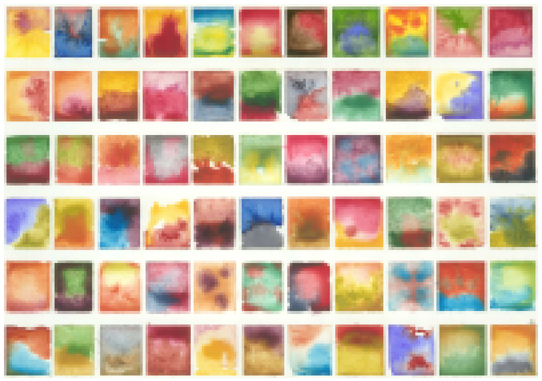


Python 3 Patterns & Idioms book

Release 1.0

Bruce Eckel

November 25, 2008

CONTENTS

1 Contributors **1**
- 1.1 Thanks To 1

2 ToDo List **3**

3 The remainder are from context, from the book. **5**

4 A Note To Readers **7**

5 Introduction **9**
- 5.1 A Team Effort 9
- 5.2 Not an Introductory Book 9
- 5.3 The License 10
- 5.4 The Printed Book 10
- 5.5 Translations 10
- 5.6 My Motives 10

6 Teaching Support **13**

7 Book Development Rules **15**
- 7.1 Contribute What You Can 15
- 7.2 Don’t Get Attached 15
- 7.3 Credit 16
- 7.4 Mechanics 16
- 7.5 Diagrams 16

8 Developer Guide **19**
- 8.1 Getting Started: The Easiest Approach 19
- 8.2 For Windows Users 19
- 8.3 Installing Sphinx 20
- 8.4 Getting the Development Branch of the Book 20
- 8.5 Building the Book 20
- 8.6 Building the PDF 21
- 8.7 Working with BitBucket and Mercurial 21
- 8.8 A Simple Overview Of Editing and Merging 22

9 Part I: Foundations **23**

10 Python 3 Language Changes **25**

11 Decorators **27**
11.1 Decorators vs. the Decorator Pattern . . . 27
11.2 History of Macros . . . 27
11.3 The Goal of Macros . . . 28
11.4 What Can You Do With Decorators? . . . 28
11.5 Function Decorators . . . 28
11.6 Slightly More Useful . . . 30
11.7 Using Functions as Decorators . . . 30
11.8 Review: Decorators without Arguments . . . 31
11.9 Decorators with Arguments . . . 32
11.10 Decorator Functions with Decorator Arguments . . . 34
11.11 Further Reading . . . 35

12 Generators, Iterators, and Itertools **37**

13 Comprehensions **39**
13.1 A More Complex Example . . . 39

14 Coroutines & Concurrency **43**
14.1 Further Reading . . . 43

15 Part II: Idioms **45**

16 Discovering the Details About Your Platform **47**

17 A Canonical Form for Command-Line Programs **49**

18 Messenger/Data Transfer Object **51**

19 Part III: Patterns **53**

20 Indices and tables **55**

Index **57**

CHAPTER

ONE

CONTRIBUTORS

List of contributors.

Note: This needs some thought. I want to include everyone who makes a contribution, but I'd also like to indicate people who have made larger contributions – there's no automatic way to do this now that we have moved to BitBucket and are using Wikis to allow people to make contributions more simply in the beginning.

1.1 Thanks To

- BitBucket.org and the creators of Mercurial
- Creator(s) of Sphinx
- And of course, Guido and the team for their incessant improvement of Python, especially for taking the risk in breaking backward compatibility in Python 3.0 to refactor the language.

Todo

Yarko (example label of ToDo):

- update CSS styles for todo's & todo lists;
- look at http://sphinx.pocoo.org/ext/coverage.html for example.
- Autogenerated ToDoLists do not appear in LaTeX output - debug, fix;
- DONE: - ToDo does not appear to be created by make dependencies (it's autogenerated); - update Makefile to always re-generate todo lists;

CHAPTER

TWO

TODO LIST

Currently, this doesn't seem to link into the index, as I'd hoped.

- Refine "Printed Book" and "Translations"
- Code extractor for rst files (maybe part of intro chapter?)
- Code updater to put code in/refresh code into book.
- Move frontmatter into its own directory
- <!> Seems to be a warning sign (but initial tests didn't work)
- Idea: decorator on a dictionary object, to turn it into an ordered dictionary.
- "Other resources" at the end of each chapter
- For print version, convert hyperlinks into footnotes.
  - build tool for this, or check int rst handling of this - see if it works with Sphinx;

CHAPTER

THREE

THE REMAINDER ARE FROM CONTEXT, FROM THE BOOK.

Todo

Yarko (example label of ToDo):

- update CSS styles for todo's & todo lists;
- look at http://sphinx.pocoo.org/ext/coverage.html for example.
- Autogenerated ToDoLists do not appear in LaTeX output - debug, fix;
- DONE: - ToDo does not appear to be created by make dependencies (it's autogenerated); - update Makefile to always re-generate todo lists;

(The original entry is located in Contributors.rst, line 27 and can be found *here*.)

Todo

The remainder of this document needs rewriting. Rewrite this section for BitBucket & Mercurial; make some project specific diagrams;

(The original entry is located in DeveloperGuide.rst, line 135 and can be found *here*.)

Todo

Add additional steps here.

(The original entry is located in DeveloperGuide.rst, line 144 and can be found *here*.)

Todo

This section still work in progress:

- `hg branch lp:python3patterns`
- `hg commit -m 'initial checkout'`
- (hack, hack, hack....)
- `hg merge` (pull new updates)
- `hg commit -m 'checkin after merge...'`
- ... and so on...

(The original entry is located in DeveloperGuide.rst, line 164 and can be found *here*.)

CHAPTER

FOUR

A NOTE TO READERS

What you see here is an early version of the book. We have yet to get everything working right and rewritten for Python. Or even to get the book testing systems in place.

If you're here because you're curious, that's great. But please don't expect too much from the book just yet. When we get it to a point where everything compiles and all the Java references, etc. have been rewritten out, then this note will disappear. Until then, *caveat emptor*.

