. When working with a pure function, you can
  - Remove the function if no other functions rely on its output
  - Obtain the same results every time you call the function with a given set of inputs
  - Reverse the order of calls to different functions without any change to application functionality
  - Process the function calls in parallel without any consequence
  - Evaluate the function calls in any order, assuming that the entire language doesn't allow side effects
- **Recursion:** Functional language implementations rely on recursion to implement looping. In general, recursion works differently in functional languages because no change in application state occurs.
- **Referential transparency:** The value of a variable (a bit of a misnomer because you can't change the value) never changes in a functional language implementation because functional languages lack an assignment operator.

You often find a number of other considerations for performing tasks in functional programming language implementations, but these issues aren't consistent across languages. For example, some languages use strict (eager) evaluation, while other languages use non-strict (lazy) evaluation. Under strict evaluation, the language fully checks the function before evaluating it. Even when a term within the function isn't used, a failing term will cause the function as a whole to fail. However, under non-strict evaluation, the function fails only if the failing term is used to create an output. The Miranda, Clean, and Haskell languages all implement non-strict evaluation.

Various functional language implementations also use different type systems, so the manner in which the underlying computer detects the type of a value changes from language to language. In addition, each language supports its own set of data structures. These kinds of issues aren't well defined as part of the functional pro
gramming paradigm, yet they're important to creating an application, so you must rely on the language you use to define them for you. Assuming a particular implementation in any given language is a bad idea because it isn't well defined as part of the paradigm.

Discovering Languages That Support Functional Programming

To actually use the functional programming paradigm, you need a language that implements it. As with every other paradigm discussed in this chapter, languages often fall short of implementing every idea that the paradigm provides, or they implement these ideas in unusual ways. Consequently, knowing the paradigm's rules and seeing how the language you select implements them helps you to understand the pros and cons of a particular language better. Also, understanding the paradigm makes comparing one language to another easier. The functional programming paradigm supports two kinds of language implementation, pure and impure, as described in the following sections.

Considering the pure languages

A pure functional programming language is one that implements only the func
tional programming paradigm. This might seem a bit limited, but when you read through the requirements in the "Using Functional Programming to Perform Tasks" section, earlier in the chapter, you discover that functional programming is mutually exclusive to programming paradigms that have anything to do with the imperative paradigm (which applies to most languages available today).

Trying to discover which language best implements the functional programming paradigm is nearly impossible because everyone has an opinion on the topic. You can find a list of 21 functional programming language implementations with their pros and cons at `https://www.slant.co/topics/485/~best-languages-for learning-functional-programming`.

Considering the impure languages

Python is likely the epitome of the impure language because it supports so many coding styles. That said, the flexibility that Python provides is one reason that people like using it so much: You can code in whatever style you need at the moment. The definition of an impure language is one that doesn't follow the rules for the functional programming paradigm fully (or at least not fully enough to call it pure). For example, allowing any modification of application state would instantly disqualify a language from consideration.

One of the more common and less understood reasons for disqualifying a lan guage as being a pure implementation of the functional programming paradigm is the lack of pure-function support. A pure function defines a specific relationship between inputs and outputs that has no side effects. Every call to a pure function with specific inputs always garners precisely the same output, making pure func tions extremely reliable. However, some applications actually rely on side effects to work properly, which makes the pure approach somewhat rigid in some cases. Chapters 4 and 5 provide specifics on the question of pure functions. You can also discover more in the article at `http://www.onlamp.com/2007/07/12/ introduction-to-haskell-pure-functions.html`.

Finding Functional Programming Online

Functional programming has become extremely popular because it solves so many problems. As covered in this chapter, it also comes with a few limitations, such as an inability to use mutable data; however, for most people, the pros outweigh the cons in situations that allow you to define a problem using pure math. (The lack of mutable data support also has pros, as you discover later, such as an ability to perform multiprocessing with greater ease.) With all this said, it's great to have resources when discovering a programming paradigm. This book is your first resource, but a single book can't discuss everything.

Online sites, such as Kevin Sookochef (`https://sookocheff.com/post/fp/a functional-learning-plan/`) and Wildly Inaccurate (`https://wildlyinaccurate.      com/functional-programming-resources/`), offer a great many helpful resources.

Hacker News (`https://news.ycombinator.com/item?id=16670572`) and Quora (`https://www.quora.com/What-are-good-resources-for-teaching-children functional-programming`) can also be great resources. The referenced Quora site is especially important because it provides information that's useful in getting children started with functional programming. One essential aspect of using online sites is to ensure that they're timely. The resource shouldn't be more than two years old; otherwise, you'll be getting old news.

Sometimes you can find useful videos online. Of course, you can find a plethora of videos of varying quality on YouTube (`https://www.youtube.com/ results?search_query=Functional+Programming`), but don't discount sites, such as tinymce (`https://go.tinymce.com/blog/talks-love-functional programming/`). Because functional programming is a paradigm and most of these videos focus on a specific language, you need to choose the videos you watch with care or you'll get a skewed view of what the paradigm can provide (as contrasted with the language).

One resource that you can count on being biased are tutorials. For example, the tutorial at `https://www.hackerearth.com/practice/python/functional programming/functional-programming-1/tutorial/` is all about Python, which, as noted in previous sections of this chapter, is an impure implementation. Likewise,
even solid tutorial makers, such as Tutorials Point
(`https://www.tutorialspoint.
com/functional_programming/functional_programming_introduction.htm`), have a hard time with this topic because you can't demonstrate a principle without a language. A tutorial can't teach you about a paradigm — at least, not easily, and not much beyond an abstraction. Consequently, when viewing a tutorial, even a tutorial that purports to provide an unbiased view of functional programming (such as the one at
`https://codeburst.io/a-beginner-friendly-intro-to-functional programming-4f69aa109569`), count on some level of bias because the examples will likely appear using a subset of the available languages.

» **Obtaining and using Python**

» **Downloading and installing the datasets and example code**

» **Running an application**

» **Writing Python code**

Chapter 2 Getting and Using Python

As mentioned in Chapter 1, Python is a flexible language that supports multiple coding styles, including an implementation of the functional programming paradigm. However, Python's implementation is impure because it does support the other coding styles. Consequently, you choose between flexibility and the features that functional programming can provide when you choose Python. Many developers choose flexibility (and therefore Python), but there is no right or wrong choice — just the choice that works best for you. This chapter helps you set up, configure, and become familiar with Python so that you can use it in the book chapters that follow.

This book uses Anaconda 5.1, which supports Python 3.6.4. If you use a different distribution, some of the procedural steps in the book will likely fail to work as expected, the screenshots will likely differ, and some of the example code may not run. To get the maximum benefit from this book, you need to use Anaconda 5.1, configured as described in the remainder of this chapter. The example application and other chapter features help you test your installation to ensure that it works as needed, so following the chapter from beginning to end is the best idea for a good programming experience.

Working with Python in This Book

You could download and install Python 3.6.4 to work with the examples in this book. Doing so would still allow you to gain an understanding of how functional programming works in the Python environment. However, using the pure Python installation will also increase the amount of work you must perform to have a good coding experience and even potentially reduce the amount you learn because your focus will be on making the environment work, rather than seeing how Python implements the functional programming paradigm. Consequently, this book relies on the Jupyter Notebook Integrated Development Environment (IDE) (or user interface or editor, as you might prefer) of the Anaconda tool collection to perform tasks for the reasons described in the following sections.

Creating better code

A good IDE contains a certain amount of intelligence. For example, the IDE can sug gest alternatives when you type the incorrect keyword, or it can tell you that a cer tain line of code simply won't work as written. The more intelligence that an IDE contains, the less hard you have to work to write better code. Writing better code is essential because no one wants to spend hours looking for errors, called *bugs.*

IDEs vary greatly in the level and kind of intelligence they provide, which is why so many IDEs exist. You may find the level of help obtained from one IDE to be insuf ficient to your needs, but another IDE hovers over you like a mother hen. Every developer has different needs and, therefore, different IDE requirements. The point is to obtain an IDE that helps you write clean, efficient code quickly and easily.

Debugging functionality

Finding bugs (errors) in your code involves a process called *debugging.* Even the most expert developer in the world spends time debugging. Writing perfect code on the first pass is nearly impossible. When you do, it's cause for celebration because it won't happen often. Consequently, the debugging capabilities of your IDE are critical. Unfortunately, the debugging capabilities of the native Python tools are almost nonexistent. If you spend any time at all debugging, you quickly find the native tools annoying because of what they don't tell you about your code.

The best IDEs double as training tools. Given enough features, an IDE can help you explore code written by true experts. Tracing through applications is a time honored method of learning new skills and honing the skills you already possess. A seemingly small advance in knowledge can often become a huge savings in time later. When looking for an IDE, don't just look at debugging features as a means to remove errors — see them also as a means to learn new things about Python.

Defining why notebooks are useful

Most IDEs look like fancy text editors, and that's precisely what they are. Yes, you get all sorts of intelligent features, hints, tips, code coloring, and so on, but at the end of the day, they're all text editors. Nothing is wrong with text editors, and this chapter isn't telling you anything of the sort. However, given that Python devel opers often focus on scientific applications that require something better than pure text presentation, using notebooks instead can be helpful.

A *notebook* differs from a text editor in that it focuses on a technique advanced by Stanford computer scientist Donald Knuth called literate programming. You use *literate programming* to create a kind of presentation of code, notes, math equa tions, and graphics. In short, you wind up with a scientist's notebook full of everything needed to understand the code completely. You commonly see literate programming techniques used in high-priced packages such as Mathematica and MATLAB. Notebook development excels at

- Demonstration
- Collaboration
- Research
- Teaching objectives
- Presentation

This book uses the Anaconda tool collection because it provides you with a great Python coding experience, but also because it helps you discover the enormous potential of literate programming techniques. If you spend a lot of time perform ing scientific tasks, Anaconda and products like it are essential. In addition, Anaconda is free, so you get the benefits of the literate programming style without the cost of other packages.

Obtaining Your Copy of Anaconda

As mentioned in the previous section, Anaconda doesn't come with your Python installation. With this in mind, the following sections help you obtain and install Anaconda on the three major platforms supported by this book.

Obtaining Analytics Anaconda

The basic Anaconda package comes as a free download that you obtain at `https://www.anaconda.com/download/`. Simply click the symbol for your operating

system, such as the window icon for Windows, and then click Download in the platform's section of the page to obtain access to the free product. (Depending on the Anaconda server load, the download can require a while to complete, so you may want to get a cup of coffee while waiting.) Anaconda supports the following platforms:

- Windows 32-bit and 64-bit (the installer might offer you only the 64-bit or 32-bit version, depending on which version of Windows it detects)
- Linux 32-bit and 64-bit
- Mac OS X 64-bit (both graphical and command-line installer)

You can obtain Anaconda with older versions of Python. If you want to use an older version of Python, click the How to Get Python 3.5 or Other Python Versions link near the middle of the page. You should use an older version of Python only when you have a pressing need to do so, however.

The free product is all you need for this book. However, when you look on the site, you see that many other add-on products are available. These products can help you create robust applications. For example, when you add Accelerate to the mix, you obtain the capability to perform multicore and GPU-enabled operations. The use of these add-on products is outside the scope of this book, but the Anaconda site gives you details on using them.

