Solid Django

Release 0.1dev

Reinout van Rees

Jul 06, 2017

Contents

CHAPTER 1

Preface

Hi, I'm Reinout van Rees. You're now reading my book on Django, so I'll give you some background information on myself and the book, first. You're free to read the actual Django meat of the book, but this introduction might be fun, too.

I'll tell you a bit about *my background*, about the *the background of this book* and about the *book's structure*. I'll tell you how you can *pay me luxurious sums* for this wonderful piece of timeless prose and how to *give feedback*.

Reinout's background

By education, I'm a civil engineer. A military engineer builds missiles, a civil engineer builds targets. So if an architect designs something, we get to make sure it doesn't collapse. And if there's a big storm surge barrier or something like that: civil engineers.

By hobby and self-education, I'm a programmer. Pascal in the 1990s, Python since 2000. Mostly Zope and Plone in the beginning; Django since about three years. So I've got a quite diverse background, especially since I've delved into the deep, deep pits of Zope/Plone complexity. Lovely stuff. Learned a lot. Now I'm working full-time with Django. Lovely stuff!

By profession, I'm both a civil engineer and a programmer at the moment. I work at Nelen & Schuurmans in the Netherlands, a firm that does consultancy and IT development for water quality and water protection clients. So if you don't want water to flood your house and if you don't want your water to stink because of floating dead fish: give us a call. We'll give you a nice Django website.

By internet visibility, I'm pretty visible. At least in the Python/Django community, because my weblog at http://reinout.vanrees.org/weblog/ is mostly about Python and Django. I write down a lot of what I discover or build, but I also make summaries of each and every (Django) conference or Dutch Python/Django user group meeting I attend. That sure helps a lot.

By hobby, I am a recumbent cyclist and a medieval swordfighter. The first isn't of interest here, but the second one is. Medieval swordfighter, you say? Yes, that's relevant to the book. I'm picking a medieval theme as my main example, that's why.

Background of the book

Just a couple of weeks after 2011's Djangocon.eu conference I got an email from the pragmatic programmers whether I was interested to write a book about Django for them. I'm pretty sure that was because I just wrote a pretty decent summary of each and every talk on that year's conference. Liveblogging. Writing a book! Fun! An honour! But why are they asking me just when I'm moving to another house with my family?!?

Anyhow, I started writing the book and postponed quite some work on our new house. For about 9 months I was busy writing. They even brought in a co-author (thanks, Harry, for all you did!). But after 9 months is was clear that it was simply taking too much time. Abort abort abort: the book got canceled.

Rightly so, as it really took too much time. I think my writing style was out of sync with theirs. And the level of penmanship they require wasn't what I could manage. I *do* tend to be a tad verbose... If I *can* drag in another sub-subject, I *will* drag it in. Getting me in line took too much time.

After the cancellation I took time off. Finish the front garden, for instance. And I took a rest from the relentless book-writing pace. Writing a book really takes a lot of energy! I couldn't keep it up anymore so I took a well-deserved rest.

In november 2012, I restarted my book project. Django 1.5's christmas 2012 release date told me it was good to get a core part of the book finished by the same time. *Somebody* had to tell a good story about class based views, right? First get the core finished, then add the other necessary parts. and if there's interest, finish it all off with bonus content.

Book structure

The book is divided up into three parts.

- Django's foundation: an intro, models, views, templates, URLs. That's the absolute basic part of Django that you need to understand.
- Django's most common building blocks: what you'll use and customize most of the time when building your Django websites. Model queries, admin customization, packaging Django applications, javascript/css, security, editing/forms and hosting/webserver configuration.
- Extras that can be very handy. REST APIs, south automatic database migrations, Python intro for new programmers, debugging, management commands, context processors, buildout, nosql.

By the time you read this, the book won't be fully ready yet. I aim to finish the first part to co-incide with Django 1.5's christmas 2012 release. I'll finish the second part in the months after that.

