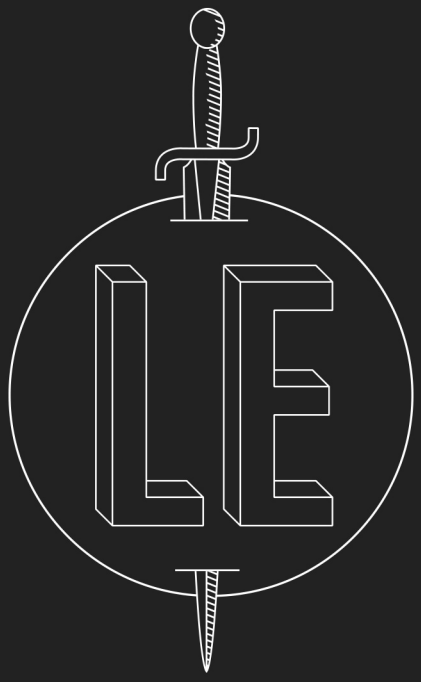


LEARN ENOUGH

TUTORIAL WRITING

TO BE DANGEROUS

TUTORIAL BY

MICHAEL HARTL

Learn Enough Tutorial Writing to Be Dangerous

Michael Hartl

1 Introduction

The purpose of *Learn Enough Tutorial Writing to Be Dangerous* is to codify the process I've used to write instructional books and articles such as *The Ruby on Rails Tutorial*, *The Tau Manifesto*, and the Learn Enough tutorials. While there are undoubtedly aspects of this process that "can't be bottled", my hope is that this meta-tutorial will help others produce instructive tutorials of their own. The presentation is especially geared toward authors using the Softcover system to write their tutorials (and even more especially toward authors collaborating with me directly), but the general principles should be applicable to a wide variety of contexts.

When discussing the details of tutorial-writing techniques, it's useful to keep in mind the overarching goal—to keep in mind, as it were, the forest, even as we consider the trees. Many novice (and even experienced) tutorial creators prioritize clarity over other goals, believing that what separates good tutorials from bad is how clear the language is. One manifestation of this approach is the common practice in instructional video presentations of speaking artificially slowly and clearly, which in most cases simply drains the presentation of all spontaneity and passion. Even the most clearly expressed tutorial can still bore the audience to tears.

To be sure, clarity is a virtue, but the most essential element is *story*. Human beings have been learning and retaining information through stories for thousands, probably millions of years (Figure 1).[1] In our particular case, the story takes the form of a *long-form technical narrative*. (Here *long-form* refers to any length in excess of a typical short blog post. For example, *The Tau Manifesto*, although shorter than a book, is still a 50-page PDF.) The virtue of narrative is that, in addition to clarity, it has *motivation* and *structure*. When done well, these make the tutorial more engaging, which makes it easier for the reader[2] to pay attention. The clearest tutorial in the world is useless if the reader stops paying attention and loses interest.

[1] Image retrieved from https://en.wikipedia.org/wiki/Odyssey#/media/File:Odysseus_Sirens_BM_E440_n2.jpg on 2015-08-03

[2] Here I use the term *reader* because our principal subject is written tutorials, but similar considerations apply to instruction in the form of screencasts, classroom teaching, etc.

Figure 1: Humans have been learning from stories for a long time.

In the context of technical writing, having a "story" doesn't mean embedding the tutorial in a fictional narrative; with a technical subject, this approach is nearly impossible to pull off.[3] What it means is that the exposition should be broadly structured as a *narrative arc* containing a primary goal which, when reached, serves as the climax of the story. The specific form varies—"writing a mini Twitter clone" in the *Ruby on Rails Tutorial*, "understanding why π is wrong" in *The Tau Manifesto*, etc.—but the basic principle of the narrative arc is the same. Indeed, this narrative structure applies on smaller scales as well, with both small and large story arcs corresponding to short- and long-term goals that keep the material constantly engaging and relevant.

Throughout the rest of this meta-tutorial, we'll first discuss some general principles to consider when producing technical tutorials (Section 2). Then we'll discuss the process of actually making the darn things (Section 3). Finally, in Section 4, we'll talk about some miscellaneous tricks of the trade (such as Figure 2)[4] to make the tutorial more polished, easier to understand, and more fun.

2 General principles

To construct a long-form technical narrative, we apply two overarching and closely related principles, one local and one global. The local principle is *motivation*: at every stage in the narrative, readers should know *why* they are studying the subject at hand. (Appropriately enough, this leads to the second meaning of "motivation": motivating the *subject* helps motivate the *reader* as well.) As we'll see in Section 2.1, a failure to find a good motivation is often a good clue that the subject can be deferred till later (and sometimes indefinitely).

