Pro Django

Copyright © 2009 by Marty Alchin

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1047-4

ISBN-13 (electronic): 978-1-4302-1048-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editor: Steve Anglin
Technical Reviewers: Jacob Kaplan-Moss, George Vilches
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editors: Liz Welch, Ami Knox
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Kinetic Publishing Services, LLC
Proofreader: April Eddy
Indexer: Julie Grady
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com, or visit http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com`. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

Understanding Django

Code alone isn't enough. Sure, it's what the computer runs, but code has to come from somewhere. A programmer has to sit down and decide what features to include, how they should be implemented, what other software to utilize and how to provide hooks for future enhancements to be added. It's easy to skip straight to code, ignoring the cognitive process that produces it, but great programmers always have reasons for the decisions they make.

With a framework, like Django, many such decisions have already been made, and the tools provided are shaped by these decisions and by the programmers who made them. By adopting these philosophies in your own code, not only will you be consistent with Django and other applications, but you may even be amazed at what you're able to accomplish.

Beneath even the most fundamental code is the thought process that went into its creation. Decisions were made about what it should do and how it should do it. This thought process is a step often overlooked in books and manuals, leading to an army of technicians slaving away, writing code that manages to accomplish the task at hand but without a vision for its future.

While the rest of this book will explain in detail the many basic building blocks Django provides for even the most complicated of projects, this chapter will focus on these even more fundamental aspects of the framework. For those readers coming from other backgrounds, the ideas presented in this chapter may seem considerably foreign, but that doesn't make them any less important. All programmers working with Python and Django would do well to have a solid understanding of the reasons Django works the way it does, and how those principles can be applied to other projects.

You may want to read this chapter more than once, and perhaps refer to it often as you work with Django. Many of the topics covered in this chapter are common knowledge in the Django community, so reading this chapter carefully is essential if you plan to interact with other programmers.

Philosophy

Django relies heavily on philosophy, both in how its code is written and how decisions are made about what goes into the framework. This isn't unique in programming, but it's something newcomers often have trouble with. It is essential to maintain both consistency and quality, and having a set of common philosophies to refer to when making decisions helps maintain both. Since these concepts are also important to individual applications, and even collections of applications, a firm grasp on these philosophies will yield similar benefits.