The extras? Those are extra. Some of them I'll want to write for my own fun. Some of them I'll write if there's interest in my book (see below) :-)

Paying me ridiculous amounts of money

Why am I writing this book? Am I doing it for the money? Mostly not, but partially yes.

I can write. I've written a 250 page PhD thesis. I write a lot on my weblog. So writing is fun. And I've had the idea of writing a (second) book in my mind for some time. Being asked to write a Django book (see above) provided the spark to actually start!

Writing takes a ridiculous amount of time, though. Time I'm not spending cycling or swordfighting or fixing up the house or writing software. So a bit of compensation is in order. The goal that I have in mind is to try and earn 5000 Euro, enough to pay off my remaining study debt. How's that for a goal? (I don't yet know how it stands with taxes, so I might need to actually earn more to get my hands on 5000).

On the other hand, I'm not going to bother with DRM-protected ebooks. Or HTML pages behind a paywall. You can read the HTML for free and download the PDF right from my website. I *am* asking for a **donation**, though.

So... if this book is useful to you, giving me some money is a good way to say thanks. I'd say something between 10 and 20 Euro, more or less. Actually, anything is good, but that's my suggestion. I haven't yet given you my bank account details or paypal account, though. The reason? I haven't finished even the first part of the book yet. I don't want to accept money before I've finished with that!

Giving me feedback

I'm writing a book on programming and Django. So there's only one real choice for giving structured feedback: github. https://github.com/reinout/soliddjango . Open issues or give me pull requests.

If you've got generic feedback, it is probably easiest to send me a regular email at reinout@vanrees.org.

Anyway... have fun with the book!

Part 1: Django's foundation

CHAPTER 2

Getting started TODO

TODO

CHAPTER 3

Models and the admin interface TODO

TODO

CHAPTER 4

Working with views and templates TODO

TODO

CHAPTER 5

Controlling Django with URLs TODO

TODO

Part 2: Django’s common building blocks

CHAPTER 6

Labor division: python packaging TODO

TODO

CHAPTER 7

Advanced views and templates TODO

TODO

CHAPTER 8

Present yourself well: javascript and css TODO

TODO

CHAPTER 9

Secure your website: permissions TODO

TODO

CHAPTER 10

Editing and forms TODO

TODO

CHAPTER 11

Debugging and problem solving TODO

TODO

CHAPTER 12

Hosting and webserver integration TODO

TODO

Part 3: Django's extras

CHAPTER 13

Maintenance Tasks: Management Commands TODO

TODO

CHAPTER 14

Always available: context processors TODO

TODO

CHAPTER 15

Be a micro manager: middleware TODO

TODO

CHAPTER 16

Neighbor integration: REST TODO

TODO

CHAPTER 17

Buildout: more powerful deployment TODO

TODO

CHAPTER 18

Dig deep: full text search TODO

TODO

CHAPTER 19

Escape the relational confines: NoSQL TODO

TODO

Appendices

CHAPTER 20

Introducing Python for Django programmers

Understanding the basics of Python is critical for programming in Django. Fortunately, Python is easy to learn if you already know another language. This appendix gives you a high level overview to help you understand Django code if you're new to Python.

First I'll explain Python as an *interpreted and dynamically typed* language, followed by the main *data types* and *flow control* (loops and conditions). *Structure within Python files* and *structure between Python files* are next. To close off the chapter I look at *Python's philosophy*.

Interpreted and dynamically typed

You don't have to compile Python code every time you make a change. Python handles compiling and optimization behind the scenes. You can try out a Python statement, for example a print statement, and see the results immediately. Run `python` (or the full path to Python if necessary) on your commandline to display a Python prompt (which looks like >>>):

```
>>> print("Wow, a Python prompt!")
Wow, a Python prompt!
>>> exit()  # Or ctrl-d or ctrl-z.
```

All your regular Python code will be in files with the `.py` extension.