The global principle is *structure*: what gets covered where. Structure may

[3] Perhaps the most famous example of this approach, in a philosophical if not quite technical context, is Ayn Rand's novel *Atlas Shrugged*, which promulgates Rand's philosophy of *Objectivism* in the form of an adventure-mystery story. The enduring popularity of *Atlas Shrugged* suggests that her approach basically works, but even in this case Rand resorts to an awkward kluge, shoehorning a long philosophical treatise into the flow of the narrative. (Tellingly, Rand's other major philosophical novel, *The Fountainhead*, resorts to similar trickery, this time in the form of the protagonist's discursive courtroom exposition.)

[4] Image retrieved from https://www.flickr.com/photos/tambako/3557843402 on 2015-08-03

Figure 2: This sleeping baby anteater is adorable and helps keep the reader's attention.

be the most difficult aspect of a narrative to appreciate, because when done well it's invisible. Typically, much of the structure emerges organically through a process of successive refinement (Section 3), but we still need a overall structural form to guide our composition—the placement of muscles may motivate the placement of tendons, but we still need a skeleton. In particular, as discussed briefly in the introduction, the tutorial should have a narrative arc, with an overriding *goal* to be reached. (Indeed, in the case of a longer document like the *Ruby on Rails Tutorial*, each chapter may have one or more narrative arcs of its own.)

Guided by the need to motivate each subject, we assemble the building blocks of the narrative, which are composed mainly of concrete examples (Section 2.2) carefully chosen to repeat key ideas throughout the discussion (Section 2.3). We then place the examples according to the structure, making adjustments as necessary (Section 3).

2.1 Defer

When writing an instructional tutorial, there's a temptation to follow a "logical" approach, collecting all the material on each particular subject in one place. While this is a perfectly sensible strategy when making a *reference* work, it's ill-suited to an instructional tutorial.[5]

When optimizing for the learning experience of the reader, a better approach is to defer material as much as possible: if a subject doesn't *have* to be covered

[5]Mathematicians are especially guilty of following this "logical" approach (which among other things bears little resemblance to how mathematics is actually done). For example, virtually every calculus textbook in the known universe includes a thorough treatment of *limits*, including fancy limits like

$$\lim_{x \to 0} \frac{\sin x}{x} = 1,$$

before ever touching on differentiation or integration. But in fact you can start differentiating functions like $f(x) = x^2$ right away using trivial limits like this:

$$f'(x) = \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h} = \lim_{h \to 0} \frac{2xh + h^2}{h} = \lim_{h \to 0} (2x + h) = 2x.$$

Likewise, you can state and prove the fundamental theorem of calculus in special cases like $f(x) = 1$ and $f(x) = x$ before undertaking the much more difficult task of proving the general case.

at a particular point, it's better to wait until *needs* to be covered. For example, the *Ruby on Rails Tutorial* develops a signup page that includes the following code:

```
@user = User.new(params[:user])    # Not the final implementation!
```

As indicated in the text following the Ruby comment character #, this is not the final implementation. The reason is that the actual implementation uses a technique called *strong parameters* to enforce a security model on submitted information, but this technique is not needed to write the initial signup page. By deferring this material, we lighten the burden on the reader (and author).

The principle of *motivation* can help determine whether material should be deferred or not. In the case of the Rails Tutorial signup page, the immediate motivation is making a page that can handle *invalid* submissions (with displayed error messages), a task which doesn't require the more advanced strong parameters technique to work. As a result, the more advanced material can be deferred till later, where it in turn is motivated by the need to handle *valid* submissions and save the user to the database. The even more advanced subject of handling invalid submissions in-browser using JavaScript is never necessary in the course of learning Rails web development, so it is deferred indefinitely, i.e., it's omitted entirely.

One concrete method for deferring material is forward–backward reference pairs. This involves covering some basic material initially while including a forward reference to more advanced material. For instance, the initial discussion of the *Rails Tutorial* signup page mentioned above appears in Figure 3, where the code with the "Not the final implementation!" comment is followed by a forward reference to "Section 7.3.2".

Upon reaching the more advanced section, we include a backward reference, reminding the reader of the previous material, as seen in Figure 4. Note that the numbers match up: the more advanced section is indeed Section 7.3.2, as promised by the forward reference, and the original code listing is Listing 7.16, as promised by the backward reference. (Figure 4 also includes a second backward reference, linking the material in Section 7.3.2 with an idea

Listing 7.16: A `create` action that can handle signup failure.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