CHAPTER

FIVE

INTRODUCTION

5.1 A Team Effort

This book is an experiment to see if we can combine everyone's best efforts to create something great.

You can find the contributors right before this introduction. They are listed in order of *Karma Points*, a system Launchpad.net uses to keep track of contributions from everyone working on an open-source project.

In my case, I will write new material, as well as rewriting other contributions to clarify and give voice, setting up the architecture and being the *Benevolent Dictator* for the book. But I definitely won't be doing everything; my goal is that this is a team project and that everyone who wants to will have something useful to contribute.

We'll be using Launchpad.net's "Blueprints" facility to add "features" to the book, so that's where you'll find the main repository of things to do.

What can you contribute? Anything as small as spelling and grammatical correctons, and as large as a whole chapter. Research into new topics and creating examples is what takes me the most time, so if you know something already or are willing to figure it out, don't worry if your writing or programming isn't perfect – contribute it and I and the rest of the group will improve it.

You may also have talents in figuring things out. Sphinx formatting, for example, or how to produce camera-ready formatting. These are all very useful things which will not only benefit this book but also any future book built on this template (every bit of the build system for the book will be out in the open, so if you want to take what we've done here and start your own book, you can).

Of course, not everything can make it into the final print book, but things that don't fit into the main book can be moved into an "appendix book" or a "volume 2" book or something like that.

5.2 Not an Introductory Book

Although there is an introduction for programmers, this book is not intended to be introductory. There are already lots of good introductory books out there.

You can think of it as an "intermediate" or "somewhat advanced" book, but the "somewhat" modifier is very important here. Because it is not introductory, two difficult constraints are removed.

1. In an introductory book you are forced to describe everything in lock step, never mentioning anything before it has been thoroughly introduced. That's still a good goal, but we don't have to agonize over it when it doesn't happen (just cross-reference the material).
2. In addition, the topics are not restricted; in this book topics are chosen based on whether they are interesting and/or useful, not on whether they are introductory or not.

That said, people will still be coming to a topic without knowing about it and it will need to be introduced, as much as possible, as if they have never seen it before.

5.3 The License

Unless otherwise specified, the material in this book is published under a Creative Commons Attribution-Share Alike 3.0 license.

If you make contributions, you must own the rights to your material and be able to place them under this license. Please don't contribute something unless you are sure this is the case (read your company's employment contract – these often specify that anything you think of or create at any time of day or night belongs to the company).

5.4 The Printed Book

Because of the creative commons license, the electronic version of the book as well as all the sources are available, can be reproduced on other web sites, etc. (again, as long as you attribute it).

You can print your own version of the book. I will be creating a printed version of the book for sale, with a nice cover and binding. Many people do like to have a print version of the book, and part of the motivation for doing a print version is to make some income off the effort I put into the book.

But buying my particular print version of the book is optional. All of the tools will be downloadable so that you can print it yourself, or send it to a copy shop and have it bound, etc. The only thing you won't get is my cover and binding.

5.5 Translations

Launchpad.net, where this project is hosted, has support for doing translations and this gave me an idea. I had just come back from speaking at the Python conference in Brazil, and was thinking about the user group there and how I might support them. (We had done a seminar while I was there in order to help pay for my trip and support the organization).

If the book can be kept in a Sphinx restructured text format that can be turned directly into camera-ready PDF (the basic form is there but it will take somebody messing about with it to get it into camera-ready layout), then the job of translation can be kept to something that could be done by user groups during sprints. The user group could then use a print-on-demand service to print the book, and get the group members to take them to local bookstores and do other kinds of promotions. The profits from the book could go to the user group (who knows, just like the Brazillian group, your group may end up using some of those profits to bring me to speak at your conference!).

If possible, I would like a royalty from these translations. To me, 5% of the cover price sounds reasonable. If the user group would like to use my cover, then they could pay this royalty. If they wanted to go their own way, it's creative commons so as long as the book is attributed that's their choice.

5.6 My Motives

Just so it's clear, I have the following motives for creating this book:

1. Learn more about Python and contribute to the Python community, to help create more and better Python programmers.
2. Develop more Python consulting and training clients through the publicity generated by the book (see here).
3. Experiment with group creation of teaching materials for the book, which will benefit me in my own training (see the previous point) but will also benefit anyone choosing to use the book as a text in a course or training seminar. (See *Teaching Support*).
4. Generate profits by selling printed books. (But see above about the ability to print the book yourself).
5. Help raise money for non-U.S. Python user groups via translations, from which I might gain a small percentage.

CHAPTER

SIX

TEACHING SUPPORT

Teachers and lecturers often need support material to help them use a book for teaching. I have put some exercises in, and I hope we can add more.

I'd also like to create teaching materials like slides and exercises as a group effort with a creative commons license, especially by allying with those people who are in the teaching professions. I'd like to make it something that everyone would like to teach from – including myself.

Here are some places to get ideas for exercises and projects:

- For coding dojos
- Rosetta Code

CHAPTER

SEVEN

BOOK DEVELOPMENT RULES

Guidelines for the creation process.

Note: This is just a start. This document will evolve as more issues appear.

7.1 Contribute What You Can

One of the things I've learned by holding open-spaces conferences is that everyone has something useful to contribute, although they often don't know it.

Maybe you're don't feel expert enough at Python to contribute anything yet. But you're in this field, so you've got some useful abilities.

- Maybe you're good at admin stuff, figuring out how things work and configuring things.
- If you have a flair for design and can figure out Sphinx templating, that will be useful.
- Perhaps you are good at Latex. We need to figure out how to format the PDF version of the book from the Sphinx sources, so that we can produce the print version of the book without going through hand work in a layout program.
- You probably have ideas about things you'd like to understand, that haven't been covered elsewhere.
- And sometimes people are particularly good at spotting typos.

7.2 Don't Get Attached

Writing is rewriting. Your stuff will get rewritten, probably multiple times. This makes it better for the reader. It doesn't mean it was bad. Everything can be made better.