Python is a dynamically typed language, so it doesn't perform static type checks. You do not have to declare your variables or the *type* of your variables in Python. Assigning a variable is done with a single *equals* sign:

```
>>> greeting = "Hello, variable"
>>> print(greeting)
Hello, variable
```

Just assign variables and use them.

Data in Python

Python uses the same basic data types found in other languages. To program in Django, you need to understand the difference in how Python handles that data as opposed to other languages.

Numbers

When you use only whole numbers, Python treats the number as an *integer* data type. If you use a decimal, the type is a *float*.

Python handles division differently than many other languages. As long as there's at least one non-integer number, division works exactly as it would on any calculator in the world. However, when there are only integers, Python rounds the result to always return an integer.

Type this into your Python prompt for an example of Python's division:

```
>>> type(2)
<type 'int'>
>>> type(2.0)
<type 'float'>
>>> 3/2
1
>>> 3.0/2
1.5
>>> 3/2.0
1.5
```

As you can see in the code, 3 divided by 2 returns 1 when using only whole numbers. When you add a decimal point to either integer, the result is 1.5 (which is what we would normally expect). Remember, in Python, add a decimal to integers to save yourself from strange calculations.

Note: In Python 3, this slightly weird behaviour will be gone. `3 / 2` will just be `1.5` as you'd expect.

Strings

Strings (a number of characters) are surrounded by double or single quotes in Python. Either choice is fine. Here's an example:

```
>>> ''
''
>>> 'string'
'string'
>>> "a book's apostroph"
"a book's apostroph"
>>> 'a book\'s apostroph'
"a book's apostroph"
```

The last two examples show that you can use single quotes as-is inside a double quote string (and vice versa) or that you can escape them with a backslash.

There are two cases where you need something else than a regular string: for international characters and for regular expressions.

International characters: unicode strings

International characters are those characters outside of the ASCII characters range. Basically everything except a-z, A-Z, numbers and a few characters like `-/+:;`. If you use international characters (chinese signs, IKEA product names), you need to use unicode strings. Unicode is the world-wide standard for encoding *all* possible characters.

Django always gives you unicode strings when you retrieve strings from a web form or from a database, so you do not have to worry about the local encodings. To save yourself headaches, you should also always use unicode strings. Because if you don't, Python will have to do the conversion between your strings and unicode itself. And which encoding are your strings in? Does Python have to guess? Search online for unicodedecodeerror to give you an idea of the mess you can be in.

Python uses 7 bit ascii for its strings if you don't tell it otherwise. To get a unicode string, prefix the string with a `u`:

```
>>> 'regular string'
'regular string'
>>> u'unicode string'
u'unicode string'
```

The `u`-prefix is tedious to type all the time. Python 3 uses unicode strings by default (handy!). To get the same unicode-only behaviour, add a special import near the top of your files:

```
.. literalinclude:: /code/python/with_charset.py
```

When you want to use a non-ascii character in Python code, one way is with a unicode code:

```
>>> print(u'latin small letter e with diaeresis is \u00eb')
latin small letter e with diaeresis is ë
```

The last example shows one way to enter non-ASCII characters in Python code—a `\u` followed by the unicode code for the character (which you can find on the unicode website)_. `00eb` in this example is the code for an *ë*.

When you're writing Python files, it is easier to tell Python which character set to use, and let Python handle the conversion to unicode. You do this with a comment on the first line of the file:

```
# -*- coding: utf-8 -*-
print("The Dutch word for seas is zeeën.")
```

Many code editors also recognize the dash-star-dash pattern on the first line. This allows you to use non-ASCII characters in your editor instead of looking up the unicode code (which can be a hassle). The character set used most is `utf-8` (an 8-bit unicode encoding).