Perhaps the best-known and most-quoted passage of Python philosophy comes from Tim Peters, a longtime Python guru who wrote down many of the principles that guide Python's own development process. The 19 lines he came up with have been so influential to Python programmers over time that they are immortalized as Python Enhancement Proposal (PEP) 20[1] and in the Python distribution itself, as an "easter egg" module called `this`.

```
>>> import this
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

While some of this is clearly intended for humor, the majority is a good summation of many Python philosophies. The remainder of this chapter highlights some specific principles that are often cited within the Django community, but all professional Python programmers should keep this text in mind and reference it often.

One important thing to keep in mind is that many of the lines in the Zen of Python are subjective. For example, while "beautiful" may be better than "ugly," definitions of "beautiful" are plentiful and can vary as much as the people who provide them. Similarly, consider notions of simplicity and complexity, practicality and purity; each developer will have a different opinion on which side of the line a particular piece of code should be placed.

Django's Interpretation of the MVC Pattern

One of the most common application architectures—as adopted by hobbyists and corporations alike—is the Model-View-Controller (MVC) pattern, as it provides clean separation of tasks and responsibilities among the prominent aspects of an application. Django only loosely follows this model. A proper discussion should kick off with a quick overview of its components.

- The model is generally responsible for managing data and core business logic.
- The view displays that data to the user.
- The controller accepts user input and performs logic specific to the application.

1 http://prodjango.com/pep-20/

While this pattern has proven very effective in many domains, Django's authors weren't looking to conform to any form of pattern at the outset. They were simply interested in finding the most effective way to develop software for the Web. After all, Django was built for the daily needs of a working newspaper, where things have to happen very quickly if they're to happen at all. Ultimately, the separation of tasks into discrete groups serves a few different purposes.

- Code that is relegated to a specific set of tasks is much more maintainable, since it doesn't need to make assumptions about completely unrelated parts of the application.
- Application development is afforded additional flexibility, as multiple distinctly different view and controller layers may connect to a single model layer. This enables a variety of applications to share the same business logic and data, presenting it and interacting with it in different ways, for different audiences.
- Developers are able to learn just those parts of the system that are pertinent to the work being performed. This specialization helps to curb frustration and fatigue, while fostering creativity and excellence within each developer's domain of specialty.

There are certainly other smaller benefits, but these are generally the main goals achieved with the use of MVC. It's interesting to note, however, that the only part of those benefits that applies to any specific division in the MVC pattern is the ability to plug multiple applications into a single model layer. The rest is just an arbitrary division based on common development plans.

Django's developers sought these same benefits, but with an emphasis on rapid development, without worrying about creating a development pattern. After getting a set of tools that made sense for their workflow, they ended up with what some have called a Model-Template-View (MTV) pattern. However, there are really four primary code divisions in a Django application, which are outlined next.

Model

Given the benefit of keeping models apart from the rest of the application, Django follows that part of MVC to the letter. Django models provide easy access to an underlying data storage mechanism, and can also encapsulate any core business logic, which must always remain in effect, regardless of which application is using it.

Models exist independent of the rest of the system, and are designed to be used by any application that has access to them. In fact, the database manipulation methods that are available on model instances can be utilized even from the interactive interpreter, without loading a Web server or any application-specific logic.

Chapter 3 covers Django models in more detail, including how they're defined and utilized, how to include your own business logic and much more.

View

While they share a name with the original MVC definition, Django views have little else in common with the traditional paradigm. Instead, they combine some of the traditional view's responsibility with the entirety of the controller's tasks. A view accepts user input, including simple requests for information; behaves according to the application's interaction logic; and returns a display that is suitable for users to access the data represented by models.

Views are normally defined as standard Python functions that are called when a user requests a specific URL. In terms of the Web, even a simple request for information is considered an action, so views are intended to handle that alongside data modifications and other submissions. They can access the models, retrieving and updating information as necessary to accomplish the task requested by the user.

Since views are simply called as functions, without requiring any specific structure, they can be specified a number of ways. In addition to a simple function, a view could take the form of any Python callable, including instance methods, callable objects and curried or decorated functions.

Template

While views are technically responsible for presenting data to the user, the task of how that data is presented is generally relegated to templates, which are an important enough part of Django development to be considered a separate layer entirely. Many have drawn a parallel between Django templates and the traditional view layer, since templates handle all the presentational details the user will see.