By the same measure, don't worry if your examples or prose aren't perfect. Don't let that keep you from contributing them early. They'll get rewritten anyway, so don't worry too much – do the best you can, but don't get blocked.

There's also an editor who will be rewriting for clarity and active voice. He doesn't necessarily understand the technology but he will still be able to improve the flow. If you discover he has introduced technical errors in the prose, then feel free to fix it but try to maintain any clarity that he has added.

If something is moved to Volume 2 or the electronic-only appendices, it's just a decision about the material, not about you.

7.3 Credit

As much as possible, I want to give credit to contributions. Much of this will be taken care of automatically by the Launchpad.net "Karma" system. However, if you contribute something significant, for example the bulk of a new chapter, then you should put "contributed by" at the beginning of that chapter, and if you make significant improvements and changes to a chapter you should say "further contributions by" or "further changes by", accordingly.

7.4 Mechanics

- Automate everything. Everything should be in the build script; nothing should be done by hand.
- All documents will be in Sphinx restructured text format. Here's the link to the Sphinx documentation.
- Everything goes through Launchpad.net and uses Launchpad's Bazzar distributed version control system.
- Follow PEP8 for style. That way we don't have to argue about it.
- Camelcasing for naming. PEP8 suggests underscores as a preference rather than a hard-and fast rule, and camelcasing *feels* more like OO to me, as if we are emphasizing the design here (which I want to do) and putting less focus on the C-ish nature that *can* be expressed in Python.

```
The above point is still being debated.
```

- Four space indents.
- We're not using chapter numbers because we'll be moving chapters around. If you need to cross-reference a chapter, use the chapter name and a link.
- Index as you go. Indexing will happen throughout the project. Although finalizing the index is a task in itself, it will be very helpful if everyone adds index entries anytime they occur to you. You can find example index entries by going to the index, clicking on one of the entries, then selecting "view source" in the left-side bar (Sphinx cleverly shows you the Sphinx source so you can use it as an example).
- Don't worry about chapter length. Some chapters may be very small, others may be quite significant. It's just the nature of this book. Trying to make the chapters the same length will end up fluffing some up which will not benefit the reader. Make the chapters however long they need to be, but no longer.

7.5 Diagrams

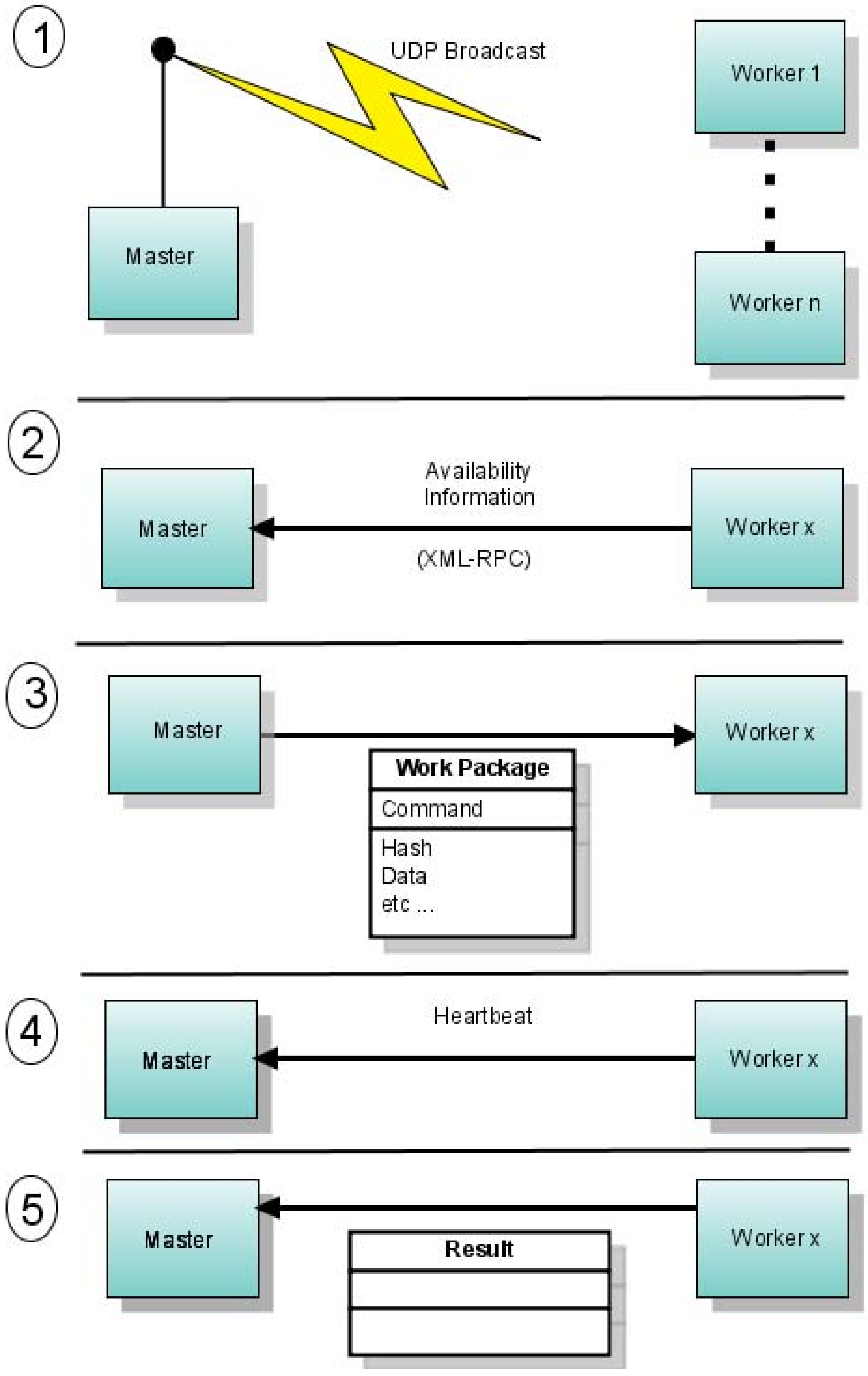
Create diagrams using whatever tool is convenient for you, as long as it produces formats that Sphinx can use.