Regular expressions: raw strings

Django uses regular expressions in its URL configuration, see *Controlling Django with URLs TODO*. You use backslashes a lot in regular expression syntax. The problem is, Python also uses backslashes to give some characters in a string a special meaning. For examples:

```
>>> print("A string with two newlines.\n\nAnd a second line.")
A string with two newlines.

And a second line.
```

```
>>> print("And \ttabs \twork \t\talso.")
And     tabs    work            also.
>>> print("And also \b , though you won't see output.")
And also , though you won't see output.
```

You see here that Python treats `\n` and `\t` as newline and tab characters respectively. `\b` is the ancient bell (or beep) signal.

Here you have a problem. A `\b` means "whitespace at the start or end of a word" in a regular expression. To preserve the backslash in the string, you need to escape it with another backslash:

```
>>> print("This \b is not preserved")
This  is not preserved
>>> print("This \\b is properly preserved")
This \b is properly preserved
```

To retain a backslash in a Python string, you need to put in two backslashes. If you have an elaborate regular expression with lots of backslashes, all those double backslashes are error-prone and make the regular expression harder to read.

Python gives you a special kind of string that preserves all backslashes: a raw string. Just put an `r` in front of the string's quotes:

```
>>> print("The basic solution is a double backslash: \\b")
The basic solution is a double backslash: \b
>>> print(r"Alternative: a raw string. \b stays \b")
Alternative: a raw string. \b stays \b
```

Remember, you only need a raw string's special treatment of backslash characters for regular expressions.

Collections: lists, tuples and dictionaries

Python has the most common collection datastructures built in, including syntax that makes them easy to use.

Dictionary

A dictionary is a key/value mapping. It is often called a *hash table* in other languages. In Python, you create it by using curly braces:

```
>>> my_data = {'name': 'Reinout',
...            'city': 'Nieuwegein',
...            'country': 'The Netherlands'}
>>> my_data.keys()
['city', 'name', 'country']
>>> my_data['city']
'Nieuwegein'
>>> my_data['continent'] = 'Eurasia'
>>> my_data.keys()
['city', 'continent', 'name', 'country']
```

`my_data` in the example above starts out as a dictionary with three keys (name, city and country). You can access the values by asking for the key in square brackets.

You can always add additional items to a dictionary, like `my_data['continent'] = 'Eurasia'` as in the example. Note that a key's value can be whatever you want: a string, a class, even a list or another dict.

List

A list in Python is a modifiable list of values; you can sort it in-place and add or remove items. You write it with square brackets:

```
>>> my_kids = ['Rianne', 'Floris']
>>> my_kids.append('Elizabeth')
>>> my_kids
['Rianne', 'Floris', 'Elizabeth']
>>> my_kids[0]
'Rianne'
>>> my_kids[-1]
'Elizabeth'
```

Accessing items happens with square brackets just as it does with dictionaries. A list's index starts at zero, so `my_kids[0]` gives you the first kid. A negative index starts from the end, so `my_kids[-1]` gives you the last one.

You can change the list by appending or removing items (the latter sounds a bit harsh when you're talking about kids).

Tuple

A tuple is like a list, only immutable. Once created, it cannot be changed. This is handy for configuration; in a Django settings file, you'll see tuples rather than lists. If you want to add something, you need to create a new tuple. You create one by using regular parentheses and at least one comma:

```
>>> my_parents = ('Alie', 'Herman')
>>> my_parents[0]
'Alie'
```

Like lists, you access tuple items with an index between square brackets.

You must watch out with those parentheses that indicate a tuple. Parentheses are also used for grouping, like `(1 + 2) * 3`. What makes a tuple a tuple is that there is at least one comma between the parentheses. So `('reinout')` is the string `'reinout'`, but `('reinout',)` is a one-item tuple.

Boolean and nothing

`True` and `False` are Python's boolean values. `None` is used as *no value*.

In your Python code, you often want to test whether something is empty or whether something exists. For instance, *if* an address field is empty *then* print a warning. Python treats the following as False: `None`, an empty string, zero, an empty list, empty tuple, or an empty dictionary.