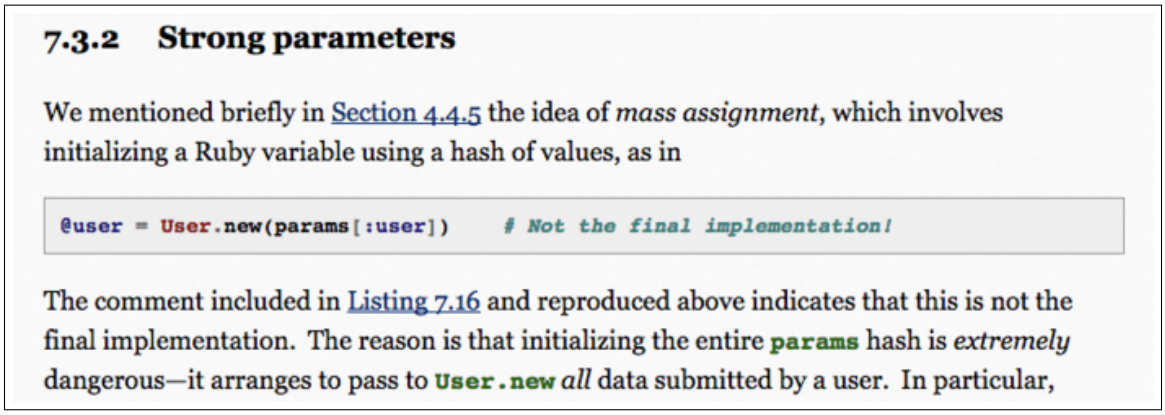
  def show
    @user = User.find(params[:id])
  end

  def new
    @user = User.new
  end

  def create
    @user = User.new(params[:user])    # Not the final implementation!
    if @user.save
      # Handle a successful save.
    else
      render 'new'
    end
  end
end
```

Note the comment: this is not the final implementation. But it's enough to get us started, and we'll finish the implementation in Section 7.3.2.

Figure 3: A forward reference to more advanced material.



7.3.2 Strong parameters

We mentioned briefly in Section 4.4.5 the idea of *mass assignment*, which involves initializing a Ruby variable using a hash of values, as in

```
@user = User.new(params[:user])    # Not the final implementation!
```

The comment included in Listing 7.16 and reproduced above indicates that this is not the final implementation. The reason is that initializing the entire `params` hash is *extremely* dangerous—it arranges to pass to `User.new` *all* data submitted by a user. In particular,

Figure 4: A backward reference to the relevant code listing.

introduced all the way back in Section 4.4.5.) We'll learn how to make such forward and backward references, or *cross-references*, in Section 3.3. (See what I did there (Figure 5)?)[6]

2.1.1 Reassuring the reader

When covering a simplified version of a subject and deferring more advanced material, it's often helpful to explicitly reassure the reader that they don't need to understand everything right away. Forward references are especially useful for this; when I read a tutorial that promises to cover a subject in greater depth "later", it always makes me nervous—*What if the author forgets? What if I don't recognize it when it occurs?*—and adding a forward reference relieves such anxiety. When we write "We'll learn how to make such forward and backward references, or *cross-references*, in Section 3.3," there's no doubt that the material will in fact be covered. Even better, since ebook production systems like Softcover automatically link cross-references, it's possible to skip ahead with a single click and see that the promise to cover more advanced material is in fact fulfilled. Likewise, when the future section is reached, the back reference lets the reader pop back to the previous material for a quick reminder of what got covered before.

[6] Image is a common Internet meme.

Figure 5

Figure 6: How ’bout them apples?

2.2 Concrete examples

Any technical tutorial should be grounded in concrete examples. Abstraction has its place, but it usually belongs *after* a series of examples, not before. The main reason is that human brains are better at generalizing from the concrete to the abstract than *vice versa*. Rather than describe the idea of a sweet, firm fruit with woody stems and shiny skin, just show a bunch of different apples (Figure 6).[7]

For example, suppose our task is to explain the Unix command `ls` (list). (This is one of the subjects covered in *Learn Enough Command Line to Be Dangerous*.) Imagine if we took the abstract approach exemplified by the Unix manual (or “man”) page for `ls` (Listing 1).

[7] Image retrieved from https://www.flickr.com/photos/msr/448820990 on 2015-08-03

Listing 1: The man page for `ls`.

```
$ man ls
LS(1)                    BSD General Commands Manual                    LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file ...]
.
.
.
```

Yikes! That's a formidable `SYNOPSIS` for a command that most of the time looks like this:

```
$ ls
```

Other common variations are the "long form" view,

```
$ ls -l
```

and the individual file/directory view, as in

```
$ ls foo.txt
```

In fact, we can see in Listing 1 that the letter `l` (for "long") does appear in the bracketed list of options, and (as indicated by `[file ...]`) the `ls` command can take a filename as an argument. But absent the context provided by concrete examples, the man page approach would be completely overwhelming. (Indeed, in my own experience I find that even the most fastidiously written man pages are less useful than two or three well-chosen examples.)

To be maximally instructive, at any given point in the narrative the examples should strike a balance between being too easy and too hard (Figure 7).[8]

[8] Image is in the public domain.

Figure 7: Goldilocks chooses examples that are just right.

Choosing such examples takes practice and judgment, but one useful guideline is, per Section 2.1, to defer that which *can* be deferred. Thus, the signup page from the *Rails Tutorial* shown in Figure 3 is challenging but not *too* challenging because it defers the more advanced material to a future section. In addition, as discussed further in Section 3.3, it's helpful to put ourselves in the mind of the learner: given the knowledge of the intended reader, how much extra information can be handled at this point in the tutorial? When done well, such Goldilocks-style examples can induce a state of *flow* that makes learning immersive and fun (Box 1).[9]

Box 1. Going with the flow

[9] Figure 8 image retrieved from https://www.flickr.com/photos/mariachily/5234325547 on 2015-08-04

Flow is a mental state characterized by complete focus, total immersion, and seemingly effortless concentration. The flow state is sometimes called being "in the zone". In his book *Flow*, psychologist Mihaly Csikszentmihalyi ("MEE-hi chick-SENT-mee-hi") argues that flow is induced in part by engaging in tasks that are right at the limit of competence, so that, with concentration, they can just barely be completed successfully. For example, a rock climber (Figure 8) can promote a flow state by choosing a route that is right at the limit of her abilities—too easy and she'll get bored, too hard and she'll get frustrated.

By deferring unnecessary material, putting ourselves in the mind of the learner, and otherwise exercising judgment in selecting examples, tutorial authors can help readers enter a state of flow. The resulting feeling can be difficult to identify exactly, but if your readers say things like "I'm not sure what it is about this tutorial, it just seems to carry me along," you'll know you're on the right track.

2.3 Repetition

Choosing a series of concrete examples as in Section 2.2 naturally produces a certain degree of repetition. But this repetition is a feature, not a bug, as repetition is one of the most underrated pedagogical tools. The key is to include enough variety that the reader doesn't become bored—variety which happens naturally when producing related examples of increasing sophistication. Such a pattern, known as the *spiral method*, reinforces previous material while challenging readers with just enough novelty to keep them interested and engaged (Box 1).

One example of this technique in *Learn Enough Command Line to Be Dangerous* is the treatment of *pipes*, which involve sending the output from one command-line program to the input of another. Over the course of several sections, that tutorial introduces the simple pipe

Figure 8: A well-chosen route can help rock climbers reach a state of flow.

```
$ head sonnets.txt | wc
```

and then immediately includes an exercise whose answer appears as

```
$ head -n 18 sonnets.txt | tail -n 14
```

Later, when covering the Unix **grep** utility, the tutorial pipes the output of the **ps** (process status) command to select programs matching a particular string ("spring"):