Django provides a simple template language for this purpose, to ensure that template designers don't need to learn Python just to work with templates. Django's template language is also not dependent on any particular presentation language. It's primarily used for HTML but can be used to generate any text-based format.

Keep in mind, however, that this template engine is just one tool that views can use to render a display for a user. Many views may use HTTP redirects to other URLs, third-party Portable Document Format (PDF) libraries or anything else to generate their output.

URL Configuration

By combining this architectural philosophy with its nature as a framework for the Web, Django provides a separate layer of glue to make views available to the outside world via specific URLs. By supplying a regular expression as the URL component, a single declaration can accommodate a wide variety of specific URLs, in a highly readable and highly maintainable manner.

This configuration is defined separately from views themselves to allow a view to be configured at more than one URL, possibly with different options at each location. In fact, one of the core features of Django is the concept of generic views. These are views intended for common needs, with configuration options that allow them to be used in any application, requiring only a URL configuration to enable them.

Perhaps most important of all, having URLs as a separate part of the process encourages developers to think of URLs as part of an application's overall design. Since they must be used in bookmarks, blog posts and marketing campaigns, URLs are sometimes more visible than your application. After all, users who are paying attention while browsing the Web will see your URL before they even decide to visit your site. URLs get even more important when using print media for advertising campaigns.

Chapter 4 covers URL configurations in more detail, including some guidelines on proper URL design.

Loose Coupling

One key feature of the MVC architecture, and of Django's slightly modified form, is the notion that sections of code that perform significantly different functions shouldn't know how the others operate. This is called loose coupling. Contrast this with tight coupling, where modules often rely heavily on the internal details of other modules' implementations.

Tight coupling causes a whole host of problems with long-term code maintenance, as significant changes to one section will invariably affect others. This creates a mountain of extra work for the programmer, having to change code that has little—if anything—to do with the work that needs to be done. This extra work is unfortunate for not only the programmer; it's often quite costly for the employers as well.

It may seem like this principle would advocate that no code should ever know anything about any other code, but that's hardly the case, as a program written like that couldn't actually do anything. Some sections of code will always need to reference others; that's unavoidable. The key is to make sure that the details of one feature are hidden from the others.

In Python, loose coupling is typically provided in a number of ways, some of which are shown in the following list. There are countless others, which could fill a book on their own, but the techniques shown here are described in detail in Chapter 2.

- Duck-typing
- Operator overloading
- Signals and dispatching
- Plugins

Don't Repeat Yourself (DRY)

If you've been around the block a few times, you know all too well how easy it is to write "boilerplate" code. You code once for one purpose, then again for another, and again, and again and again. After a while, you realize how much code has been duplicated, and if you're lucky, you have the time, energy and presence of mind to look at what's common and move those pieces into a common location.

This process is one of the primary reasons for a framework to exist. Frameworks provide much of this common code, while attempting to make it easier to avoid duplicating your own code in the future. This combines to represent a common programming practice: Don't Repeat Yourself.

Often abbreviated DRY, this term comes up quite often in conversations, and can be used as

- A noun—"This code violates DRY."
- An adjective—"I like that approach, it's very DRY."
- A verb—"Let's try to DRY this up a bit."

The basic idea is that you should only write something once. It should be as reusable as possible, and if other code needs to know something about what you've already written, it should be able to get the necessary information automatically using Python, without requiring the programmer to repeat any of that information.

To facilitate this, Python provides a wealth of resources for peeking inside your code, a process called introspection. Many of these resources, covered in Chapter 2, are incredibly useful when supporting DRY in your code.

A Focus on Readability

"Readability counts." It's mentioned specifically in the Zen of Python, as noted earlier, and is perhaps one of the most important features of Python. Indeed, many Python programmers take pride in the readability of both the language and the code they write. The idea is that code is read far more often than it's written, especially in the world of open source.

To this end, Python as a language provides a number of features designed to improve readability. For instance, its minimal use of punctuation and forced indentation allow the language itself to help maintain the readability of your code. When you're working with code in the real world, however, there's far more to consider.

For real life, the Python community has developed a set of guidelines for writing code, intended to improve readability. Set forth in PEP-8,[2] these guidelines are designed to maintain not only readability of an individual program, but also consistency across multiple programs. Once you get the feel for one well-written program, you'll be able to easily understand others.