Flow control and indentation

To control data, Python has conditions and loops like other languages, but it also has *list comprehensions*, a friendly and modern way to work without using a loop. To use any of these, you first need to understand Python's indentation rules.

Indentation

The indentation in Python confuses many programmers when they are first learning the language. Most programming languages use something like curly braces to group statements, such as the blocks within an "if/else". Here is a JavaScript example:

```
if (kind === "2") {
    map_type = G_PHYSICAL_MAP;
} else {
    map_type = G_NORMAL_MAP;
}
```

You see the indentation in the JavaScript, but it's not mandatory. It just helps humans read the code. Python, on the other hand, makes the indentation mandatory. The beginning and end of a block of code isn't indicated by curly braces but by the start and end of indentation:

```
if kind == 2:
    map_type = G_PHYSICAL_MAP
else:
    map_type = G_NORMAL_MAP
```

This looks less cluttered. Since all Python code has the same indentation rules, reading code is easy and predictable. Python code should always be indented in steps of *four spaces*; never use tabs. Any good editor for Python will use four spaces because Python's style guide *strongly* recommends it.

Conditions

Python handles conditions with `if`. If you have more than one condition, you can add one or more `elif` statements. And `else` gives you a catch-all at the end. Here is an example:

```
if 2 == 3:
    print("equal")

if 2 != 3:
    print("not equal")

temperature = 3
if temperature == 0:
    print("Temperature is zero")
elif temperature < 0:
    print("Temperature is below zero")
else:
    print("Temperature is above zero")

story = {'prince': True,
         'princess': False,
         'horse_color': 'black'}
```

```
if story['prince'] and not story['horse_color'] == 'white':
    print("You need a white horse to rescue the princess.")

if story['prince'] or story['princess']:
    print("We have a prince or princess, so it is a fairy tale.")
```

== and != test for equality and inequality. Everything that results in a boolean value can be used as a condition. See also *Boolean and nothing*. You can combine conditions with `and` and `or` and negate with `not`.

Loops

Python has `for` and `while` loops. You'll almost exclusively see `for` loops:

```
for name in ['Reinout', 'Maurits']:
    print(name)

for i in range(10):
    print("Django")

for index, name in enumerate(['Reinout', 'Maurits']):
    print(index, name)
    # Prints '0 Reinout' and '1 Maurits'
```

Two useful tricks are `range` and `enumerate`. The first is for iterating a fixed number of times. `range(10)` produces `0, 1, 2, .., 9`. The second is for looping over a set of values and for numbering them. You recieve both an index (zero-based) and the actual value.

Dictionaries are common in Python, so you also often have to loop over the keys or the values (or both) of dictionaries:

```
cities = {'Nieuwegein': 'The Netherlands',
          'Utrecht': 'The Netherlands',
          'Ulmen': 'Germany',
          'Toronto': 'Canada'}
for city in cities:
    print(city)
    # Prints the key, so Nieuwegein, Utrecht, etc.

for city in cities.keys():
    print(city)
    # Also the keys.

for country in cities.values():
    print(country)
    # Prints the value, so Germany, Canada, etc.

for city, country in cities.items():
    print(city, country)
    # .items() returns both key and value.
```

If you loop over a dictionary without any methods, you really loop over the dictionary's keys, just like you would when using `.keys()`. Use `.values()` if you want to loop over the values instead. You may loop over both keys and values with the `.items()` method, this returns `(key, value)` tuples.