```
$ ps aux | grep spring
```

An exercise on this technique then follows, and a later section includes two more exercises using pipes.

The result of the above progression (whose details you certainly don't need to follow here) is a deeper understanding of the subject than could be achieved by putting all the material on pipes in one place. By spacing out the examples and slowly increasing their sophistication, the tutorial gives the reader time to let the previous lessons sink in, thereby implementing a simplified version of a powerful learning technique called *spaced repetition*.

Repetition is the mother of memory,[10] so good tutorials repeat the most important material in multiple places throughout the narrative.

3 Successive refinement

The high-minded principles from Section 2 are nice, but eventually we have to face the horror of the empty page—or, rather, its digital equivalent, the empty text editor (Figure 9). When facing this mighty foe, it's natural to feel some fear. (The prospect of writing a whole book is daunting, of course, but even an article can be intimidating.)

[10] *Repetītiō est māter memoriae.*

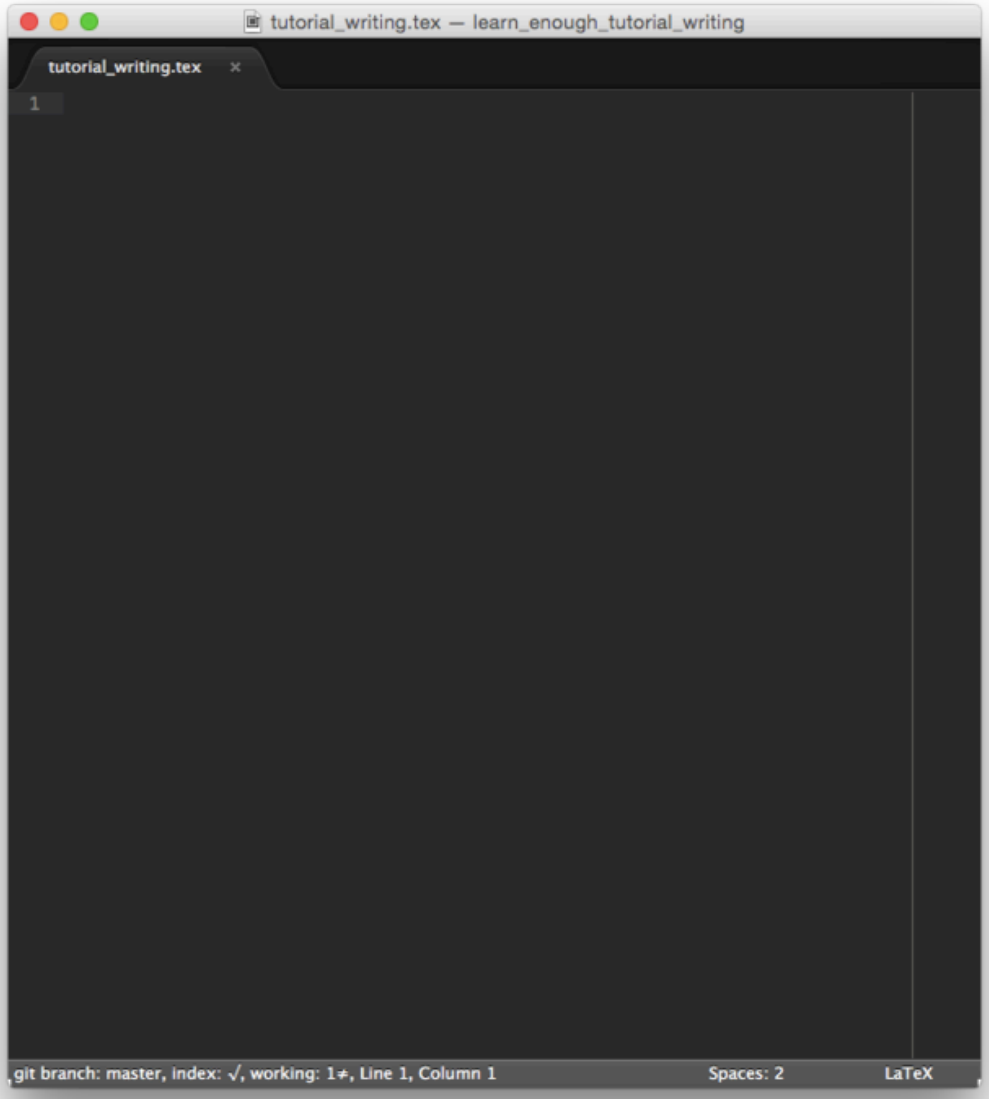


Figure 9: Facing the horror of an empty text editor.

To control this fear, my recommendation is to use a strategy of *successive refinement*: a series of individually tractable steps, each of which builds on the previous step and moves us toward our goal.[11] This makes use of what I like to call the *comb-over principle*, named after Paul Graham's reflections on the subject:[12]

> I've always been fascinated by comb-overs, especially the extreme sort that make a man look as if he's wearing a beret made of his own hair. Surely this is a lowly sort of thing to be interested in—the sort of superficial quizzing best left to teenage girls. And yet there is something underneath. The key question, I realized, is how does the comber-over not see how odd he looks? And the answer is that he got to look that way incrementally. What began as combing his hair a little carefully over a thin patch has gradually, over 20 years, grown into a monstrosity. Gradualness is very powerful. And that power can be used for constructive purposes too: just as you can trick yourself into looking like a freak, you can trick yourself into creating something so grand that you would never have dared to plan such a thing. Indeed, this is just how most good software gets created. You start by writing a stripped-down kernel (how hard can it be?) and gradually it grows into a complete operating system. Hence the next leap: could you do the same thing in painting, or in a novel?

The comb-over principle is thus the *power of gradualness*. When writing a long-form narrative tutorial, I firmly recommend applying the comb-over principle to a strategy of successive refinement. (I firmly recommend against applying comb-overs (Figure 10).)

Over time, you may end up developing your own strategy, but until you do I recommend the following sequence:

1. Free association

[11] Note that I said *tractable*, not *easy*. Some steps can still be quite painful.

[12] Paul Graham, "The Age of the Essay", (2004).

Figure 10: Trust me—you'll look better without a comb-over.

2. Content throwdown
3. Main exposition
4. Polishing pass(es)
5. Shipping

The rest of this section discusses each of these steps in more detail.

3.1 Free association

The first step in the method of successive refinement is *free association*. I recommend opening a new file in a text editor and typing out all the subjects you want to cover, roughly in the order you want to cover them (thus providing the first hints of structure). Even at this early stage, I recommend considering the narrative arc of the tutorial, but don't worry about polish or phrasing; focus on getting things out of your head and onto the page.

If you're using Softcover, the steps getting started on the free association step might look like this:[13]