Installing Anaconda on Linux

You have to use the command line to install Anaconda on Linux; you're given no graphical installation option. Before you can perform the installation, you must download a copy of the Linux software from the Continuum Analytics site. You can find the required download information in the "Obtaining Analytics Anaconda" section, earlier in this chapter. The following procedure should work fine on any Linux system, whether you use the 32-bit or 64-bit version of Anaconda:

1. **Open a copy of Terminal.**

   The Terminal window appears.

2. **Change directories to the downloaded copy of Anaconda on your system.**

   The name of this file varies, but normally it appears as `Anaconda3-5.1.0- Linux-x86.sh` for 32-bit systems and `Anaconda3-5.1.0-Linux-x86_64.sh` for 64-bit systems. The version number is embedded as part of the filename. In this case, the filename refers to version 5.1.0, which is the version used for this book. If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

3. **Type** bash Anaconda3-5.1.0-Linux-x86.sh **(for the 32-bit version) or** bash Anaconda3-5.1.0-Linux-x86_64.sh **(for the 64-bit version) and press Enter.**

   An installation wizard starts that asks you to accept the licensing terms for using Anaconda.

4. **Read the licensing agreement and accept the terms using the method required for your version of Linux.**

   The wizard asks you to provide an installation location for Anaconda. The book assumes that you use the default location of ~/anaconda. If you choose some other location, you may have to modify some procedures later in the book to work with your setup.

5. **Provide an installation location (if necessary) and press Enter (or click Next).**

   The application extraction process begins. After the extraction is complete, you see a completion message.

6. **Add the installation path to your `PATH` statement using the method required for your version of Linux.**

   You're ready to begin using Anaconda.

Installing Anaconda on MacOS

The Mac OS X installation comes in only one form: 64-bit. Before you can perform the install, you must download a copy of the Mac software from the Continuum Analytics site. You can find the required download information in the "Obtaining Analytics Anaconda" section, earlier in this chapter.

The installation files come in two forms. The first depends on a graphical installer; the second relies on the command line. The command-line version works much like the Linux version described in the preceding section of this chapter, "Installing Anaconda on Linux.". The following steps help you install Anaconda 64-bit on a Mac system using the graphical installer:

1. **Locate the downloaded copy of Anaconda on your system.**

   The name of this file varies, but normally it appears as `Anaconda3-5.1.0- MacOSX-x86_64.pkg`. The version number is embedded as part of the file name. In this case, the filename refers to version 5.1.0, which is the version used for this book. If you use some other version, you may experience prob lems with the source code and need to make adjustments when working with it.

2. **Double-click the installation file.**

   An introduction dialog box appears.

3. **Click Continue.**

   The wizard asks whether you want to review the Read Me materials. You can read these materials later. For now, you can safely skip the information.

4. **Click Continue.**

   The wizard displays a licensing agreement. Be sure to read through the licensing agreement so that you know the terms of usage.

5. **Click I Agree if you agree to the licensing agreement.**

   You see a Standard Install dialog box where you can choose to perform a standard installation, change the installation location, or customize your setup. The standard installation is the one you should use for this book. Making changes could cause some steps within the book to fail unless you know how to modify the instructions to suit your setup.

6. **Click Install.**

   The installation begins. A progress bar tells you how the installation process is progressing. When the installation is complete, you see a completion dialog box.

7. **Click Continue.**

   You're ready to begin using Anaconda.

Installing Anaconda on Windows

Anaconda comes with a graphical installation application for Windows, so getting a good installation means using a wizard, as you would for any other installation. Of course, you need a copy of the installation file before you begin, and you can find the required download information in the "Obtaining Analytics Anaconda" section, earlier in this chapter. The following procedure (which can require a while to complete) should work fine on any Windows system, whether you use the 32-bit or 64-bit version of Anaconda:

1. **Locate the downloaded copy of Anaconda on your system.**

   The name of this file varies, but normally it appears as `Anaconda3-5.1.0-Windows-x86.exe` for 32-bit systems and `Anaconda3-5.1.0-Windows-x86_64. exe` for 64-bit systems. The version number is embedded as part of the filename. In this case, the filename refers to version 5.1.0, which is the version used for this book. If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

2. **Double-click the installation file.**

(You may see an Open File – Security Warning dialog box that asks whether you want to run this file. Click Run if you see this dialog box pop up.) You see an Anaconda3 5.1.0 Setup dialog box.

3. **Click Next.**

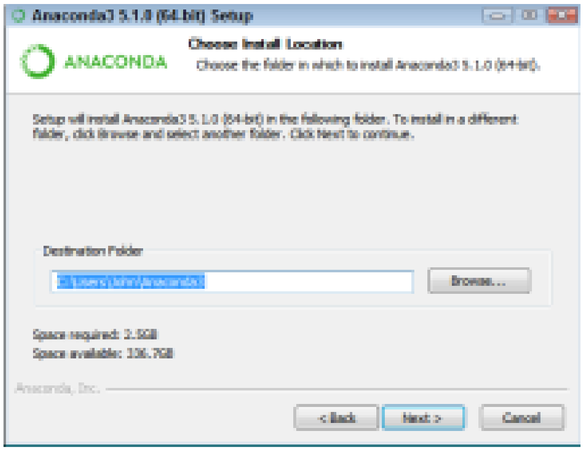
The wizard displays a licensing agreement. Be sure to read through the licensing agreement so that you know the terms of usage.

4. **Click I Agree if you agree to the licensing agreement.**

You're asked what sort of installation type to perform (personal or for everyone). In most cases, you want to install the product just for yourself. The exception is if you have multiple people using your system and they all need access to Anaconda.

5. **Choose one of the installation types and then click Next.**

The wizard asks where to install Anaconda on disk, as shown in Figure 2-1. The book assumes that you use the default location. If you choose some other location, you may have to modify some procedures later in the book to work with your setup.



Anaconda3 5.1.0 (64-bit) Setup

Choose Install Location

Choose the folder in which to install Anaconda3 5.1.0 (64-bit).

Setup will install Anaconda3 5.1.0 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

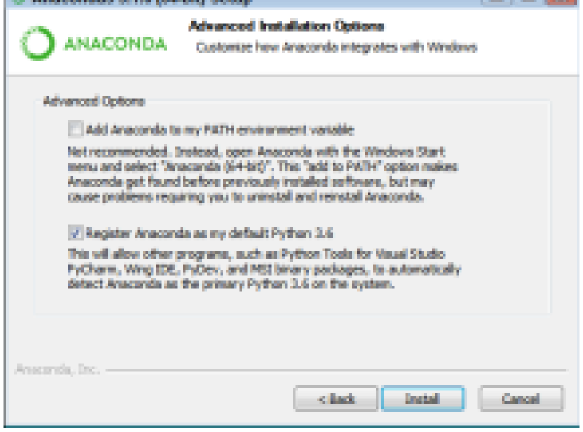
C:\Users\John\Anaconda3

Space required: 2.9GB
Space available: 336.7GB

FIGURE 2-1: Specify an installation location.

6. **Choose an installation location (if necessary) and then click Next.**

You see the Advanced Installation Options, shown in Figure 2-2. These options are selected by default, and no good reason exists to change them in most cases. You might need to change them if Anaconda won't provide your default Python 3.6.4 setup. However, the book assumes that you've set up Anaconda using the default options.


Advanced Installation Options
Customize how Anaconda integrates with Windows

Advanced Options

[ ] Add Anaconda to my PATH environment variable

Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.

[✓] Register Anaconda as my default Python 3.6

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.6 on the system.

FIGURE 2-2: Configure the advanced installation options.

7. **Change the advanced installation options (if necessary) and then click Install.**

   You see an Installing dialog box with a progress bar. The installation process can take a few minutes, so get yourself a cup of coffee and read the comics for a while. When the installation process is over, you see a Next button enabled.

8. **Click Next.**

   The wizard presents you with an option to install Microsoft VSCode. Installing this feature can cause problems with the book examples, so the best idea is not to install it. The book doesn't make use of this feature.

9. **Click Skip.**

   The wizard tells you that the installation is complete. You see options for learning more about Anaconda Cloud and getting started with Anaconda.

10. **Choose the desired learning options and then click Finish.**

    You're ready to begin using Anaconda.

Understanding the Anaconda package

The Anaconda package contains a number of applications, only one of which you use with this book. Here is a quick rundown on the tools you receive:

- **Anaconda Navigator:** Displays a listing of Anaconda tools and utilities (installed or not). You can use this utility to install, configure, and launch the various tools and utilities. In addition, Anaconda Navigator provides options to configure the overall Anaconda environment, select a project, obtain help, and interact with the Anaconda community. The "Getting Help with the Python Language" section, at the end of the chapter, tells you more about this tool.

- **Anaconda Prompt:** Opens a window into which you can type various commands to perform tasks such as starting a tool or utility from the com mand line, performing installations of sub-features using `pip`, and doing other command line-related tasks.
- **Jupyter Notebook:** Starts the IDE used for this book. The upcoming "Using Jupyter Notebook" section of the chapter gets you started using the IDE.
- **Reset Spyder Settings:** Changes the Spyder IDE settings to their original state. Use this option to correct Spyder settings when Spyder becomes unusable or otherwise fails to work as needed.
- **Spyder:** Starts a traditional IDE that allows you to type source code into an editor window and test it in various ways.

Downloading the Datasets and Example Code

This book is about using Python to perform functional programming tasks. Of course, you can spend all your time creating the example code from scratch, debugging it, and only then discovering how it relates to learning about the won ders of Python, or you can take the easy way and download the prewritten code from the Dummies site as described in the book's Introduction so that you can get right to work.

To use the downloadable source, you must install Jupyter Notebook. The "Obtaining Your Copy of Anaconda" section, earlier in this chapter, describes how to install Jupyter Notebook as part of Anaconda. You can also download Jupyter Notebook separately from `http://jupyter.org/`. Most of the code in this book will also work with Google Colaboratory, also called Colab (`https://colab.research.google.com/notebooks/welcome.ipynb`), but there is no guarantee all of the examples will work because Colab may not support all the required features and packages. Colab can be handy if you want to work through the examples on your tablet or other Android device. *Python For Data Science For Dummies,* 2nd Edition, by John Paul Mueller and Luca Massaron (Wiley) contains an entire chapter about using Colab with Python and can give you additional help.

The following sections show how to work with Jupyter Notebook, one of the tools found in the Anaconda package. These sections emphasize the capability to man age application code, including importing the downloadable source and exporting your amazing applications to show friends.

Using Jupyter Notebook

To make working with the code in this book easier, you use Jupyter Notebook. This IDE lets you easily create Python notebook files that can contain any number of examples, each of which can run individually. The program runs in your browser, so which platform you use for development doesn't matter; as long as it has a browser, you should be okay.

Starting Jupyter Notebook

Most platforms provide an icon to access Jupyter Notebook. Just click this icon to access Jupyter Notebook. For example, on a Windows system, you choose Start➪All Programs➪Anaconda 3➪Jupyter Notebook. Figure 2-3 shows how the interface looks when viewed in a Firefox browser. The precise appearance on your system depends on the browser you use and the kind of platform you have installed.