List comprehensions

You often write small loops to modify lists. You can loop over the list to remove empty items or calculate a new value for each of the list's items. Python has an alternative to writing these small loops: list comprehensions. With a list comprehension you can filter and/or modify a list in one line of code instead of using a loop to do the same work. The best way to show you is with an example:

```
some_file_text = """

Blank line above.
Here is some text

And some more

"""

lines = some_file_text.split('\n')
print(len(lines))  # 8 lines

# Filtering empty lines with a loop:
result = []
for line in lines:
    line = line.strip()  # Strip end-of-line and spaces.
    if line:
        result.append(line)

print(len(result))  # 3, three lines with actual text.

# Alternative: a list comprehension.
comprehension = [line for line in lines if line.strip()]
print(len(comprehension))  # Also 3.

# You can also modify a list:
uppercase = [line.upper() for line in comprehension]
print(uppercase[0])  # Returns BLANK LINE ABOVE.
```

The example takes a string with a couple of empty lines and filters out the empty lines. First, it uses a *for* loop by checking if a line is not empty and, if not, by appending it to the result. After that it does the same with a simple one-line list comprehension, which takes the form `[new for old in list if condition]`. Once you get used to the syntax, a list comprehension is much shorter and easier to read than a loop.

Structure within files

Within a single `.py` Python file, you can have variables and functions. Python is also an object oriented language, so you can have classes as well.

You are not required to use classes; simple variables and functions are fine. Django itself uses all three. Django models are always classes, a URL configuration uses only functions, and Django views can be either functions or classes.

Functions and arguments

Python functions are defined with `def` like this:

```python
def print_names():
    """Print names."""
    print('Reinout')
    print('Maurits')


def print_name(name):
    """Print(the name passed as argument."""
    print(name)


def print_default_name(name='Reinout'):
    """Print(the name (default is Reinout)."""
    print(name)


# You call them like this:
print_names()  # Prints Reinout and Maurits.
print_name('Maurits')  # Prints Maurits.
print_default_name()  # Prints Reinout.
print_default_name('Maurits')  # Prints Maurits.
```

The last two functions contain arguments. Python has two kinds of arguments: positional arguments and keyword arguments. A positional argument only has a name; a keyword argument has a name and default value.

Positional arguments are passed in exactly the order they are written. The position, literally, must match the order you want them passed. Positional arguments cannot be optional. When you have a limited number of arguments with a clear order, positional arguments work well.

In all other circumstances, you'll probably want to use keyword arguments. A keyword argument has the following advantages:

- Every keyword argument has a default value.
- The order in which the keyword arguments are passed doesn't matter. `your_method(a='aa', b='bb')` is the same as `your_method(b='bb', a='aa')`.
- Your functions are easier to evolve. If you decide to add a positional argument, you need to update all the places where you call your function. A keyword argument has a default value, which means you can leave most calls to your function alone.

For flexibility, keyword arguments are best. You should restrict positional arguments to those arguments that are absolutely essential to the function and will never change.

Classes

Python supports object oriented programming. You can define classes. Here is an example of defining, instantiating, and using a class:

```python
class Author(object):
    subject = 'Anything'

    def __init__(self, name):
        self.name = name

    def info(self):
        """Print author's name and subject."""
        print(self.name + ' writes about ' + self.subject)


class DjangoBookAuthor(Author):
    subject = 'Django'


# You use them like this:
author = Author('Rianne')
author2 = DjangoBookAuthor('Reinout')
author.info()
# Outputs 'Rianne writes about Anything'.
author2.info()
# Outputs 'Reinout writes about Django'.
print(author2.name)
# Outputs 'Reinout'.
```

The example shows two ways to create a class. Both use the `class` statement. The first way subclasses from Python's base `object` class. The second way subclasses from an existing class.

A class, like in any object oriented language, contains variables and functions. To be consistent with object oriented terminology, Python calls the variables in a class *attributes* and the functions *methods*.

You instantiate a class by calling it, like in the example. When you call the class, Python calls the class's specially-named `__init__()` method. If `__init__()` accepts arguments, you can pass them. In the example, you pass the name of the author as an argument when creating the class; this ends up as the `name` argument on the `__init__()` method.

By the way, every method must start with a mandatory first argument called a `self` argument. When you call a method on an object, Python automatically passes the object as this first argument.