```
$ softcover new --article learn_enough_command_line
$ cd learn_enough_command_line/
$ mv chapters/an_article.md chapters/command_line.md
<edit Book.txt to use the filename command_line.md>
```

Then open **`command_line.md`** in a text editor, select and delete the generated content, and use the resulting empty file for the free association.

To give you a better idea of what this step might look like in real life, I've gone back in the source of the present tutorial (using `git log`) to find the state of the document when I was done taking notes. The result appears in Figure 11. Note that, even at this early stage, I had inserted sections, but this is optional.

Because tracking the changes in a full tutorial (even one as relatively short as the present meta-tutorial) would be unwieldy, throughout the rest of this section

[13] I actually prefer LaTeX to Markdown, but the latter is more convenient for many authors.

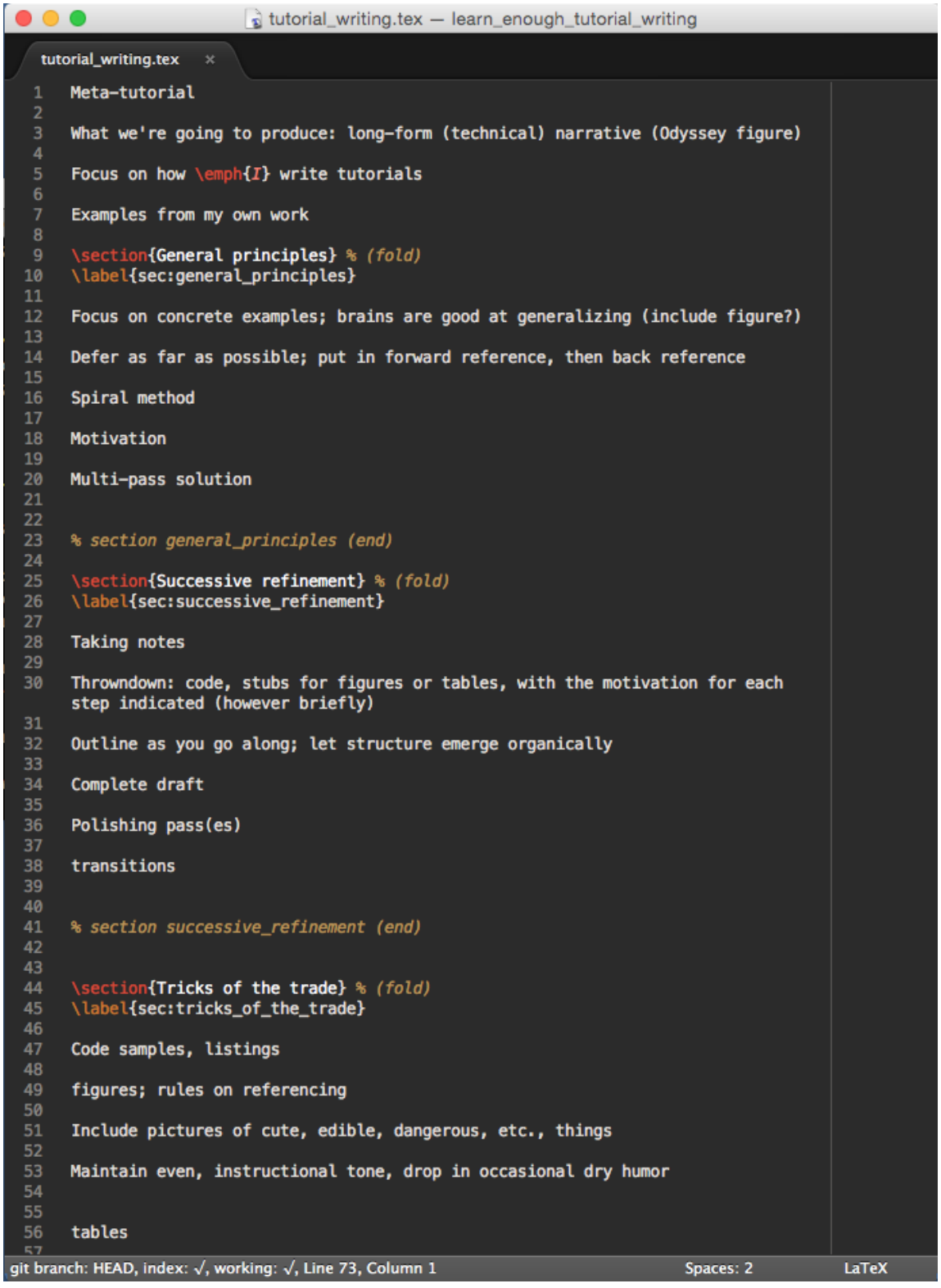


Figure 11: The present document after the free association step.

we'll develop a simplified example based on a hypothetical tutorial on the Ruby programming language. In particular, after the free-association step, part of such a tutorial might look like Listing 2.

Listing 2: A fragment of a hypothetical Ruby tutorial free association document.

```
1 Ruby data structures
2
3 numbers (integers, floats)
4
5 strings
6
7 arrays
8
9 hashes
```

3.2 Content throwdown

In my experience, the content throwndown is the second most painful step in the process of successive refinement. (The most painful is the main exposition (Section 3.3).) The throwdown requires a relatively high *activation energy* (Figure 12)[14]—just getting started requires a lot of concentration and initiative, and you have to load a lot of stuff into your head to be productive.

In the content throwdown, we go through the skeleton that resulted from the free association (Section 3.1) and begin to flesh it out with code samples (Box 2), figures, tables, sections, etc. (Section 4). For example, consider the "hashes" line from the free association (Line 9 in Listing 2). At the content throwdown stage, we could expand it by putting in some stub paragraphs and some example code, as shown in Listing 3.

Listing 3: The content throwdown for hashes.