The exact details of PEP-8 are too numerous to list here, so be sure to read it thoroughly to get a good idea for how to write good code. Also, note that if you read Django's own source code, some of the rules set forth in PEP-8 aren't followed. Ironically, this is still in the interest of readability, as certain situations would suffer unnecessarily if every last rule was followed. After all, to quote the Zen of Python again, "Practicality beats purity." The examples in this book will follow the style used by Django's own source code.

Failing Loudly

"Errors should never pass silently. / Unless explicitly silenced." This may seem like a simple sentiment, but at two lines, it comprises over 10 percent of the Zen of Python, and there's something to be said for that. Dealing with exceptions is an important part of programming, and this is especially true in Python. While all programming languages can generate errors, and most have a way to handle them gracefully, each language has its own best practices for dealing with them.

One key to keep in mind is that, while the names of most Python exceptions end in `Error`, the base class is called `Exception`. To understand how they should be used and handled, it's useful to start by learning why that particular word was used. Looking at some of the dictionary definitions for the word "exception," it's easy to see variations on a theme.

- Something excepted; an instance or case not conforming to the general rule
- One that is excepted, especially a case that does not conform to a rule or generalization
- An instance that does not conform to a rule or generalization

Rather than an error, which describes a situation where a problem occurred, an exception is simply when something unexpected occurred. This may seem like a subtle distinction, but some philosophies treat exceptions as errors, reserving them solely for unrecoverable

2. `http://prodjango.com/pep-8/`

problems like corrupted files or network failure. This is reinforced by the fact that, in some languages, raising exceptions is extremely expensive, so to prevent performance problems, exceptions are avoided whenever possible.

In Python, however, exceptions are no more expensive than simple return values, allowing them to be more accurate to their dictionary definition. If we define an exception as a violation of a rule, it stands to reason that we must first define a rule.

Defining Rules

Since this is the most important aspect of understanding exceptions, it's necessary to be perfectly clear: there's no Python syntax for defining rules. It's simply not a feature of the language. Some other languages explicitly support design by contract,[3] and many can support it through framework-level code, but Python doesn't support any form of it natively.

Instead, rules are defined by programmers in what they intend their code to do. That may seem like an oversimplification, but it's really not. A piece of code does exactly what its author intends it to do, and nothing more. Anything outside the intentions of the programmer can—and should—be considered an exception. To illustrate this, here are some of the rules used by Python and Django:

- Accessing an item in a list using the bracket syntax (`my_list[3]`) returns the item at the specified position.
- A set's `discard()` method makes sure that a specified item is no longer a member of the set.
- A QuerySet's `get()` method returns exactly one object that matches the arguments provided.

Examples like these are important because even though these rules are simple, they accurately describe how the given features will behave in various situations. To further illustrate, consider the following scenarios and how the rule impacts behavior.

- If the index provided as a reference to a list item does exist, the appropriate value will be returned. If it doesn't, an exception (`IndexError`) is raised. If the value used as an index isn't an integer, a different exception (`TypeError`) is raised.
- If the item being removed from a set using `discard()` is already a member of the set, it's simply removed. If it wasn't a member of the set, `discard()` returns without raising an exception, since discard only tries to make sure that the item is not in the set.
- If the arguments passed to a QuerySet's `get()` method match one record in the database, that record is returned as an instance of the appropriate model. If no records match, an exception (`DoesNotExist`) is raised, but if more than one record matches, a different exception (`MultipleObjectsReturned`) is raised. Finally, if the arguments can't be used to query the database (due to incorrect types, unknown attribute names or a variety of other conditions), still another exception (`TypeError`) is raised.

Clearly, even simple rules can have profound effects, as long as they're defined explicitly. While the only requirement is that to be defined in the mind of the author, rules are of little

3. `http://prodjango.com/design-by-contract/`

use if not conveyed to anyone else. This becomes especially important in the case of a framework such as Django, built for distribution to the masses.

Documenting Rules

There are a number of appropriate ways to document the specific rules a piece of code was written to follow. It's even quite useful to specify them in more than one way, and in varying levels of complexity. There are four main places where people look for this information, so providing it in any or all of these locations would serve the purpose quite well.

- Documentation—As this should be the complete collection of information about the application, it stands to reason that these rules would be included.
- Docstrings—Regardless of stand-alone documentation, developers will often peek at the code itself to see how it works. Docstrings allow you to provide plain-text explanations of these rules right alongside the code that implements them.
- Tests—In addition to providing explanations of these rules for humans to understand, it's a great idea to provide them in a way that Python can understand. This allows your rule to be verified on a regular basis. In addition, by writing doctests—tests embedded inside docstrings—the tests are also human-readable, and both purposes can be served at once.