It doesn't matter if your diagram is imperfect. Even if you just sketch something by hand and scan it in, it will help readers visualize what's going on.

At some point, diagrams will be redone for consistency using a single tool, with print publication in mind. This tool may be a commercial product. However, if you need to change the diagram you can replace it with your new version using your tool of choice. The important thing is to get the diagram right; at some point it will be redone to look good.

Note that all image tags should use a `*` at the end, not the file extension name. For example `..image:: _images/foo.*`. This way the tag will work for both the HTML output and the Latex output. Also, all images should be placed in the `_images` directory.

Here's an example which was done with the free online service Gliffy.com, then modified using the free Windows program Paint.NET (note, however, that we should not use color because it won't translate well to the print book):


1
UDP Broadcast
Worker 1
Master
Worker n
2
Availability Information
Master
(XML-RPC)
Worker x
3
Master
Worker x
Work Package
Command
Hash
Data
etc ...
4
Heartbeat
Master
Worker x
5
Master
Worker x
Result

CHAPTER

EIGHT

DEVELOPER GUIDE

Details for people participating in the book development process.

8.1 Getting Started: The Easiest Approach

If all of the details are a little overwhelming at first, there's an easy way for you to make contributions without learning about distributed version control and Sphinx:

1. Create an account at http://www.BitBucket.org.
2. In your account, you get a wiki. In your wiki, create a page for your contribution. Just add your code and descriptions using plain text.
3. Point us to your wiki via the newsgroup.
4. We'll take your contribution and do the necessary formatting.

If you want to take another step, you can learn how to format your wiki page using Sphinx by looking at the source for pages in the book. You can try it right now – on the left side of this page (in the HTML book) you'll see a header that says **This Page** and underneath it **Show Source**. Click on **Show Source** and you'll see the Sphinx source for this page. Just look at the page sources and imitate that.

When you're ready, you can learn more about Sphinx and Mercurial and begin making contributions that way.

The following sections are for those who are ready to build the book on their own machines.

8.2 For Windows Users

You need to install Cygwin; go to:

> http://www.cygwin.com

You need to install at least the `make` utility, but I find that `chere` (command prompt here) is also very useful.

Also install `openssh` (under **Net**), so you can create your RSA key for Mercurial.

I've discovered that it's best if you *don't* install Python as part of Cygwin; instead use a single Python installation under windows. Cygwin will find the installation if it is on your Windows PATH.

Because of this, you shouldn't select "mercurial" when you're installing Cygwin because that will cause Python to be installed. Instead, install them as standalone Windows applications (see below).

8.3 Installing Sphinx

Because we are sometimes pushing the boundaries of Sphinx, you'll need to get the very latest development version (a.k.a. the "tip").

1. Get mercurial:

   http://www.selenic.com/mercurial
   Avoid installing the tortoiseHG part - it has caused trouble w/ Python debuggers.

2. To get the Sphinx trunk, start with:

   ```
   $ hg clone http://www.bitbucket.org/birkenfeld/sphinx/
   ```
   and to update, use:
   ```
   $ hg pull
   ```
   Once you update, run
   ```
   $ python setup.py install
   ```
   (You can repeat this step whenever you need to update).
   We may talk about minimum version numbers to process the book. Check your version with:
   ```
   $ hg identify -n
   ```

The full anouncement from Georg (Sphinx creator) is here:

http://groups.google.com/group/sphinx-dev/browse_thread/thread/6dd415847e5cbf7c

Mercurial Cheat sheets & quick starts should be enough to answer your questions:

- http://edong.net/2008v1/docs/dongwoo-Hg-120dpi.png
- http://www.ivy.fr/mercurial/ref/v1.0/

8.4 Getting the Development Branch of the Book

This book uses BitBucket.org tools, and additional tools if necessary.

1. Sign up for an account at http://BitBucket.org.
2. You must create an rsa key. Under OSX and Linux, and if you installed `openssh with` Cygwin under windows, you run `ssh-keygen` to generate the key, and then add it to your BitBucket account.
3. Go to http://www.bitbucket.org/BruceEckel/python-3-patterns-idioms/, and you'll see instructions for getting a branch for development.
4. Work on your branch and make local commits and commits to your BitBucket account.

8.5 Building the Book

To ensure you have Cygwin installed correctly (if you're using windows) and to see what the options are, type:

```
make
```

at a shell prompt. Then you can use `make html` to build the HTML version of the book, or `make htmlhelp` to make the windows help version, etc.

You can also use the `build` system I've created (as a book example; it is part of the distribution). This will call `make` and it simplifies many of the tasks involved. Type:

```
build help
```

to see the options.

Todo

The remainder of this document needs rewriting. Rewrite this section for BitBucket & Mercurial; make some project specific diagrams;

8.6 Building the PDF

In order to build the Acrobat PDF verion of the book, you must install some additional software:

Todo

Add additional steps here.

8.7 Working with BitBucket and Mercurial

Note: Adapted from a posting by Yarko Tymciurak

This assumes that you have created a local branch on your private machine where you do work, and keep it merged with the trunk.

That is, you've done:

- Forked a branch of http://www.bitbucket.org/BruceEckel/python-3-patterns-idioms/ (the main trunk; this fork will provide a place for review and comment)
- cloned the trunk to your local machine: - hg clone https://my_login@bitbucket.org/BruceEckel/python-3-patterns-idioms/
- cloned your local copy of trunk to create a working directory: - hg clone python-3-patterns-idioms devel

Todo

This section still work in progress:

- `hg branch lp:python3patterns`
- `hg commit -m 'initial checkout'`
- (hack, hack, hack....)
- `hg merge` (pull new updates)
- `hg commit -m 'checkin after merge...'`

- ... and so on...

When you have a new function idea, or think you've found a bug, ask Bruce on the group.

- If you have a new feature, create a wiki page on BitBucket and describe what you're going to do.
- If you have found a bug, make a bug report on BitBucket (later assign it to yourself, and link your branch to it);
- If you want to work on a project, look for an unassigned bug and try to work it out - then proceed as below...