```
1 ## Data structures
2 .
3 .
```

[14] Image retrieved from https://commons.wikimedia.org/wiki/File:Activation_energy.svg on 2015-08-19

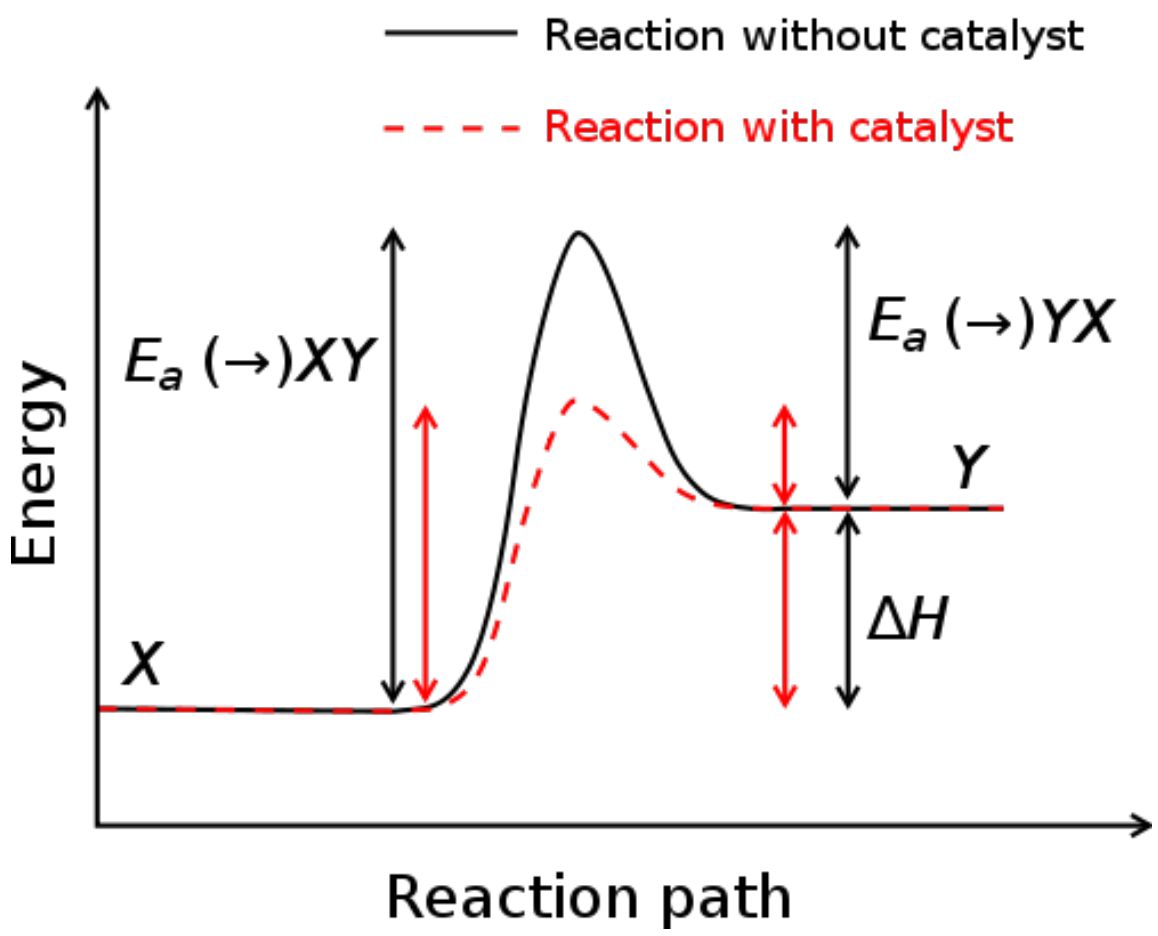


Figure 12: The activation energy for both the content throwdown & main exposition is high.

