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

John Paul Mueller

Author of *Machine Learning For Dummies*

Functional Programming

by John Paul Mueller

for dummies®
A Wiley Brand

Functional Programming For Dummies®

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

The *functional programming 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. 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 programming paradigm involves. Of course, the key is knowing when functional programming 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 considerably easier to deal with. As part of considering the functional programming paradigm, 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 disadvantages 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 matter, 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 problems. This science uses lambda calculus as a basis, so an explanation of this particularly 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 introduction 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 interest (or not, as the case may be). This section briefly describes each icon in this book.


TIP

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.


WARNING

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.


TECHNICAL STUFF

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 programming 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 programming 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 something useful out of this book, so unless you have both languages installed, skipping 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 examples. 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.

1 Getting Started with Functional Programming

IN THIS PART . . .

Discover the functional programming paradigm.

Understand how functional programming differs.

Obtain and install Python.

Obtain and install Haskell.

IN THIS CHAPTER

- 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 programming 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 paradigms 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 (including 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 discover 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 purview of the individual languages.


REMEMBER

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 function 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.


REMEMBER

Functional programming gets rid of the interdependencies by replacing procedures with pure functions, which requires the use of immutable state. Consequently, 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 understands 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 capability 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 programming 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.


WARNING

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. Others 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. However, 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 programming paradigms (as described in the "Considering Other Programming Paradigms" 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 function in some languages.


TIP

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 functional programming paradigm explored in this book, but the world of development 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.


REMEMBER

The following sections discuss just four of these other paradigms. These paradigms 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 instructions that you can create and is mimicked in early languages, such as assembler.

Procedural

Procedural programming implements imperative programming, but adds functionality such as code blocks and procedures for breaking up the code. The compiler 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 applications appear to be one monolithic piece. Unless you were the application's developer, 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 relationship 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 paradigm, but the two paradigms are separate. Other paradigms, such as logic programming, 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

Using Functional Programming 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 programming 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 Functional 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 functions 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.

REMEMBER

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 programming 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 functional 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.


REMEMBER

One of the more common and less understood reasons for disqualifying a language 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 functions 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.

TIP

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).

WARNING

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.

IN THIS CHAPTER

- 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.

WARNING

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 suggest alternatives when you type the incorrect keyword, or it can tell you that a certain 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.*


TIP

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 insufficient 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.


TIP

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.