When you are ready to share your work have others review, register a branch.

Note: You can re-use one branch for multiple bug fixes.

1. Sign up for an account on BitBucket.org
2. Go to the project and select "register branch" (`https://code.BitBucket.org/python3patterns/+addbranch` Suggest you create a hosted branch, then you can work locally, and pull/push as you make progress (see http://doc.Mercurial-vcs.org/latest/en/user-guide/index.html#organizing).
3. Once you have registered your branch, BitBucket will provide you with instructions on how to pull and push to your personal development copy.
4. Link your bug report or blueprint to your branch.
5. Merge from your "parent" (the trunk, or others you are working with) as needed.
6. Push your working copy to BitBucket as your work is ready for others to review or test.
7. Once you are done making your changes, have completed testing, and are ready for the project team to inspect & test, please select "propose for merging"
8. Somebody on the core team will make a test merge (it may include merging with other patches). Once tests pass, and your branch is accepted, it will be merged into the trunk.

8.8 A Simple Overview Of Editing and Merging

1. `hg pull http://www.bitbucket.org/BruceEckel/python-3-patterns-idioms/`
2. `hg merge` This brought up kdiff3 (note: this requires a separate installation of **kdiff3**)on any file's w/ conflicts, and you get to just visually look - left-to-right at A:base, B:mine, and C:yours.... the NICE thing is when you want BOTH the other and yours, you can click BOTH B & C buttons — sweeet! you can also review the "automatic" merges, choose which - conflicts only, or any merge.
3. ... `make html; make latex` ...... look at outputs (simultaneously, comparatively)... make any changes.... repeat....
4. `hg ci` without a message, it brought up an editor with a list of all changed files - so you can comment individually.

CHAPTER

NINE

PART I: FOUNDATIONS

CHAPTER

TEN

PYTHON 3 LANGUAGE CHANGES

Covers language features that don't require their own chapters.

Note: If a section in this chapter grows too large it may require its own chapter.

CHAPTER

ELEVEN

DECORATORS

Note: This chapter is a work in progress; it's probably better if you don't begin making changes until I've finished the original version, which is being posted as a series on my weblog.

This amazing feature appeared in the language almost apologetically and with concern that it might not be that useful.

I predict that in time it will be seen as one of the more powerful features in the language. The problem is that all the introductions to decorators that I have seen have been rather confusing, so I will try to rectify that here.

11.1 Decorators vs. the Decorator Pattern

First, you need to understand that the word "decorator" was used with some trepidation in Python, because there was concern that it would be completely confused with the *Decorator* pattern from the Design Patterns book. At one point other terms were considered for the feature, but "decorator" seems to be the one that sticks.

Indeed, you can use Python decorators to implement the *Decorator* pattern, but that's an extremely limited use of it. Python decorators, I think, are best equated to macros.

11.2 History of Macros

The macro has a long history, but most people will probably have had experience with C preprocessor macros. The problems with C macros were (1) they were in a different language (not C) and (2) the behavior was sometimes bizarre, and often inconsistent with the behavior of the rest of C.

Both Java and C# have added *annotations*, which allow you to do some things to elements of the language. Both of these have the problems that (1) to do what you want, you sometimes have to jump through some enormous and untenable hoops, which follows from (2) these annotation features have their hands tied by the bondage-and-discipline (or as Martin Fowler gently puts it: "Directing") nature of those languages.

In a slightly different vein, many C++ programmers (myself included) have noted the generative abilities of C++ templates and have used that feature in a macro- like fashion.

Many other languages have incorporated macros, but without knowing much about it I will go out on a limb and say that Python decorators are similar to Lisp macros in power and possibility.

11.3 The Goal of Macros

I think it's safe to say that the goal of macros in a language is to provide a way to modify elements of the language. That's what decorators do in Python – they modify functions, and in the case of *class decorators*, entire classes. This is why they usually provide a simpler alternative to metaclasses.

The major failings of most language's self-modification approaches are that they are too restrictive and that they require a different language (I'm going to say that Java annotations with all the hoops you must jump through to produce an interesting annotation comprises a "different language").

Python falls into Fowler's category of "enabling" languages, so if you want to do modifications, why create a different or restricted language? Why not just use Python itself? And that's what Python decorators do.

11.4 What Can You Do With Decorators?

Decorators allow you to inject or modify code in functions or classes. Sounds a bit like *Aspect-Oriented Programming* (AOP) in Java, doesn't it? Except that it's both much simpler and (as a result) much more powerful. For example, suppose you'd like to do something at the entry and exit points of a function (such as perform some kind of security, tracing, locking, etc. – all the standard arguments for AOP). With decorators, it looks like this:

```
@entryExit
def func1():
    print("inside func1()")

@entryExit
def func2():
    print("inside func2()")
```

The `@` indicates the application of the decorator.

11.5 Function Decorators

A function decorator is applied to a function definition by placing it on the line before that function definition begins. For example:

```
@myDecorator
def aFunction():
    print("inside aFunction")
```

When the compiler passes over this code, `aFunction()` is compiled and the resulting function object is passed to the `myDecorator` code, which does something to produce a function-like object that is then substituted for the original `aFunction()`.

What does the `myDecorator` code look like? Well, most introductory examples show this as a function, but I've found that it's easier to start understanding decorators by using classes as decoration mechanisms instead of functions. In addition, it's more powerful.

The only constraint upon the object returned by the decorator is that it can be used as a function – which basically means it must be callable. Thus, any classes we use as decorators must implement `__call__`.