Structure between files

Python files have the extension `py` and you can group them in directories (*packages* in Python-speak). You can use `py` files from the same or another package with *importing*.

Django is split up into many different packages by design because grouping similar code in cohesive packages helps keep Django's code neat and organized. You can do likewise with your own code by grouping related code into its own package.

Modules and packages

Python uses specific terminology for Python files and directories. A single Python file is a *module* and several modules grouped into a directory is a *package*.

Python does not treat every directory with modules as a package though, it wants you to explicitly mark it as a package by adding a `__init__.py` file to the directory. The file can be empty, though I always add a `# Package` comment in those otherwise-empty files, it can help you when you need to manually merge code from patches.

Packages can be nested by adding subdirectories. Each subdirectory should have its `__init__.py` to mark it as a package.

Importing modules and packages

Different Python files in different directories also means you need to be able to refer to them in some way so that you can use them. In Python this is called *importing*. You import modules and packages with the `import xyz` or `from abc import xyz` statement:

```
# Import a whole package or module.
import os
import os.path
# Import something specific from a module.
from os.path import exists


# Available because of import 'os' or 'os.path':
os.path.exists('.')
# Available because of directly importing it:
exists('.')
```

With the `import xyz` style you import a whole package or module with its full path. For instance, importing `os` makes everything inside that package available using the `os` name, like `os`, `os.path` and `os.path.exists`.

With the second style, `from abc import xyz`, you import something specific without needing to use the full path. In the example, you can just use `exists` because you imported it specifically; in this case you do not need to use the full `os.path.exists` path name.

In both cases, you use Python's *dotted path notation*. In this notation, every dot steps deeper into the package/module tree; for example `os.path.exists` calls the `exists` function in the `path` module in the `os` module.

The core philosophy of Python

Philosophy is build into Python. No really. And as I know you're not ready to believe me on my word yet: fire up Python and type `import this` at the prompt:

```
$ python
>>> import this
The Zen of Python, by Tim Peters

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!
```

Sure, there are some jokes in there, but most of the lines are for real and several are often quoted in Django discussions! Let me highlight three:

Explicit is better than implicit Sometimes it is handy to automatically make something happen. Add a specially-named file to your project and everything in there is automatically registered as a URL or so. The problem is that you need to know all the special magic rules.

Explicit is often clearer. Django's configuration file has a `ROOT_URLCONF` parameter, pointing at a specific Python file with a URLconf in it. This itself is an explicit list of which URLs match what. There can be no confusion over which URL matches what because everything is explicit.

Readability counts Simply keeping this one in mind will help you a lot. Single-character variable names are no good. Don't abbreviate too much. Use naming to help you. You yourself will read your own code again in half a year's time: can you still understand it then? Does `dh = sgst * 1.4` still make sense then? Would `dike_height = suggested_height * SAFETY_FACTOR` be clearer? Don't skimp on names. They don't have an 8-character restriction.

And look at Python's use of indentation. Couple that with *flat is better than nested*. If you go too deep and use too many loops and if statements, Python's mandatory indentation will show you that you went too far. Python helps you to keep your code clean this way. It comes with a build-in suggestion to split up your code a bit.

Now is better than never A good example is Django's configuration. It is simply a Python file, so you can do everything you want with it. Import other files, read in configuration from a text file, you name it. It is not the most elegant solution, but when Django got made, they needed a pragmatic solution.

Why spend years arguing? Why sink a lot of effort in finding something that can take all use cases? You'll be waiting forever. A good enough solution now is preferable to something perfect that might never come.

What we learned

We learned the most common Python language elements. Enough to get started with Django and enough to understand the example code througout this book.

For exercises, I'll keep it simple: learn more about Python. Try it out. The best place to start is dive into Python, it will take you through Python in a gentle way. For lots of exercises, look at learn Python the hard way.

CHAPTER 21

Full table of contents