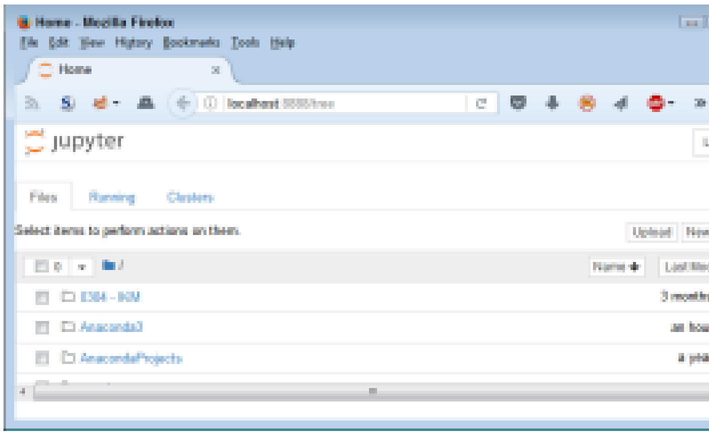


FIGURE 2-3: Jupyter Notebook provides an easy method to create machine learning examples.

Stopping the Jupyter Notebook server

No matter how you start Jupyter Notebook (or just Notebook, as it appears in the remainder of the book), the system generally opens a command prompt or termi nal window to host Jupyter Notebook. This window contains a server that makes

the application work. After you close the browser window when a session is com plete, select the server window and press Ctrl+C or Ctrl+Break to stop the server.

Defining the code repository

The code you create and use in this book will reside in a repository on your hard drive. Think of a *repository* as a kind of filing cabinet where you put your code. Notebook opens a drawer, takes out the folder, and shows the code to you. You can modify it, run individual examples within the folder, add new examples, and sim ply interact with your code in a natural manner. The following sections get you started with Notebook so that you can see how this whole repository concept works.

Defining the book's folder

It pays to organize your files so that you can access them more easily later. This book keeps its files in the FPD (Functional Programming For Dummies) folder. Use these steps within Notebook to create a new folder:

1. **Choose New ➪ Folder.**

   Notebook creates a new folder named Untitled Folder. The file appears in alphanumeric order, so you may not initially see it. You must scroll down to the correct location.

2. **Select the box next to the Untitled Folder entry.**

3. **Click Rename at the top of the page.**

   You see a Rename Directory dialog box like the one shown in Figure 2-4.

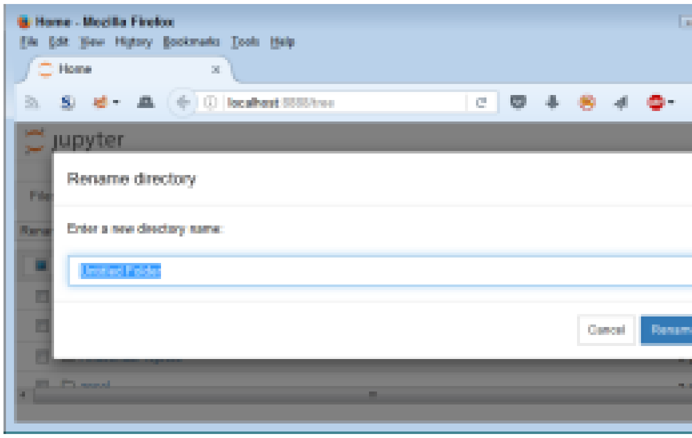


FIGURE 2-4: Rename the folder so that you remember the kinds of entries it contains.

4. **Type** FPD **and click Rename.**

   Notebook changes the name of the folder for you.

5. **Click the new FPD entry in the list.**

   Notebook changes the location to the FPD folder in which you perform tasks related to the exercises in this book.

Creating a new notebook

Every new notebook is like a file folder. You can place individual examples within the file folder, just as you would sheets of paper into a physical file folder. Each example appears in a cell. You can put other sorts of things in the file folder, too, but you see how these things work as the book progresses. Use these steps to create a new notebook:

1. **Click New ➪ Python 3.**

A new tab opens in the browser with the new notebook, as shown in Figure 2-5. Notice that the notebook contains a cell and that Notebook has highlighted the cell so that you can begin typing code in it. The title of the notebook is Untitled right now. That's not a particularly helpful title, so you need to change it.

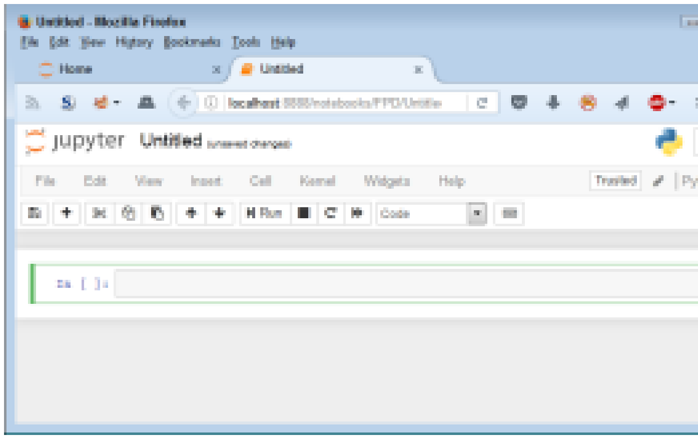

Untitled - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Home
Untitled
localhost:8888/notebooks/FPD/Untitle
jupyter Untitled (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Trusted
Run
Code
In [ ]:

FIGURE 2-5:
A notebook contains cells that you use to hold code.

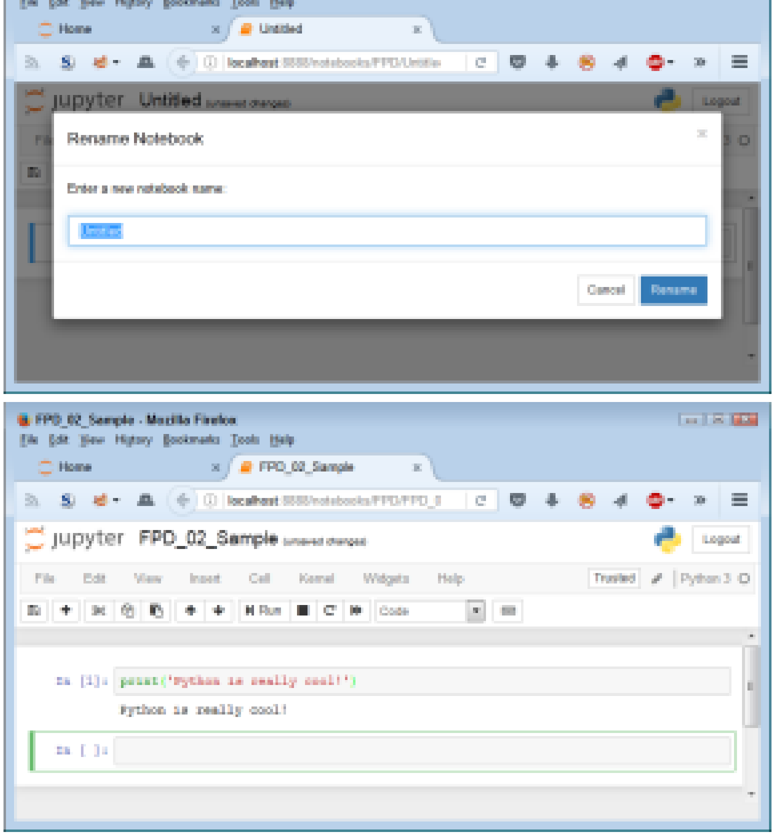
2. **Click Untitled on the page.**

Notebook asks what you want to use as a new name, as shown in Figure 2-6.

3. **Type** FPD_02_Sample **and press Enter.**

The new name tells you that this is a file for *Functional Programming For Dummies,* Chapter 2, `Sample.ipynb`. Using this naming convention lets you easily differentiate these files from other files in your repository.

Of course, the Sample notebook doesn't contain anything just yet. Place the cursor in the cell, type **print('Python is really cool!')**, and then click the Run button. You see the output shown in Figure 2-7. The output is part of the same cell as the code (the code resides in a square box and the output resides outside that square box, but both are within the cell). However, Notebook visually separates the output from the code so that you can tell them apart. Notebook creates a new cell for you.


Home
Untitled
jupyter Untitled (unsaved changes)
Logout
Cancel
Rename
FPD_02_Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Home
FPD_02_Sample
jupyter FPD_02_Sample (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

Rename Notebook

Enter a new notebook name:

Untitled

```
In [1]: print('Python is really cool!')

        Python is really cool!

In [ ]:
```

FIGURE 2-7: Notebook uses cells to store your code.

When you finish working with a notebook, shutting it down is important. To close a notebook, choose File➪Close and Halt. You return to Notebook's Home page, where you can see that the notebook you just created is added to the list.

Exporting a notebook

Creating notebooks and keeping them all to yourself isn't much fun. At some point, you want to share them with other people. To perform this task, you must export your notebook from the repository to a file. You can then send the file to someone else, who will import it into a different repository.

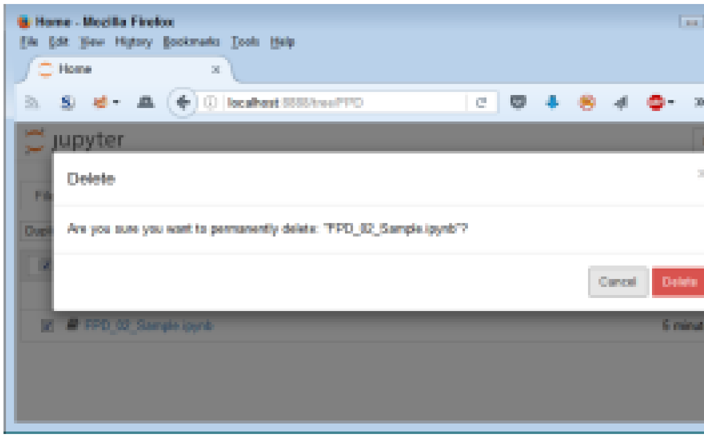
The previous section shows how to create a notebook named `FPD_02_Sample.ipynb` in Notebook. You can open this notebook by clicking its entry in the repository list. The file reopens so that you can see your code again. To export this code, choose File➪Download As➪Notebook (.ipynb). What you see next depends on your browser, but you generally see some sort of dialog box for saving the notebook as a file. Use the same method for saving the Notebook file as you use for any other file you save by using your browser. Remember to choose File➪Close and Halt when you finish so that the application shuts down.

Removing a notebook

Sometimes notebooks get outdated or you simply don’t need to work with them any longer. Rather than allow your repository to get clogged with files that you don’t need, you can remove these unwanted notebooks from the list. Use these steps to remove the file:

1. **Select the box next to the `FPD_02_Sample.ipynb` entry.**
2. **Click the trash can icon (Delete) at the top of the page.**

   You see a Delete notebook warning message like the one shown in Figure 2-8.



Are you sure you want to permanently delete: "FPD_02_Sample.ipynb"?

FIGURE 2-8: Notebook warns you before removing any files from the repository.

3. **Click Delete.**

   The file gets removed from the list.