What should the decorator do? Well, it can do anything but usually you expect the original function code to be used at some point. This is not required, however:

```python
# PythonDecorators/my_decorator.py
class my_decorator(object):

    def __init__(self, f):
        print("inside my_decorator.__init__()")
        f() # Prove that function definition has completed

    def __call__(self):
        print("inside my_decorator.__call__()")

@my_decorator
def aFunction():
    print("inside aFunction()")

print("Finished decorating aFunction()")

aFunction()
```

When you run this code, you see:

```
inside my_decorator.__init__()
inside aFunction()
Finished decorating aFunction()
inside my_decorator.__call__()
```

Notice that the constructor for `my_decorator` is executed at the point of decoration of the function. Since we can call `f()` inside `__init__()`, it shows that the creation of `f()` is complete before the decorator is called. Note also that the decorator constructor receives the function object being decorated. Typically, you'll capture the function object in the constructor and later use it in the `__call__()` method (the fact that decoration and calling are two clear phases when using classes is why I argue that it's easier and more powerful this way).

When `aFunction()` is called after it has been decorated, we get completely different behavior; the `my_decorator.__call__()` method is called instead of the original code. That's because the act of decoration *replaces* the original function object with the result of the decoration – in our case, the `my_decorator` object replaces `aFunction`. Indeed, before decorators were added you had to do something much less elegant to achieve the same thing:

```python
def foo(): pass
foo = staticmethod(foo)
```

With the addition of the `@` decoration operator, you now get the same result by saying:

```python
@staticmethod
def foo(): pass
```

This is the reason why people argued against decorators, because the `@` is just a little syntax sugar meaning "pass a function object through another function and assign the result to the original function."

The reason I think decorators will have such a big impact is because this little bit of syntax sugar changes the way you think about programming. Indeed, it brings the idea of "applying code to other code" (i.e.: macros) into mainstream thinking by formalizing it as a language construct.

11.6 Slightly More Useful

Now let's go back and implement the first example. Here, we'll do the more typical thing and actually use the code in the decorated functions:

```
# PythonDecorators/entry_exit_class.py
class entry_exit(object):

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

    def __call__(self):
        print("Entering", self.f.__name__)
        self.f()
        print("Exited", self.f.__name__)

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
```

The output is:

```
Entering func1
inside func1()
Exited func1
Entering func2
inside func2()
Exited func2
```

You can see that the decorated functions now have the "Entering" and "Exited" trace statements around the call.

The constructor stores the argument, which is the function object. In the call, we use the `__name__` attribute of the function to display that function's name, then call the function itself.

11.7 Using Functions as Decorators

The only constraint on the result of a decorator is that it be callable, so it can properly replace the decorated function. In the above examples, I've replaced the original function with an object of a class that has a `__call__()` method. But a function object is also callable, so we can rewrite the previous example using a function instead of a class, like this:

```
# PythonDecorators/entry_exit_function.py
def entry_exit(f):
    def new_f():
        print("Entering", f.__name__)
        f()
        print("Exited", f.__name__)
    return new_f

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
print(func1.__name__)
```

`new_f()` is defined within the body of `entry_exit()`, so it is created and returned when `entry_exit()` is called. Note that `new_f()` is a *closure*, because it captures the actual value of `f`.

Once `new_f()` has been defined, it is returned from `entry_exit()` so that the decorator mechanism can assign the result as the decorated function.

The output of the line `print(func1.__name__)` is `new_f`, because the `new_f` function has been substituted for the original function during decoration. If this is a problem you can change the name of the decorator function before you return it:

```
def entry_exit(f):
    def new_f():
        print("Entering", f.__name__)
        f()
        print("Exited", f.__name__)
    new_f.__name__ = f.__name__
    return new_f
```

The information you can dynamically get about functions, and the modifications you can make to those functions, are quite powerful in Python.

11.8 Review: Decorators without Arguments

If we create a decorator without arguments, the function to be decorated is passed to the constructor, and the `__call__()` method is called whenever the decorated function is invoked:

```
# PythonDecorators/decorator_without_arguments.py
class decorator_without_arguments(object):

    def __init__(self, f):
        """
        If there are no decorator arguments, the function
        to be decorated is passed to the constructor.
        """
        print("Inside __init__()")
        self.f = f

    def __call__(self, *args):
        """
        The __call__ method is not called until the
        decorated function is called.
        """
        print("Inside __call__()")
        self.f(*args)
        print("After self.f(*args)")

@decorator_without_arguments
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("After first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("After second sayHello() call")
```

Any arguments for the decorated function are just passed to `__call__()`. The output is:

```
Inside __init__()
After decoration
Preparing to call sayHello()
Inside __call__()
sayHello arguments: say hello argument list
After self.f(*args)
After first sayHello() call
Inside __call__()
sayHello arguments: a different set of arguments
After self.f(*args)
After second sayHello() call
```

Notice that `__init__()` is the only method called to perform decoration, and `__call__()` is called every time you call the decorated `sayHello()`.

11.9 Decorators with Arguments

The decorator mechanism behaves quite differently when you pass arguments to the decorator.

Let's modify the above example to see what happens when we add arguments to the decorator:

```
# PythonDecorators/decorator_with_arguments.py
class decorator_with_arguments(object):

    def __init__(self, arg1, arg2, arg3):
        """
        If there are decorator arguments, the function
        to be decorated is not passed to the constructor!
        """
        print("Inside __init__()")
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3

    def __call__(self, f):
        """
        If there are decorator arguments, __call__() is only called
        once, as part of the decoration process! You can only give
        it a single argument, which is the function object.
        """
        print("Inside __call__()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:", self.arg1, self.arg2, self.arg3)
            f(*args)
            print("After f(*args)")
        return wrapped_f

@decorator_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("after first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("after second sayHello() call")
```

From the output, we can see that the behavior changes quite significantly:

```
Inside __init__()
Inside __call__()
After decoration
Preparing to call sayHello()
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
after second sayHello() call
```

Now the process of decoration calls the constructor and then immediately invokes `__call__()`, which can only take a single argument (the function object) and must return the decorated function object that replaces the original. Notice that `__call__()` is now only invoked once, during decoration, and after that the decorated function that you return from `__call__()` is used for the actual calls.

Although this behavior makes sense – the constructor is now used to capture the decorator arguments, but the object `__call__()` can no longer be used as the decorated function call, so you must instead use `__call__()` to perform the decoration – it is nonetheless surprising the first time you see it because it's acting so much differently than the no-argument case, and you must code the decorator very differently from the no-argument case.