- Comments—Sometimes, a function may be complicated enough that a broad overview, such as might be found in full documentation or even the docstring, doesn't give sufficient information about what a particular chunk of code is expected to do. Python's emphasis on readability makes this fairly infrequent, but it does still happen. When it does, comments can be a useful way of explaining to others what the code is intended for, and thus what should be considered an exception.

Regardless of how you choose to describe your rules, there's one lesson that must always take precedence: be explicit. Remember, anything not laid out in your rule should be considered an exception, so defining the rule explicitly will help you decide how the code should behave in different situations, including when to raise exceptions.

Also, be consistent. Many classes and functions will look similar in name or interface, and where at all possible, they should behave similarly. Programmers who are accustomed to a particular behavior will expect similar behavior from similar components, and it's best to meet those expectations. This is especially true when writing code that mimics types provided by Python or Django, as they're already well-documented and well-understood by many programmers.

Community

Since being released to the public in 2005, Django has achieved great success, both technically and culturally. It has amassed a tremendous following throughout the world of Python Web development, among hobbyists and professionals alike. This community is one of the greatest assets to the framework and its users, and it's most certainly worth discussing in some detail.

AN EVOLVING COMMUNITY

It's important to realize that like any social structure, the Django community will evolve and change over time. As such, the information in this section may not always accurately reflect current practices and expectations.

There's no reason to let that deter you, though. The one thing I don't expect to change is the community's willingness to embrace new members. You'll always be able to get in touch with a variety of people, if you're willing to put yourself out there.

Management of the Framework

One of the first things to understand about development of Django—and about Python in general—is that, while the code for the framework is available for anyone to view and manipulate (it is open source, after all), the overall management of the core distribution is overseen by a small group of people. These "core developers" consist of those with access to update the main code repository.

WHAT IS "CORE"?

Because Django is open source, any user may make changes to Django's code and distribute those modified copies. Many developers have done so, adding significant features and enhancements and providing their work for others to use. Advanced users can make considerable alterations to the central code without impacting those who don't need the features provided by the copy.

In addition, developers are allowed—and encouraged—to make their applications generic and distribute them to others. These sometimes become so ubiquitous that many developers include them by default in any new project they start.

In contrast, Django's core is simply the code that is distributed through the main Django Web site, either as an official release or as the main trunk development code. So when a discussion includes a debate about whether something should be "in core," the dilemma is whether it should go into the official distribution or in some third-party format, such as a branch or a distributed application.

This structure helps ensure that those with the most experience with the framework and its history are responsible for looking over, and often tweaking, all patches before they are committed to the repository. They also regularly discuss issues concerning recent developments in the framework, major overhauls that need to be done, significant improvements that can be made and so on.

There is still someone at the top of the management chain. This position is called the Benevolent Dictator for Life, often abbreviated BDFL, and is reserved for those who have ultimate authority over all decisions, should they need to break a tie or override a majority decision. Thankfully, they are truly benevolent dictators, a distinction not taken lightly.

In fact, the idea of a BDFL is more humorous than anything else. Though they do hold ultimate authority, this power is rarely exercised, as they tend to favor group opinion. When they do need to step in and arbitrate a decision, their ruling is based on years of experience in knowing what's best for the framework and its audience. In fact, they will often raise their own

ideas to the group at large for discussion, possibly even deferring to the group if suitable counterarguments are raised.

The concept of a BDFL may seem foreign to those readers coming from corporate backgrounds, where design decisions are often made by committees, where majority rules and changes need to go through exhaustive bureaucratic processes. Instead, less direct oversight often leads to small groups of experts in different areas, who are quite capable of acting independently, producing high-quality code. This simple structure allows the process to run more quickly when it needs to, and, more importantly, helps maintain greater consistency within the framework.

In the Python world, Guido van Rossum, creator of Python itself, holds the position of BDFL. For Django, it's held by two people, each with the official title of co-BDFL: Adrian Holovaty, co-creator of the framework, and Jacob Kaplan-Moss, lead developer of the current work being done with Django. The principles and philosophies found throughout this chapter are generally reflections of the opinions and ideals of the BDFLs.

News and Resources

With a community as passionate and dynamic as Django's, it's important to keep up to date on what others are doing, what solutions they're finding to common problems, new applications that are available and many other things. Given the community's size and diversity, keeping up may seem like a daunting task, but it's really quite simple.

The first thing to keep an eye on is the Django weblog[4]—the official news outlet—which contains news and updates about the framework itself, its development and its use in major endeavors. For example, the Django weblog announces new releases, upcoming development sprints, updates to the project's Web site and more.