Importing a notebook

To use the source code from this book, you must import the downloaded files into your repository. The source code comes in an archive file that you extract to a location on your hard drive. The archive contains a list of `.ipynb` (IPython Notebook) files containing the source code for this book (see the Introduction for details on downloading the source code). The following steps tell how to import these files into your repository:

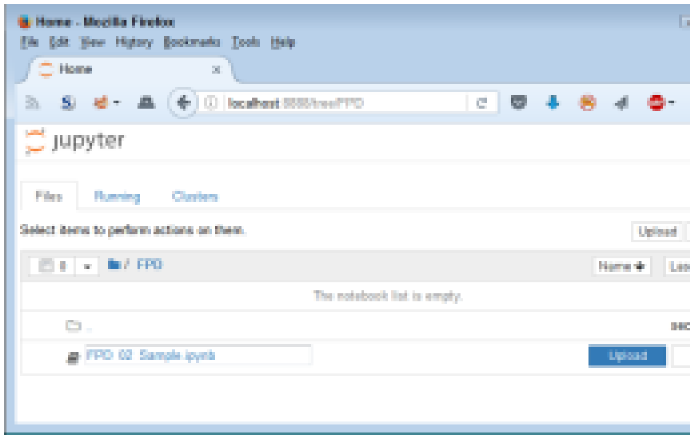


FIGURE 2-9:
The files that you want to add to the repository appear as part of an upload list consisting of one or more filenames.

1. **Click Upload at the top of the page.**

   What you see depends on your browser. In most cases, you see some type of File Upload dialog box that provides access to the files on your hard drive.

2. **Navigate to the directory containing the files that you want to import into Notebook.**

3. **Highlight one or more files to import and click the Open (or other, similar) button to begin the upload process.**

   You see the file added to an upload list, as shown in Figure 2-9. The file isn't part of the repository yet — you've simply selected it for upload.

4. **Click Upload.**

   Notebook places the file in the repository so that you can begin using it.

Getting and using datasets

This book uses a number of datasets, all of which appear in the Scikit-learn library. These datasets demonstrate various ways in which you can interact with data, and you use them in the examples to perform a variety of tasks. The following list provides a quick overview of the function used to import each of the datasets into your Python code:

» `load_boston()`: Regression analysis with the Boston house-prices dataset

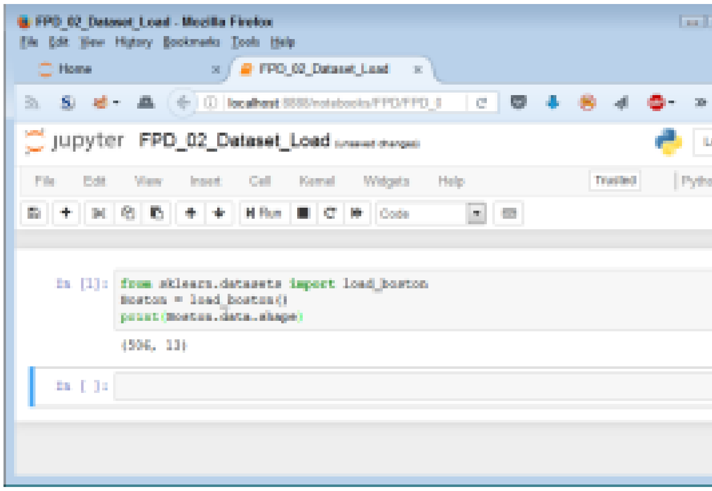
» `fetch_olivetti_faces()`: Olivetti faces dataset from AT&T

» `make_blobs()`: Generates isotropic Gaussian blobs used for clustering

The technique for loading each of these datasets is the same across examples. The following example shows how to load the Boston house-prices dataset. You can find the code in the FPD_02_Dataset_Load.ipynb notebook.

```
from sklearn.datasets import load_boston
Boston = load_boston()
print(Boston.data.shape)
```

To see how the code works, click Run. The output from the `print` call is `(506, 13)`. You can see the output shown in Figure 2-10.


FPD_02_Dataset_Load - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Home
FPD_02_Dataset_Load
localhost:8888/notebooks/FPD/FPD_0
jupyter FPD_02_Dataset_Load (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Trusted
Run
Code

```
In [1]: from sklearn.datasets import load_boston
        Boston = load_boston()
        print(Boston.data.shape)

        (506, 13)

In [ ]:
```

FIGURE 2-10: The Boston object contains the loaded dataset.

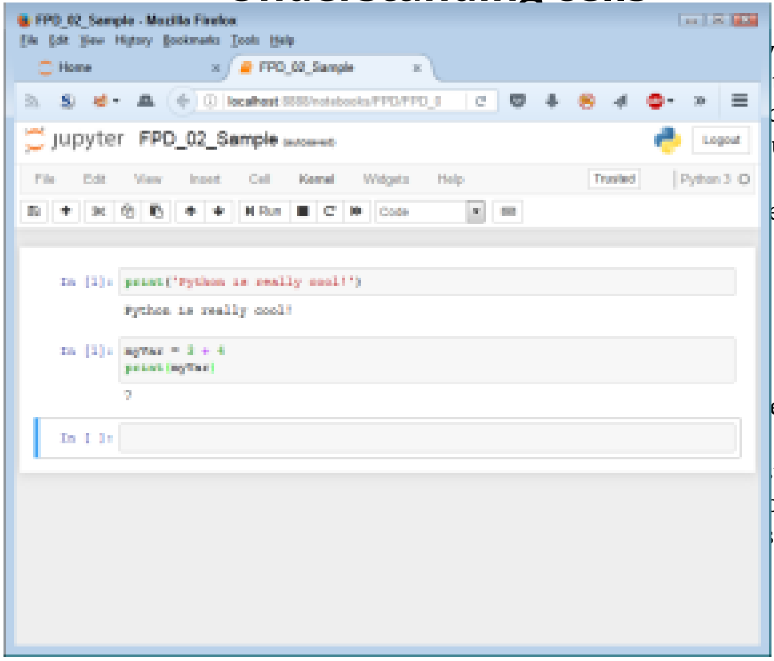
The line `from sklearn.datasets import load_boston` is special because it tells Python to use an external module. In this case, the external module is called `sklearn.datasets`, and Python loads the `load_boston` function from it. After the function is loaded, you can call it from your code, as shown in the next line. You see external modules used quite often in the book, so for now you just need to know that they exist and that you can load them as needed.

Creating a Python Application

Actually, you've already created your first Anaconda application by using the steps in the "Creating a new notebook" section, earlier in this chapter. The `print()` method may not seem like much, but you use it quite often. However, the literate programming approach provided by Anaconda requires a little more knowledge

than you currently have. The following sections don't tell you everything about this approach, but they do help you gain an understanding of what literate pro gramming can provide in the way of functionality. However, before you begin, make sure you have the `FPD_02_Sample.ipynb` file open for use because you need it to explore Notebook.

Understanding cells

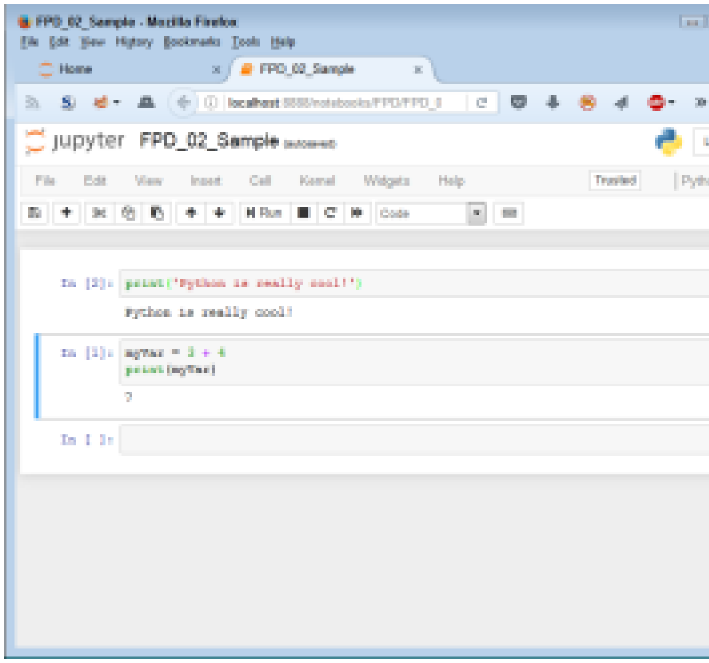

FPD_02_Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Home
FPD_02_Sample
localhost:8888/notebooks/FPD/FPD_0
jupyter FPD_02_Sample (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [1]: print('Python is really cool!')

        Python is really cool!

In [1]: myVar = 3 + 4
        print(myVar)

        7

In [ ]:
```

e cells. What you'd have is a doc
tements. To separate various
different because each cell is
us cells matter, but if a cell is
and run it. To see how this
e next cell of the `FPD_02_Sample`

e executes, and you see the
expected. However, notice the
t cell executed during this ses
ore restart the numbers at 1),
tart options).

FIGURE 2-11: Cells execute individually in Notebook.



```
In [2]: print('Python is really cool!')
        Python is really cool!

In [1]: myVar = 3 + 4
        print(myVar)
        7

In [ ]:
```

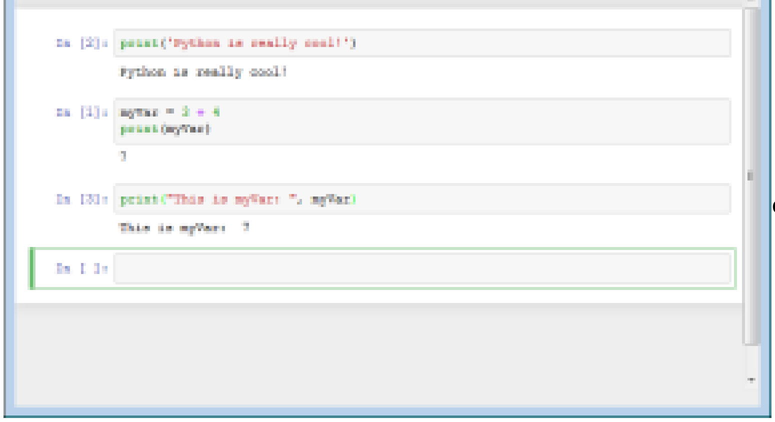
FIGURE 2-12:
Cells can execute in any order in Notebook.

Note that the first cell also has an In [1]: entry. This entry is still from the previous session. Place your cursor in that cell and click Run. Now the cell contains In [2]:, as shown in Figure 2-12. However, note that the next cell hasn't been selected and still contains the In [1]: entry.

Now place the cursor in the third cell — the one that is currently blank — and type `print("This is myVar: ", myVar)`. Click Run. The output in Figure 2-13 shows that the cells have executed in anything but a rigid order, but that `myVar` is global to the notebook. What you do in other cells with data affects every other cell, no matter in what order the execution takes place.

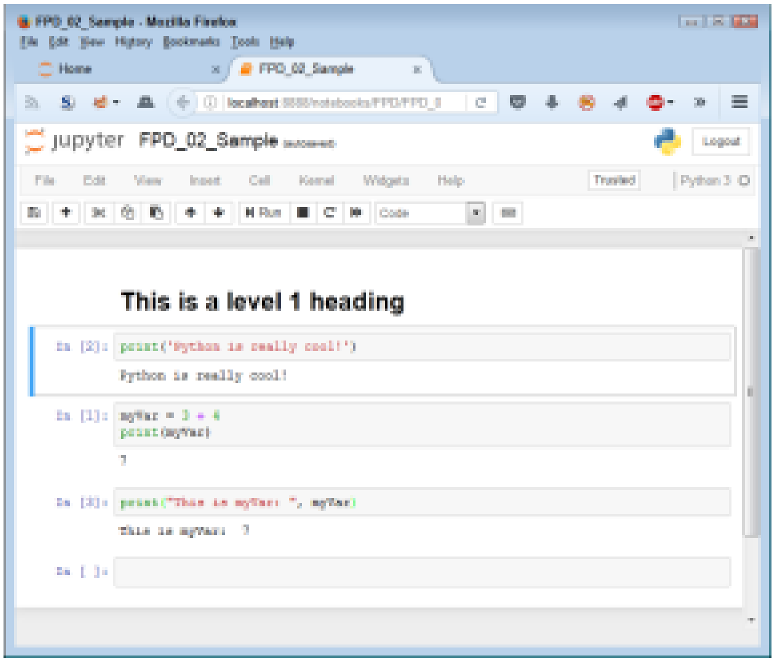
Adding documentation cells

Cells come in a number of different forms. This book doesn't use them all. How ever, knowing how to use the documentation cells can come in handy. Select the first cell (the one currently marked with a 2). Choose Insert➪Insert Cell Above. You see a new cell added to the notebook. Note the drop-down list that currently shows the word *Code*. This list allows you to choose the kind of cell to create. Select Markdown from the list and type **# This is a level 1 heading**. Click Run (which may seem like an extremely odd thing to do, but give it a try). You see the text change into a heading, as shown in Figure 2-14. However,



```
In [2]: print('Python is really cool!')
        Python is really cool!