11.10 Decorator Functions with Decorator Arguments

Finally, let's look at the more complex decorator function implementation, where you have to do everything all at once:

```python
# PythonDecorators/decorator_function_with_arguments.py
def decorator_function_with_arguments(arg1, arg2, arg3):
    def wrap(f):
        print("Inside wrap()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:", arg1, arg2, arg3)
            f(*args)
            print("After f(*args)")
        return wrapped_f
    return wrap

@decorator_function_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("after first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("after second sayHello() call")
```

Here's the output:

```
Inside wrap()
After decoration
Preparing to call sayHello()
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
after second sayHello() call
```

The return value of the decorator function must be a function used to wrap the function to be decorated. That is, Python will take the returned function and call it at decoration time, passing the function to be decorated. That's why we have three levels of functions; the inner one is the actual replacement function.

Because of closures, `wrapped_f()` has access to the decorator arguments `arg1`, `arg2` and `arg3`, *without* having to explicitly store them as in the class version. However, this is a case where I find "explicit is better than implicit," so even though the function version is more succinct I find the class version easier to understand and thus to modify and maintain.

11.11 Further Reading

http://wiki.python.org/moin/PythonDecoratorLibrary More examples of decorators. Note the number of these examples that use classes rather than functions as decorators.

http://scratch.tplus1.com/decoratortalk Matt Wilson's *Decorators Are Fun*.

http://loveandtheft.org/2008/09/22/python-decorators-explained Another introduction to decorators.

http://www.phyast.pitt.edu/~micheles/python/documentation.html Michele Simionato's decorator module wraps functions for you. The page includes an introduction and some examples.

CHAPTER

TWELVE

GENERATORS, ITERATORS, AND ITERTOOLS

CHAPTER

THIRTEEN

COMPREHENSIONS

History: where did they come from?

They require a mind shift.

What makes them so compelling (once you 'get it')?

A two-level list comprehension using `os.walk()`:

```
# Comprehensions/os_walk_comprehension.py
import os
restFiles = [os.path.join(d[0], f) for d in os.walk(".")
             for f in d[2] if f.endswith(".rst")]
for r in restFiles:
    print(r)
```

13.1 A More Complex Example

Note: This will get a full description of all parts.

```
# CodeManager.py
"""
TODO: Break check into two pieces?
TODO: update() is still only in test mode; doesn't actually work yet.

Extracts, displays, checks and updates code examples in restructured text (.rst)
files.

You can just put in the codeMarker and the (indented) first line (containing the
file path) into your restructured text file, then run the update program to
automatically insert the rest of the file.
"""
import os, re, sys, shutil, inspect, difflib

restFiles = [os.path.join(d[0], f) for d in os.walk(".") if not "_test" in d[0]
             for f in d[2] if f.endswith(".rst")]

class Languages:
    "Strategy design pattern"

    class Python:
        codeMarker = "::\n\n"
        commentTag = "#"
        listings = re.compile("::\n\n( {4}#.*(?:\n+ {4}.*)*)")

    class Java:
        codeMarker = "..  code-block:: java\n\n"
        commentTag = "//"
        listings = \
            re.compile(".. *code-block:: *java\n\n( {4}//.*(?:\n+ {4}.*)*)")

def shift(listing):
    "Shift the listing left by 4 spaces"
    return [x[4:] if x.startswith("    ") else x for x in listing.splitlines()]

# TEST - makes duplicates of the rst files in a test directory to test update():
dirs = set([os.path.join("_test", os.path.dirname(f)) for f in restFiles])
if [os.makedirs(d) for d in dirs if not os.path.exists(d)]:
    [shutil.copy(f, os.path.join("_test", f)) for f in restFiles]
testFiles = [os.path.join(d[0], f) for d in os.walk("_test")
             for f in d[2] if f.endswith(".rst")]

class Commands:
    """
    Each static method can be called from the command line. Add a new static
    method here to add a new command to the program.
    """

    @staticmethod
    def display(language):
        """
        Print all the code listings in the .rst files.
        """
        for f in restFiles:
            listings = language.listings.findall(open(f).read())
            if not listings: continue
            print('=' * 60 + "\n" + f + "\n" + '=' * 60)
            for n, l in enumerate(listings):
                print("\n".join(shift(l)))
                if n < len(listings) - 1:
                    print('-' * 60)

    @staticmethod
    def extract(language):
        """
        Pull the code listings from the .rst files and write each listing into
        its own file. Will not overwrite if code files and .rst files disagree
        unless you say "extract -force".
        """
        force = len(sys.argv) == 3 and sys.argv[2] == '-force'
        paths = set()
        for listing in [shift(listing) for f in restFiles
                    for listing in language.listings.findall(open(f).read())]:
            path = listing[0][len(language.commentTag):].strip()
            if path in paths:
                print("ERROR: Duplicate file name: %s" % path)
                sys.exit(1)
            else:
                paths.add(path)
            path = os.path.join("..", "code", path)
            dirname = os.path.dirname(path)
            if dirname and not os.path.exists(dirname):
                os.makedirs(dirname)
            if os.path.exists(path) and not force:
                for i in difflib.ndiff(open(path).read().splitlines(), listing):
                    if i.startswith("+ ") or i.startswith("- "):
                        print("ERROR: Existing file different from .rst")
                        print("Use 'extract -force' to force overwrite")
                        Commands.check(language)
                        return
            file(path, 'w').write("\n".join(listing))

    @staticmethod
    def check(language):
        """
        Ensure that external code files exist and check which external files
        have changed from what's in the .rst files. Generate files in the
        _deltas subdirectory showing what has changed.
        """
        class Result: # Messenger
            def __init__(self, **kwargs):
                self.__dict__ = kwargs
        result = Result(missing = [], deltas = [])
        listings = [Result(code = shift(code), file = f)
                    for f in restFiles for code in
                    language.listings.findall(open(f).read())]
        paths = [os.path.normpath(os.path.join("..", "code", path)) for path in
                    [listing.code[0].strip()[len(language.commentTag):].strip()
                     for listing in listings]]
        if os.path.exists("_deltas"):
            shutil.rmtree("_deltas")
        for path, listing in zip(paths, listings):
            if not os.path.exists(path):
                result.missing.append(path)
            else:
                code = open(path).read().splitlines()
                for i in difflib.ndiff(listing.code, code):
                    if i.startswith("+ ") or i.startswith("- "):
                        d = difflib.HtmlDiff()
                        if not os.path.exists("_deltas"):
                            os.makedirs("_deltas")
                        html = os.path.join("_deltas",
                            os.path.basename(path).split('.')[0] + ".html")
                        open(html, 'w').write(
                            "<html><h1>Left: %s<br>Right: %s</h1>" %
                            (listing.file, path) +
                            d.make_file(listing.code, code))
                        result.deltas.append(Result(file = listing.file,
                            path = path, html = html, code = code))
                        break
        if result.missing:
            print("Missing %s files:\n%s" %
                  (language.__name__, "\n".join(result.missing)))
        for delta in result.deltas:
            print("%s changed in %s; see %s" %
                  (delta.file, delta.path, delta.html))
        return result

    @staticmethod
    def update(language): # Test until it is trustworthy
        """
        Refresh external code files into .rst files.
        """
        check_result = Commands.check(language)
        if check_result.missing:
            print(language.__name__, "update aborted")
            return
        changed = False
        def _update(matchobj):
            listing = shift(matchobj.group(1))
            path = listing[0].strip()[len(language.commentTag):].strip()
            filename = os.path.basename(path).split('.')[0]
            path = os.path.join("..", "code", path)
            code = open(path).read().splitlines()
            return language.codeMarker + \
                "\n".join([("    " + line).rstrip() for line in listing])
        for f in testFiles:
            updated = language.listings.sub(_update, open(f).read())
            open(f, 'w').write(updated)

if __name__ == "__main__":
    commands = dict(inspect.getmembers(Commands, inspect.isfunction))
    if len(sys.argv) < 2 or sys.argv[1] not in commands:
        print("Command line options:\n")
        for name in commands:
            print(name + ": " + commands[name].__doc__)
    else:
        for language in inspect.getmembers(Languages, inspect.isclass):
            commands[sys.argv[1]](language[1])
```