Perhaps more important is the Django community news aggregator,[5] which gathers articles from developers around the world, displaying them all in one place. The variety of information available here is much more diverse, as it's generated by community members, making it an extremely valuable resource. Example content could include new and updated applications, tips and tricks for solving common problems, new Django-powered Web sites and much more.

Reusable Applications

One of the most valuable aspects of Django is its focus on application-based development. Rather than building each site from scratch, developers are encouraged to write applications for specific purposes, and then combine them to build a site. This philosophy encourages many community members to release their applications to the public, as open source, so that others can benefit from their features and be a part of its success.

While developers are free to host their applications anywhere they wish, many choose Google Code,[6] the open source hosting service from Google. It uses Subversion, the same repository software as Django itself, which makes it easy to work with, and incorporates its

4. http://prodjango.com/django-weblog/
5. http://prodjango.com/community/
6. http://prodjango.com/google-code/

own issue tracking system. Many applications[7] are hosted there, so it's definitely a good idea to spend a few minutes looking around to see if someone has already written something you need.

After all, that's one of the primary goals of open source software: a larger community can produce better, cleaner, more functional code than a smaller group of dedicated programmers. The Django community both exhibits this behavior and encourages others to take advantage of it.

Getting Help

Despite all the knowledge contained in this and other books, it would be foolish to pretend that every potential situation can be documented ahead of time. What's more, the documentation that is available isn't always easy to find or to understand. In any of these cases, you may well find yourself needing to pose your situation to live people, with real-world experience, in hopes that someone can identify the problem and propose a solution.

The first thing to know is that this isn't a problem. Anyone can run into an unexpected situation, and even the best and brightest of us can get confounded by the simplest of syntax errors. If this happens to you, know that Django's community is very gentle, and you should definitely ask for help when you need it.

Read the Documentation

The first step when trying to resolve any problem is always to read the official documentation. It's quite thorough and updated regularly, as new features are added and existing behaviors are changed. When running into an error, the documentation will help ensure that you're using Django the way it's intended.

Once your code matches what the documentation shows to be appropriate, it's time to look at other common problems.

Check Your Version

As mentioned previously, the official documentation keeps up with Django's trunk development, so there's a definite possibility that the documented features don't match the features available in the code you're using. While this is more likely to occur if you're using an official release, it can still happen if you're tracking trunk, depending on how often you update your local copy.

When you're tracking trunk, the article on backwards-incompatible[8] changes should be considered an essential part of the official documentation. If you run into problems after updating, make sure that none of the features you're using have changed.

Frequently Asked Questions (FAQ)

After a few years of answering questions using the methods that follow, the Django community has heard a variety of questions that come up on a regular basis. To help answer these

7. http://prodjango.com/google-code-projects/
8. http://prodjango.com/backwards-incompatible-changes/

questions more easily, there are two articles. While the official FAQ[9] includes many questions not related to troubleshooting problems, there are still several common issues listed there.

The Internet Relay Chat (IRC) channel has its own set of questions and answers, and has its own FAQ.[10]

Mailing Lists

One of the easiest ways to get help is to ask your question on the django-users mailing list.[11] Because it operates over standard email, it's accessible to everyone, without requiring any special software. Simply join the list and you'll be able to post your questions for thousands of other users to look at. There are no guarantees, but most questions get answered quickly.

One key advantage of the mailing list is that all conversations are archived for future reference. In addition to the FAQs listed previously, the django-users mailing list archive can be an invaluable resource when you're trying to track down a problem that might have occurred to someone before.

IRC

If you need answers more quickly, the best option is the Django IRC channel,[12] where many knowledgeable members of the Django community are available for direct conversation. It's a very helpful environment, but you should be prepared to provide specific details about the problem. This may include the exact error traceback, snippets of the models, views and other code that might be involved with the problem.

This code is most often shared using an online "pastebin"—a place to put some code temporarily, for others to look at. Code can be pasted onto a public Web site for a limited time, allowing it to be shared with others. The Django community has its own pastebin called dpaste,[13] which is the recommended tool for sharing code with users on IRC.

Now What?

Of course, learning about philosophy and community doesn't get any code written. It helps to know how to put tools to good use, but that's nothing without a set of tools to work with. The next chapter outlines many of the less commonly used tools that Python itself has to offer, while the remaining chapters explore much of Django's own toolset.

9. http://prodjango.com/faq/
10. http://prodjango.com/irc-faq/
11. http://prodjango.com/django-users/
12. http://prodjango.com/irc/
13. http://prodjango.com/dpaste/