In [1]: myVar = 3 + 4
        print(myVar)
        7

In [3]: print("This is myVar: ", myVar)
        This is myVar:  7

In [ ]:
```

e cells have.



FPD_02_Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Home
FPD_02_Sample
localhost:8888/notebooks/FPD/FPD_0
jupyter FPD_02_Sample (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

This is a level 1 heading

```
In [2]: print('Python is really cool!')
        Python is really cool!

In [1]: myVar = 3 + 4
        print(myVar)
        7

In [3]: print("This is myVar: ", myVar)
        This is myVar:  7

In [ ]:
```

FIGURE 2-14: Adding headings helps you separate and document your code.

About now, you might be thinking that these special cells act just like HTML pages, and you'd be right. Choose Insert⇨Insert Cell Below, select Markdown in the drop-down list, and then type **## This is a level 2 heading.** Click Run. As you can see, the number of hashes (#) you add to the text affects the heading level, but the hashes don't show up in the actual heading.

Other cell content

This chapter (and book) doesn't demonstrate all the kinds of cell content that you can see by using Notebook. However, you can add things like graphics to your notebooks, too. When the time comes, you can output (print) your notebook as a report and use it in presentations of all sorts. The literate programming technique is different from what you may have used in the past, but it has definite advan tages, as you see in upcoming chapters.

Running the Python Application

The code you create using Notebook is still code and not some mystical unique file that only Notebook can understand. When working with any file, such as the `FPD_02_Sample`, you can choose File⇨Download As⇨Python (.py) to output the Notebook as a Python file. Try it and you end up with `FPD_02_Sample.py`.

To see the code run as it would using Python directly, open an Anaconda Prompt,
tart⇨All Programs⇨Ana
as special features that make
ge Directory (CD) command
holds the source code file.
our code will execute as

FIGURE 2-15: You can use the Python interpreter

directly to
execute
your code.

This book doesn't spend much time using this approach because, as you can see, it's harder to use and understand than working with Notebook. However, it's still a perfectly acceptable way to execute your own code.

Understanding the Use of Indentation

As you work through the examples in this book, you see that certain lines are indented. In fact, the examples also provide a fair amount of white space (such as extra lines between lines of code). Python ignores extra lines for the most part, but relies on indentation to show certain coding elements (so the use of indenta
tion is essential). For example, the code associated with a function is indented under that function so that you can easily see where the function begins and ends. The main reason to add extra lines is to provide visual cues about your code, such as the end of a function or the beginning of a new coding element.

The various uses of indentation will become more familiar as go through the examples in the book. However, you should know at the outset why indentation is used and how it gets put in place. To that end, it's time for another example. The following steps help you create a new example that uses indentation to make the relationship between application elements a lot more apparent and easier to figure out later:

1. **Choose New ➪Python3.**

   Jupyter Notebook creates a new notebook for you. The downloadable source uses the filename `FPD_02_Indentation.ipynb`, but you can use any name you want.

2. **Type** print("This is a really long line of text that will " **+.**

   You see the text displayed normally onscreen, just as you expect. The plus sign (+) tells Python that there is additional text to display. Adding text from multiple lines together into a single long piece of text is called *concatenation.* You learn more about using this feature later in the book, so you don't need to worry about it now.

3. **Press Enter.**

   The insertion point doesn't go back to the beginning of the line, as you might expect. Instead, it ends up directly under the first double quote, as shown in Figure 2-16. This feature is called automatic indention and is one of the features that differentiates a regular text editor from one designed to write code.

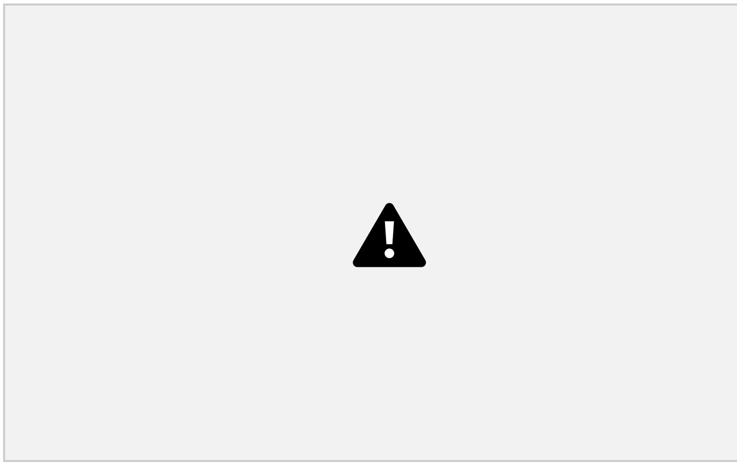


FIGURE 2-16:
The Edit window automatically indents some types of text.

4. **Type** “appear on multiple lines in the source code file.”) **and press Enter.**

Notice that the insertion point goes back to the beginning of the line. When Notebook senses that you have reached the end of the code, it automatically outdents the text to its original position.

5. **Click Run.**

You see the output shown in Figure 2-17. Even though the text appears on multiple lines in the source code file, it appears on just one line in the output. The line does break because of the size of the window, but it’s actually just one line.

FIGURE 2-17:
Use
concatenation to make multiple lines of text appear on a
single line in the output.

HEADINGS VERSUS COMMENTS

You may find headings and comments a bit confusing at first. Headings appear in sepa rate cells; comments appear with the source code. They serve different purposes.

Headings serve to tell you about an entire code grouping, and individual comments tell you about individual code steps or even lines of code. Even though you use both of them for documentation, each serves a unique purpose. Comments are generally more detailed than headings.

Adding Comments

People create notes for themselves all the time. When you need to buy groceries, you look through your cabinets, determine what you need, and write it down on a list. When you get to the store, you review your list to remember what you need. Using notes comes in handy for all sorts of needs, such as tracking the course of a conversation between business partners or remembering the essential points of a lecture. Humans need notes to jog their memories. *Comments* in source code are just another form of note. You add comments to the code so that you can remem ber what task the code performs later. The following sections describe comments in more detail. You can find these examples in the `FPD_02_Comments.ipynb` file in the downloadable source.

Understanding comments

Computers need some special way to determine that the text you're writing is a comment, not code to execute. Python provides two methods of defining text as a comment and not as code. The first method is the single-line comment. It uses the hash, also called the number sign (#), like this:

```
 # This is a comment.
print("Hello from Python!") #This is also a comment.
```

A single-line comment can appear on a line by itself or after executable code. It appears on only one line. You typically use a single-line comment for short descriptive text, such as an explanation of a particular bit of code. Notebook shows comments in a distinctive color (usually blue) and in italics.

Python doesn't actually support a multiline comment directly, but you can create one using a triple-quoted string. A multiline comment both starts and ends with three double quotes (""") or three single quotes (''') like this:

```
"""
 Application: Comments.py
 Written by: John
 Purpose: Shows how to use comments.
"""
```

These lines aren't executed. Python won't display an error message when they appear in your code. However, Notebook treats them differently, as shown in Figure 2-18. Note that the actual Python comments, those preceded by a hash (#) in cell 1, don't generate any output. The triple-quote strings, however, do gener
ate output. If you plan to output your notebook as a report, you need to avoid using triple-quoted strings. (Some IDEs, such as IDLE, ignore the triple-quoted strings completely.)

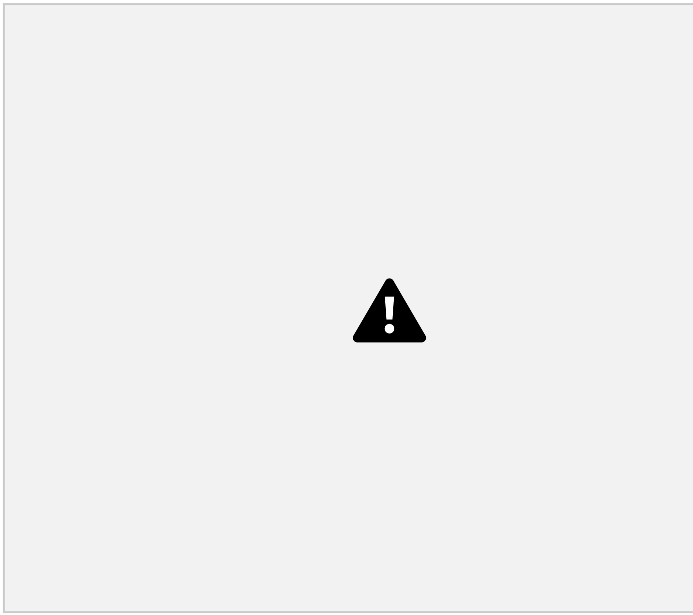


FIGURE 2-18: Multiline comments do work, but they also provide output.

You typically use multiline comments for longer explanations of who created an application, why it was created, and what tasks it performs. Of course, no hard

rules exist for precisely how to use comments. The main goal is to tell the com puter precisely what is and isn't a comment so that it doesn't become confused.

Using comments to leave yourself reminders

A lot of people don't really understand comments—they don't quite know what to do with notes in code. Keep in mind that you might write a piece of code today and then not look at it for years. You need notes to jog your memory so that you remember what task the code performs and why you wrote it. Here are some com mon reasons to use comments in your code:

» Reminding yourself about what the code does and why you wrote
it » Telling others how to maintain your code
» Making your code accessible to other developers
» Listing ideas for future updates
» Providing a list of documentation sources you used to write the
code » Maintaining a list of improvements you've made

You can use comments in a lot of other ways, too, but these are the most common ways. Look at how comments are used in the examples in the book, especially as you get to later chapters where the code becomes more complex. As your code becomes more complex, you need to add more comments and make the comments pertinent to what you need to remember about it.

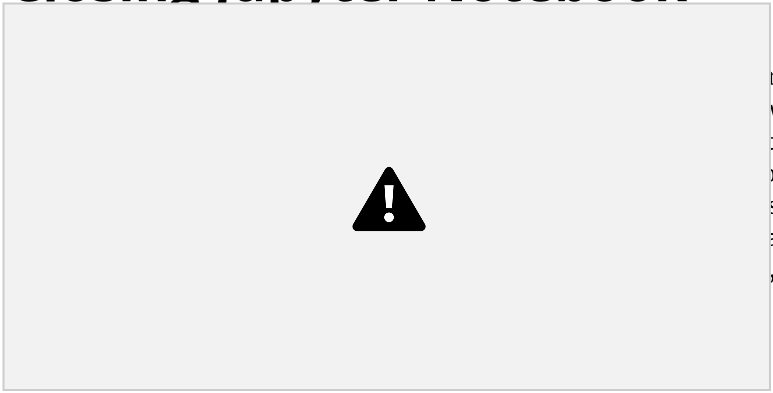
Using comments to keep code from executing

Developers also sometimes use the commenting feature to keep lines of code from executing (referred to as *commenting out*). You might need to do this to determine whether a line of code is causing your application to fail. As with any other com ment, you can use either single-line commenting or multiline commenting. How ever, when using multiline commenting, you do see the code that isn't executing

as part of the output (and it can actually be helpful to see where the code affects the output). Here is an example of both forms of commenting out:

```
# print("This print statement won't print")

"""
  print("This print statement appears as output")
"""
```

Closing Jupyter Notebook

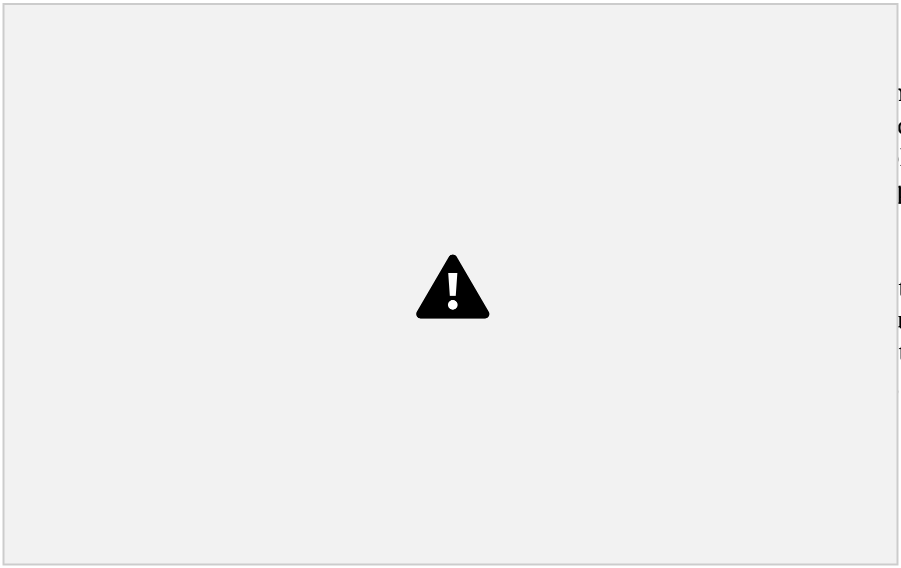


nd to close each of the note
vs), you can simply close the
to end your session. However,
ontinues to run in the back
s when you start Notebook,
ains open until you stop the
, and the window will close.

FIGURE 2-19: Make sure to close the server window.

Look again at Figure 2-19 to note a number of commands. These commands tell you what the user interface is doing. By monitoring this window, you can deter mine what might go wrong during a session. Even though you won't use this feature very often, it's a handy trick to know.

Getting Help with the Python Language



many of them appear
ce you need to know
lication by choosing
pplication requires a

t working with the
re 2-20, is different
ted documentation,
simply click the one

FIGURE 2-20: Use the Learning tab to get standardized information.

Note that the page contains more than just Python-specific or Anaconda-specific resources. You also gain access to information about common Python resources, such as the SciPy library.

The Community tab, shown in Figure 2-21, provides access to events, forums, and social entities. Some of this content changes over time, especially the events. To get a quick overview of an entry, hover the mouse over it. Reading an overview is especially helpful when deciding whether you want to learn more about events.

Forums differ from social media by the level of formality and the mode of access. For example, the Stack Overflow allows you to ask Python-related questions, and

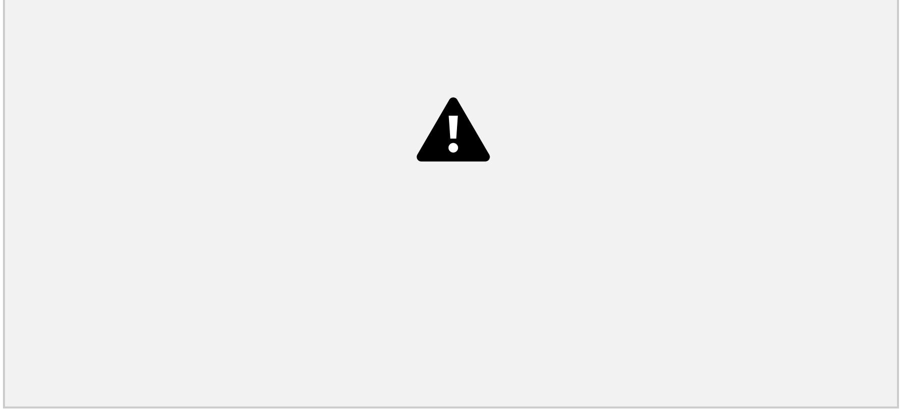


FIGURE 2-21: Use the Community tab to discover interactive information resources.

» Obtaining and using Haskell » Using GHCi and WinGHCi » Writing Haskell code

» Finding additional information

Chapter 3

Getting and Using Haskell

The first sections of this chapter discuss the goals behind the Haskell instal lation for this book, help you obtain a copy of Haskell, and then show you how to install Haskell on any one of the three supported book platforms: Linux, Mac, and Windows. Overall, this chapter focuses on providing you with the simplest possible installation so that you can clearly see how the functional pro gramming paradigm works. You may eventually find that you need a different installation to meet specific needs or tool requirements.

After you have Haskell installed, you perform some simple coding tasks using it. The main purpose of writing this code is to verify that your copy of Haskell is working properly, but it also helps familiarize you with Haskell just a little. A sec ond example helps you become familiar with using Haskell libraries, which is important when viewing the examples in this book.

The final section of the chapter helps you locate some Haskell resources. This book doesn't provide you with a solid basis for learning how to program in Haskell. Rather, it focuses on the functional programming paradigm, which can rely on Haskell for a pure implementation approach. Consequently, even though the text gives some basic examples, it doesn't provide a complete treatment of the lan guage, and the aforementioned other resources will help you fill in the gaps if you're new to Haskell.

Working with Haskell in This Book

You can encounter many different, and extremely confusing, ways to work with Haskell. All you need to do is perform a Google search and, even if you limit the results to the past year, you find that everyone has a differing opinion as to how to obtain, install, and configure Haskell. In addition, various tools work with Haskell configured in different ways. You also find that different platforms sup port different options. Haskell is both highly flexible and relatively new, so you have stability issues to consider. This chapter helps you create a Haskell configu ration that's easy to work with and allows you to focus on the task at hand, which is to discover the wonders of the functional programming paradigm.

To ensure that the code that you find in this book works well, make sure to use the 8.2.2 version of Haskell. Older versions may lack features or require bug fixes to make the examples work. You also need to verify that you have a compatible installation by using the instructions found in the upcoming "Obtaining and Installing Haskell" section. Haskell provides a number of very flexible installation options that may not be compatible with the example code.

Obtaining and Installing Haskell

You can obtain Haskell for each of the three platforms supported by this book at `https://www.haskell.org/platform/prior.html`. Simply click the icon corre sponding to the platform of your choice. The page takes you to the section that corresponds with the platform.

In all three cases, you want to perform a full installation, rather than a core instal lation, because the core installation doesn't provide support for some of the pack ages used in the book. Both Mac and Windows users can use only a 64-bit installation. In addition, unless you have a good reason to do otherwise, Mac users should rely on the installer, rather than use Homebrew Cask. Linux users should rely on the 64-bit installation as well because you obtain better results. Make sure that you have plenty of drive space for your installation. For example, even though the Windows download file is only 269MB, the Haskell Platform folder will con sume 2.6GB of drive space after the installation is complete.

You can encounter a problem when clicking the links on the initial page. If you find that the download won't start, go to `https://downloads.haskell.org/~platform/8.2.2/` instead and choose the particular link for your platform:

- **Generic Linux:** haskell-platform-8.2.2-unknown-posix--full-i386.tar.gz
- **Specific Linux:** See the installation instructions in the "Installing Haskell on a Linux system" section that follows
- **Mac:** Haskell Platform 8.2.2 Full 64bit-signed.pkg
- **Windows:** HaskellPlatform-8.2.2-full-x86_64-setup.exe

Haskell supports some Linux distributions directly. If this is the case, you don't need to download a copy of the product. The following sections get you started with the various installations.

USING HASKELL IDEs AND ENVIRONMENTS

You can find a number of IDEs and environments online for Haskell. Many of these options, such as Vim (`https://www.vim.org/download.php`), neoVim (`https://neovim.io/`), and Emacs (`https://www.gnu.org/software/emacs/download.html`), are enhanced text editors. The problem is that the editors provide uneven fea ture sets for the platforms that they support. In addition, in each case you must per form additional installations to obtain Haskell support. For example, emacs requires the use of haskell-mode (`https://github.com/haskell/haskell-mode/wiki`). Consequently, you won't find them used in this book.

Likewise, you can find a Jupyter Notebook add-on for Haskell at `https://github.com/gibiansky/IHaskell`. The add-on works well as long as you have either Mac or supported Linux as your platform. No Windows support exists for this add-on unless you want to create a Linux virtual machine in which to run it. You can read a discussion of the issues surrounding this add-on at `https://news.ycombinator.com/ item?id=12783913`.

Yet another option is a full-blown Integrated Development Environment (IDE), such as Leksah (`http://leksah.org/`), which is *Haskell* spelled backward with just one *L*, or HyperHaskell (`https://github.com/HeinrichApfelmus/hyper-haskell`). Most of these IDEs require that you perform a build, and the setups can become horribly com plex for the novice developer. Even so, an IDE can give you advanced functionality, such as a debugger. There really isn't a correct option, but the focus of this book is to make things simple.

Installing Haskell on a Linux system

Linux users numerous options from which to choose. If you see instructions for your particular Linux distribution, you may not even need to download Haskell directly. The `$ sudo apt-get` command may do everything needed. Use this option if possible. Otherwise, rely on the installation tarball for generic Linux. The specific Linux installations are:

- Ubuntu
- Debian
- Linux Mint
- Redhat
- Fedora
- Gentoo

A generic Linux installation assumes that you don't own one of the distributions in the previous list. In this case, make sure that you download the tarball found in the introduction to this section and follow these instructions to install it:

1. **Type** tar xf haskell-platform-8.2.2-unknown-posix--full-i386.tar.gz **and press Enter.**

   The system extracts the required files for you.
2. **Type** sudo ./install-haskell-platform.sh **and press Enter.**

   The system performs the required installation for you. You'll likely see prompts during the installation process, but these prompts vary by system. Simply answer the questions as you proceed to complete the installation.

This book won't help you build Haskell from source, and the results are unreliable enough that this approach isn't recommended for the novice developer. If you find that you absolutely must build Haskell from source files, make sure that you rely on the instructions found in the README file provided with the source code, rather than online instructions that may reflect the needs of an older version of

Linux.

Installing Haskell on a Mac system

When working with a Mac platform, you need to access a Haskell installer specifi cally designed for a Mac. This chapter assumes that you don't want to take time or

effort to create a custom configuration using source code. The following steps describe how to perform the installation using the graphical installer.

1. **Locate the downloaded copy of Haskell Platform 8.2.2 Full 64bit-signed. pkg on your system.**

   If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

2. **Double-click the installation file.**

   You see a Haskell Platform 8.2.2 64-bit Setup dialog box.

3. **Click Next.**

   The wizard displays a licensing agreement. Be sure to read the licensing agreement so that you know the terms of usage.

4. **Click I Agree if you agree to the licensing agreement.**

   The setup wizard asks where you want to install your copy of Haskell. This book assumes that you use the default installation location.

5. **Click Next.**

   You see a dialog box asking which features to install. This book assumes that you install all the default features.

6. **Click Next.**

   You see a new dialog box appear that asks where to install the Haskell Stack. Use the default installation location to ensure that your setup works correctly.

7. **Click Next.**

   The setup wizard asks you which features to install. You must install all of them.

8. **Click Install.**

   You see the Haskell Stack Setup wizard complete.

9. **Click Close.**

   You see the Haskell Platform wizard progress indicator move. At some point, the installation completes.

10. **Click Next.**

    You see a completion dialog box.

11. **Click Finish.**

    Haskell is now ready for use on your system.

Installing Haskell on a Windows system

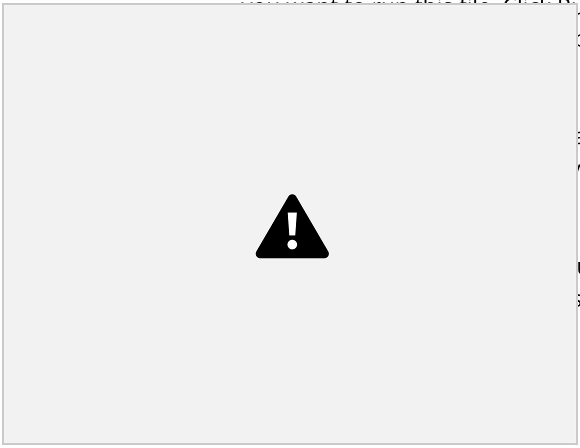
When working with a Windows platform, you need access to a Haskell installer specifically designed for Windows. The following steps assume that you've down loaded the required file, as described in the introduction to this section.

1. **Locate the downloaded copy of HaskellPlatform-8.2.2-full-x86_64-setup. exe on your system.**

   If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

2. **Double-click the installation file.**

   (You may see an Open File – Security Warning dialog box that asks whether you want to run this file. Click Run if you see this dialog box pop up.) You see



bit Setup dialog box.

agreement. Be sure to read through the

you know the terms of usage.

he licensing agreement.

u want to install your copy of Haskell, as shown

that you use the default installation location,

one.

FIGURE 3-1:

Specify a Haskell installation location.

FIGURE 3-2: Choose which Haskell features to install.

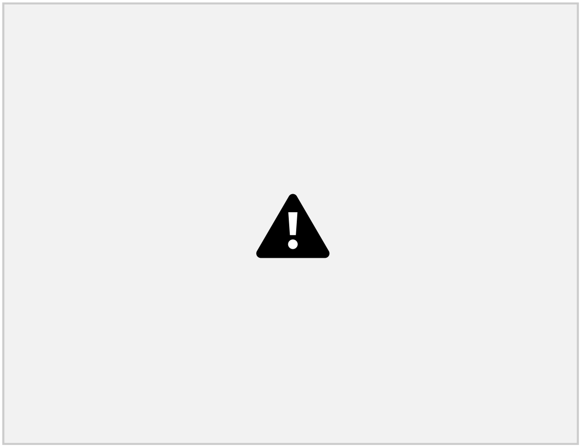
FIGURE 3-3: Type a Start menu folder name, if desired.

5. **Optionally provide an installation location and then click Next.**

You see a dialog box asking which features to install. This book assumes that you install all the default features, as shown in Figure 3-2. Note especially the Update System Settings option. You must ensure that this option is selected to obtain proper functioning of the Haskell features.

6. **Choose the features you want to use and click Next.**

The setup wizard asks you which Start menu folder to use, as shown in Figure 3-3. The book assumes that you use the default Start menu folder, but you can enter a name that you choose.



7. **Optionally type a new Start menu folder name and click Install.**

   You see a new dialog box appear that asks where to install the Haskell Stack. Use the default installation location unless you need to change it for a specific reason, such as using a local folder rather than a roaming folder.

8. **Optionally type a new location and click Next.**

   The setup wizard asks you which features to install. You must install all of them.

9. **Click Install.**

   You see the Haskell Stack Setup wizard complete.

10. **Click Close.**

    You see the Haskell Platform wizard progress indicator move. At some point, the installation completes.

11. **Click Next.**

    You see a completion dialog box.

12. **Click Finish.**

    Haskell is now ready for use on your system.

Testing the Haskell Installation

As explained in the "Using Haskell IDEs and Environments" sidebar, you have access to a considerable number of environments for working with Haskell. In fact, if you're using Linux or Mac platforms, you can rely on an add-in for the Jupyter Notebook environment used for Python in this book. However, to make things simple, you can use the Glasgow Haskell Compiler interpreter (GHCi) that comes with the Haskell installation you created earlier. Windows users have a graphical interface they can use called WinGHCi that works precisely the same as GHCi, but with a nicer appearance, as shown in Figure 3-4.

You can find either GHCi or WinGHCi in the folder used to store the Haskell appli cation icons on your system. When working with Windows, you find this file at Start➪All Programs➪Haskell Platform 8.2.2. No matter how you open the inter preter, you see the version number of your installation, as shown in Figure 3-4.

FIGURE 3-4:
The WinGHCi interface offers a nice appearance and is easy
to use.

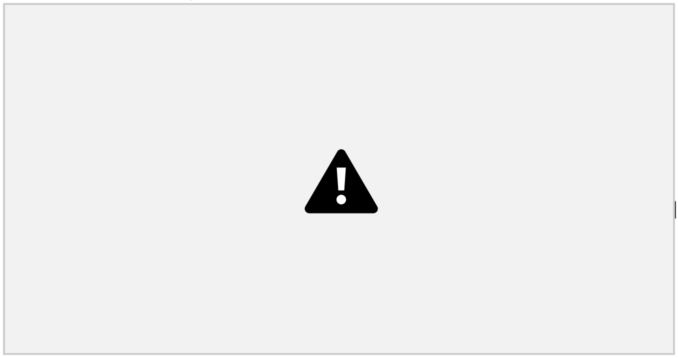
The interpreter can provide you with a great deal of information about Haskell, and simply looking at what's available can be fun. The commands all start with a colon, including the help commands. So to start the process, you type **:?** and press Enter. Figure 3-5 shows typical results.

FIGURE 3-5:
Make sure to
precede all help commands with a colon (:) in the
interpreter.

Playing with Haskell is the best way to learn it. Type **"Haskell is fun!"** and press Enter. You see the string repeated onscreen, as shown in Figure 3-6. All Haskell has done is evaluate the string you provided.

As a next step, try creating a variable by typing **x = "Haskell is really fun!"** and pressing Enter. This time, Haskell doesn't interpret the information but simply places the string in `x`. To see the string, you can use the `putStrLn` function. Type **putStrLn x** and press Enter. Figure 3-7 shows what you should see. At this point, you know that the Haskell installation works.

As you look through the list, you see that all commands begin with a colon. For example, to



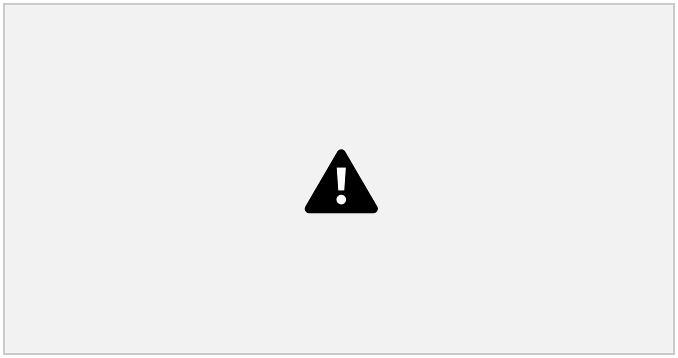


FIGURE 3-7: Haskell uses variables and functions to interact with the user.

Compiling a Haskell Application

Even though you'll perform most tasks in this book using the interpreter, you can also load modules and interpret them. In fact, this is how you use the download able source: You load it into the interpreter and then execute it. To see how this works, create a text file on your system called `Simple.hs`. You must use a pure text editor (one that doesn't include any formatting in the output file), such as Notepad or TextEdit. Type the following code into the file and save it on disk:

```
main = putStrLn out
 where
 out = "5! = " ++ show result
 result = fac 5

fac 0 = 1
fac n = n * fac (n - 1)
```

This code actually demonstrates a number of Haskell features, but you don't need to fully understand all of them now. To compile a Haskell application, you must have a main function, which consists of a single statement, which in this case is `putStrLn out`. The variable `out` is defined as part of the `where` clause as the con catenation of a string, `"5! = "`, and an integer, `result`, that you output using the `show` function. Notice the use of indentation. You must indent the code for it to compile correctly, which is actually the same use of indentation as found in Python.

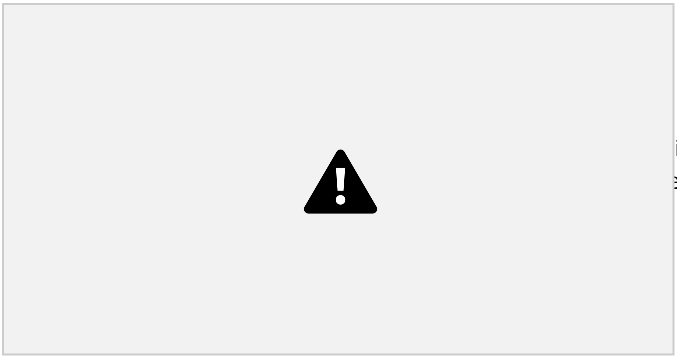
The code calculates the result by using the `fac` (factorial) function that appears below the `main` function. As you can see, Haskell makes it easy to use recursion. The first line defines the stopping point. When the input is equal to `0`, the function outputs a value of `1`. Otherwise, the second line is used to call `fac` recursively, with each succeeding call reducing the value of `n` by `1` until `n` reaches `0`.

After you save the file, you can open GHCi or WinGHCi to experiment with the application. The following steps provide the means to load, test, and compile the application:

1. **Type** :cd <*Source Code Directory*> **and press Enter.**

   Supply the location of the source code on your system. The location of your source code will likely differ from mine.

2. **Type** :load Simple.hs **and press Enter.**



in>, as shown in Figure 3-8. If you're
e ➪ Load menu command to accom

FIGURE 3-8: The prompt changes when you load a source file.

FIGURE 3-9: Executing the `main` function shows what the application file can do.

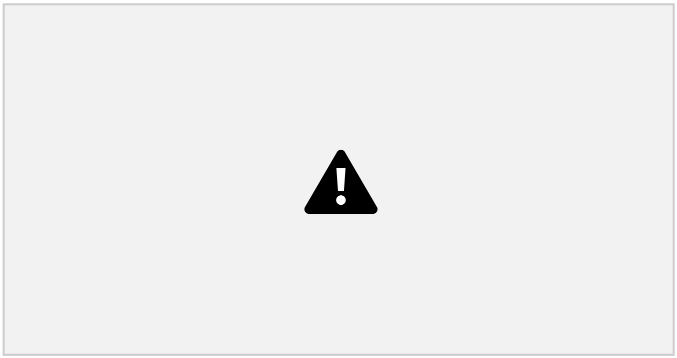
FIGURE 3-10: Compiling the loaded module creates an executable on disk.

3. Type :main **and press Enter.**

You see the output of the application as shown in Figure 3-9. When working with WinGHCi, you can also use the Actions ➪ Run "main" command or you can click the red button with the right-pointing arrow on the toolbar.

4. Type :! ghc --make ″Simple.hs″ **and press Enter.**

The interpreter now compiles the application, as shown in Figure 3-10. You see a new executable created in the source code directory. When working with WinGHCi, you can also use the Tools ➪ GHC Compiler menu command to perform this task. You can now execute the application at the command prompt and get the same results as you did in the interpreter.



5. Type :module **and press Enter.**

This act unloads all the existing modules. Notice that the prompt changes back to Prelude>. You can also perform this task using the Actions ➪ Clear Modules menu command.

6. Type :quit **and press Enter.**

The interpreter closes. You’re done working with Haskell for now.

These steps show just a small sample of the kinds of tasks you can perform using GHCi. As the book progresses, you see how to perform more tasks, but this is a good start on discovering what Haskell can do for you.

Using Haskell Libraries

Haskell has a huge library support base in which you can find all sorts of useful functions. Using library code is a time saver because libraries usually contain well-constructed and debugged code. The `import` function allows you to use external code. The following steps take you through a simple library usage example:

1. Open GHCi, if necessary.

2. Type import Data.Char **and press Enter.**

Note that the prompt changes to Prelude Data.Char> to show that the import is successful. The `Data.Char` library contains functions for working with the `Char` data type. You can see a listing of these functions at `http://hackage.haskell.org/package/base-4.11.1.0/docs/Data-Char.html`. In this case, the example uses the `ord` function to convert a character to its ASCII numeric representation.

3. **Type** ord('a') **and press Enter.**

   You see the output value of 97.

The "Getting and using datasets" section of Chapter 2 discusses how to obtain a dataset for use with Python. You can obtain these same datasets for Haskell, but first you need to perform a few tasks. The following steps will work for any plat form if you have installed Haskell using the procedure in the earlier part of this chapter.

1. **Open a command prompt or Terminal window with administrator privileges.**
2. **Type** cabal update **and press Enter.**

   You see the update process start. The cabal utility provides the means to perform updates in Haskell. The first thing you want to do is ensure that your copy of cabal is up to date.

3. **Type** cabal install Datasets **and press Enter.**

   You see a rather long list of download, install, and configure sequences. All these steps install the Datasets module documented at `https://hackage.haskell.org/package/datasets-0.2.5/docs/Numeric-Datasets.html` onto your system.

4. **Type** cabal list Datasets **and press Enter.**

   The cabal utility outputs the installed status of Datasets, along with other information. If you see that Datasets isn't installed, try the installation again by typing **cabal install Datasets --force-reinstalls** and pressing Enter instead.

Chapter 2 uses the Boston Housing dataset as a test, so this chapter will do the same. The following steps show how to load a copy of the Boston Housing dataset in Haskell.

1. **Open GHCi or WinGHCi.**
2. **Type** import Numeric.Datasets (getDataset) **and press Enter.**

   Notice that the prompt changes. In fact, it will change each time you load a new package. The step loads the `getDataset` function, which you need to load the Boston Housing dataset into memory.

3. **Type** import Numeric.Datasets.BostonHousing (bostonHousing) **and press Enter.**

   The `BostonHousing` package loads as `bostonHousing`. Loading the package doesn't load the dataset. It provides support for the dataset, but you still need to load the data.

4. **Type** bh <- getDataset bostonHousing **and press Enter.**

   This step loads the Boston Housing dataset into memory as the object `bh`. You can now access the data.

5. **Type** print (length bh) **and press Enter.**

   You see an output of `506`, which matches the length of the dataset in Chapter 2.

Getting Help with the Haskell Language

The documentation that the wizard installs as part of your Haskell setup is the first place you should look when you have questions. There are three separate files for answering questions about: GHC, GHC flags, and the Haskell libraries.

In addition, you see a link for HackageDB, which is the Haskell Software Reposi tory where you get packages such as Datasets used in the "Using Haskell Librar ies" section of this chapter. All these resources help you see the wealth of functionality that Haskell provides.

Tutorials make learning any language a lot easier. Fortunately, the Haskell com munity has created many tutorials that take different approaches to learning the language. You can see a listing of these tutorials at `https://wiki.haskell.org/Tutorials`.

No matter how adept you might be, documentation and tutorials won't be enough to solve every problem. With this in mind, you need access to the Haskell com munity. You can find many different groups online, each with people who are willing to answer questions. However, one of the better places to look for help is StackOverflow at `https://stackoverflow.com/search?q=haskell`.

Starti

ng

Functional Programming Tasks

Understand how functional programming differs
from other paradigms.

Discover uses for lambda calculus.

Use lambda calculus to perform practical

work. Perform basic tasks using lists and

strings.

» Examining declarations » Working

with functional data » Creating and

using functions

Chapter 4

Defining the Functional Difference

As described in Chapter 1 and explored in Chapters 2 and 3, using the func tional programming paradigm entails an approach to problems that differs from the paradigms that languages have relied on in the past. For one thing, the functional programming paradigm doesn't tie you to thinking about a problem as a machine would; instead, you use a mathematical approach that doesn't really care about how the machine solves the problem. As a result, you focus on the problem description rather than the solution. The difference means that you use *declarations* —formal or explicit statements describing the problem — instead of procedures — step-by-step problem solutions.

To make the functional paradigm work, the code must manage data differently than when using other paradigms. The fact that functions can occur in any order and at any time (allowing for parallel execution, among other things) means that functional languages can't allow mutable variables that maintain any sort of state or provide side effects. These limitations force developers to use better coding practices. After all, the use of side effects in coding is really a type of shortcut that can make the code harder to understand and manage, besides being far more prone to bugs and other reliability issues.

This chapter provides examples in both Haskell and Python to demonstrate the use of functions. You see extremely simple uses of functions in Chapters 2 and 3, but this chapter helps move you to the next level.

Comparing Declarations to Procedures

The term *declaration* has a number of meanings in computer science, and different people use the term in different ways at different times. For example, in the con text of a language such as C, a declaration is a language construct that defines the properties associated with an identifier. You see declarations used for defining all sorts of language constructs, such as types and enumerations. However, that's not how this book uses the term *declaration.* When making a declaration in this book, you're telling the underlying language to do something. For example, consider the following statement:

1. Make me a cup of tea!

The statement tells simply what to do, not how to do it. The declaration leaves the execution of the task to the party receiving it and infers that the party knows how to complete the task without additional aid. Most important, a declaration enables someone to perform the required task in multiple ways without ever changing the declaration. However, when using a procedure named `MakeMeTea` (the identifier associated with the procedure), you might use the following sequence instead:

1. Go to the kitchen.
2. Get out the teapot.
3. Add water to the teapot.
4. Bring the pot to a boil.
5. Get out a teacup.
6. Place a teabag in the teacup.
7. Pour hot water over the teabag and let steep for five minutes.
8. Remove the teabag from the cup.
9. Bring me the tea.

A *procedure* details what to do, when to do it, and how to do it. Nothing is left to chance and no knowledge is assumed on the part of the recipient. The steps appear in a specific order, and performing a step out of order will cause problems. For example, imagine pouring the hot water over the teabag before placing the teabag

in the cup. Procedures are often error prone and inflexible, but they do allow for precise control over the execution of a task. Even though making a declaration might seem to be superior to a procedure, using procedures does have advantages that you must consider when designing an application.

Declarations do suffer from another sort of inflexibility, however, in that they don't allow for interpretation. When making a declarative statement ("Make me a cup of tea!"), you can be sure that the recipient will bring a cup of tea and not a cup of coffee instead. However, when creating a procedure, you can add conditions that rely on state to affect output. For example, you might add a step to the pro cedure that checks the time of day. If it's evening, the recipient might return cof fee instead of tea, knowing that the requestor always drinks coffee in the evening based on the steps in the procedure. A procedure therefore offers flexibility in its capability to interpret conditions based on state and provide an alternative output.

Declarations are quite strict with regard to input. The example declaration says that a *cup* of tea is needed, not a pot or a mug of tea. The `MakeMeTea` procedure, however, can adapt to allow variable inputs, which further changes its behavior. You can allow two inputs, one called `size` and the other `beverage`. The `size` input can default to `cup` and the `beverage` input can default to `tea`, but you can still change the procedure's behavior by providing either or both inputs. The identifier, `MakeMeTea`, doesn't indicate anything other than the procedure's name. You can just as easily call it `MyBeverageMaker`.

One of the hardest issues in moving from imperative languages to functional lan guages is the concept of declaration. For a given input, a functional language will produce the same output and won't modify or use application state in any way. A declaration always serves a specific purpose and only that purpose.

The second hardest issue is the loss of control. The language decides how to per form tasks, not the developer. Yet, you sometimes see functional code where the developer tries to write it as a procedure, usually producing a less-than-desirable result (when the code runs at all).

Understanding How Data Works

Data is a representation of something — perhaps a value. However, it can just as easily represent a real-world object. The data itself is always abstract, and existing computer technology represents it as a number. Even a character is a number: The letter *A* is actually represented as the number 65. The letter is a value, and the num ber is the representation of that value: the data. The following sections discuss data with regard to how it functions within the functional programming paradigm.

Working with immutable data

Being able to change the content of a variable is problematic in many languages. The memory location used by the variable is important. If the data in a particular memory location changes, the value of the variable pointing to that memory loca tion changes as well. The concept of immutable data requires that specific mem ory locations remain untainted. All Haskell data is immutable.

Python data, on the other hand, isn't immutable in all cases. The "Passing by reference versus by value" section that appears later in the chapter gives you an example of this issue. When working with Python code, you can rely on the `id` function to help you determine when changes have occurred to variables. For example, in the following code, the output of the comparison between `id(x)` and `oldID` will be false.

```
x = 1
oldID = id(x)
x = x +1
id(x) == oldID
```

Every scenario has some caveats, and doing this with Python does as well. The `id` of a variable is always guaranteed unique except in certain circumstances:

» One variable goes out of scope and another is created in the same

location. » The application is using multiprocessing and the two variables exist on different processors.

» The interpreter in use doesn't follow the CPython approach to handling variables.

When working with other languages, you need to consider whether the data sup ported by that language is actually immutable and what set of events occurs when code tries to modify that data. In Haskell, modifications aren't possible, and in Python, you can detect changes, but not all languages support the functionality required to ensure that immutability is maintained.

Considering the role of state

Application *state* is a condition that occurs when the application performs tasks that modify global data. An application doesn't have state when using functional programming. The lack of state has the positive effect of ensuring that any call to a function will produce the same results for a given input every time, regardless of when the application calls the function. However, the lack of state has a