CHAPTER

FOURTEEN

COROUTINES & CONCURRENCY

Primary focus should be on:

1. Using `yield` to create coroutines
2. Using the new `multiprocessing` module

and then showing some alternative techniques.

foo bar `input()` baz.

14.1 Further Reading

> This article argues that large-scale parallelism – which is what `multiprocessing` supports – is the more important problem to solve, and that functional languages don't help that much with this problem.

CHAPTER

FIFTEEN

PART II: IDIOMS

CHAPTER

SIXTEEN

DISCOVERING THE DETAILS ABOUT YOUR PLATFORM

The Python library XXX will give you some information about your machine, but it falls short. Here's a rather messy, but useful way to figure out everything else.

Just a starting point:

```
# MachineDiscovery/detect_CPUs.py
def detect_CPUs():
    """
    Detects the number of CPUs on a system. Cribbed from pp.
    """
    # Linux, Unix and MacOS:
    if hasattr(os, "sysconf"):
        if os.sysconf_names.has_key("SC_NPROCESSORS_ONLN"):
            # Linux & Unix:
            ncpus = os.sysconf("SC_NPROCESSORS_ONLN")
            if isinstance(ncpus, int) and ncpus > 0:
                return ncpus
        else: # OSX:
            return int(os.popen2("sysctl -n hw.ncpu")[1].read())
    # Windows:
    if os.environ.has_key("NUMBER_OF_PROCESSORS"):
            ncpus = int(os.environ["NUMBER_OF_PROCESSORS"]);
            if ncpus > 0:
                return ncpus
    return 1 # Default
```

CHAPTER

SEVENTEEN

A CANONICAL FORM FOR COMMAND-LINE PROGRAMS

Creating Python programs for command-line use involves a certain amount of repetitious coding, which can often be left off or forgotten. Here is a form which includes everthing.

Note that if you are using Windows, you can add Python programs to your "File New" menu and automatically include the above text in the new file. This article shows you how. Other operating systems have their own automation features.

CHAPTER

EIGHTEEN

MESSENGER/DATA TRANSFER OBJECT

The *Messenger* or *Data Transfer Object* is a way to pass a clump of information around. The most typical place for this is in return values from functions, where tuples or dictionaries are often used. However, those rely on indexing; in the case of tuples this requires the consumer to keep track of numerical order, and in the case of a **dict** you must use the `d["name"]` syntax which can be slightly less desireable.

A Messenger is simply an object with attributes corresponding to the names of the data you want to pass around or return:

```
# Messenger/MessengerIdiom.py

class Messenger:
    def __init__(self, **kwargs):
        self.__dict__ = kwargs

m = Messenger(info="some information", b=['a', 'list'])
m.more = 11
print m.info, m.b, m.more
```

The trick here is that the `__dict__` for the object is just assigned to the **dict** that is automatically created by the `**kwargs` argument.

Although one could easily create a `Messenger` class and put it into a library and import it, there are so few lines to describe it that it usually makes more sense to just define it in-place whenever you need it – it is probably easier for the reader to follow, as well.

CHAPTER

NINETEEN

PART III: PATTERNS

CHAPTER

TWENTY

INDICES AND TABLES

- *Index*
- *Search Page*

INDEX

canonical form
  script command-line, 47
command-line
  canonical form, script, 47
comprehension
  generator, 37
  list, 37
concurrency, 42
coroutines, 42

data transfer object (messenger), 49
decorator: Python decorators, 25

generator
  comprehension, 37
generators, 35
GIL: Global Interpreter Lock, 42

iterators, 35
itertools, 35

Language differences
  Python 3, 23
list
  comprehension, 37

messenger (data transfer object), 49
multiprocessing, 42

parallelism, 42
Python 3
  Language differences, 23

script
  command-line canonical form, 47

threads, 42