```
4  .
5  ### Hashes
6  \label{sec:hashes}
7
8  Hashes are like arrays, but where the index doesn't have to be an integer.
9  Very common in Ruby web frameworks like Sinatra and Rails, will use
10 frequently
11
12 start with empty hash {}
13
14 ```irb
15 >> hash = {}
16 >> hash['foo'] = 'bar'
17 >> hash['baz'] = 'quux'
18 >> hash
19 => {"foo"=>"bar", "baz"=>"quux"}
20 ```
21
22 Note that the hash elements are in the same order as defined. This is true in
23 Ruby 2 or later, but it's a bad idea to rely on this fact, as generally
24 speaking hashes (also called *dictionaries* or *maps*) don't always guarantee
25 the order of their elements.
```

Listing 3 includes a code sample (Lines 14–20) and adds a section on data structures (Line 1) containing a subsection on hashes (Line 5, label for cross-references on Line 6). (The vertical ellipsis in Listing 3 represents omitted content.) Note in particular the section levels (indicated in Markdown with hash symbols such as `###`), which are essentially like HTML header tags (`h1`, `h2`, etc.). Don't worry too much about how deep each section is (e.g., whether it's `##` or `###`), as this can be changed during the main exposition (Section 3.3) or in one of the polishing passes (Section 3.4).

Box 2. Sample code

When writing a programming tutorial, it's a good idea to include sample code to provide concrete examples of abstract programming concepts. If your code samples are small and self-contained, they can be developed as part of the content throwndown (Section 3.2), but if you're making a more extensive *sample application* (as used, for example, in the *Rails Tutorial*), I recommend breaking out its

development into a separate step, typically immediately before the content throwdown. Try to design the sample application using principles similar to those described in Section 2.1; e.g., if a particular feature isn't needed for the exposition, err on the side of deferring it till later (or leave it out entirely).

At the content throwdown stage, it's common to move the subject order around as we get a better idea of the most natural progression of ideas. By adding sections as we go along and rearranging as necessary, much of the structure of the document should emerge organically (which means you will probably not have to make an explicit outline of your document). In addition, the scope should start to become clear, especially in the form of the number of sections in each chapter or article. Although there's a little wiggle room, I generally recommend limiting the scope to 4–6 sections (plus a possible conclusion). See the *Ruby on Rails Tutorial* and *Learn Enough Command Line to Be Dangerous* for concrete examples.

At the content throwndown stage, it's also fine not to have a polished exposition, and things like introductions and transitions can be skipped. Essentially, anything that feels like an excuse for not throwing down content can be deferred to the main exposition (Section 3.3). It's important not to forget motivation, though; the *reason* for including every part of the content throwdown should be clearly indicated in the text. For instance, Listing 3 takes care to mention that Ruby hashes are important for understanding Ruby web frameworks, a likely goal for readers of a Ruby tutorial.

3.3 Main exposition

The third pass through the tutorial is the main exposition. As with the throwdown, the main exposition requires a high activation energy (Figure 12). In my experience, this is the most painful step, because it's in the main exposition that most of the elements we deferred in previous steps have to be addressed. In particular, we need to include drafts of full explanations for any subjects covered. To facilitate this, I suggest (as mentioned in Section 2.2) that you try to

put yourself in mind of the learner—at every stage, think "What do we know at this point in the exposition?" In addition, the main exposition should include the first attempts at transitions and cross-references (Box 3),[15] and I also recommend writing at least stub versions of any aside boxes you think you might want in the finished product.

Box 3. Cross-references

As discussed in Section 2.1, cross-references give us a natural way to defer more advanced material to a future section and then link back to the less advanced material once that section is reached. The resulting references and links really tie the tutorial together (Figure 13).

(Not all typesetting systems support such cross-references, which is one reason I and my cofounders developed the Softcover publishing system. Softcover supports cross-references using the powerful LaTeX typesetting language, either embedded in Markdown or in raw LaTeX. The examples in this box assume you're using Softcover or a similar system.)

One technique I particularly like during the content throwdown (Section 3.2) or main exposition (Section 3.3) is to put in a *broken* (undefined) forward-reference as a reminder to fill it in later:

> We'll cover some basic stuff here. In Section **??**, we'll cover more advanced stuff.

This uses a LaTeX *label* to identify the section (`sec:advanced_stuff`), which is also the label that should be used when the more advanced section is reached:

```
\section{Advanced stuff}
\label{sec:advanced_stuff}
```

At this point, the forward reference will be filled in automatically:

[15] Image retrieved from http://www.lifedaily.com/wp-content/uploads/2013/06/the-carpet-e1370859815968.jpg on 2015-08-24

Figure 13: Cross-references are like a rug that really ties the tutorial together.

> We'll cover some basic stuff here. In Section 3, we'll cover more advanced stuff.

It's also a good idea to put a back reference in as well:

> In Section 2, we saw some basic stuff. In this section, we cover some more advanced stuff.

For more information on the syntax needed to create such references, see "Labels and cross-references" in *The Softcover Book*.

Returning to the example of a Ruby tutorial, recall that Listing 3 indicated the basic direction of the exposition and threw down a code sample, but it made no attempt at a complete explanation. Listing 4 shows an example of how we might fill in Listing 3 with a better introduction and references to previous ma-

terial. I suggest reviewing Listing 3 and then reading Listing 4 all the way through to help develop an intuition for the changes involved, but don't worry about catching every little diff.

Listing 4: The main exposition for hashes.

```
## Data structures
.
.
.
### Arrays
\label{sec:arrays}
.
.
.
We'll discuss the closely related *hash* data structure in
Section~\ref{sec:hashes}.
.
.
.
### Hashes
\label{sec:hashes}

We saw in Section~\ref{sec:arrays} that Ruby arrays can be modeled as a
sequence of elements indexed by integers, so that `a[0]` is the first element
of the array, `a[1]` is the second element, and so on. We can think of hashes
as being arrays where the index doesn't have to be an integer. (Indeed, in
some languages, notably Perl, hashes are called "associative arrays" for this
reason.) Hashes are commonly used, among other places, in Ruby web frameworks
such as Sinatra and Rails, and understanding them is essential to using these
frameworks.

Recall from Listing~\ref{code:empty_array} that we can write an empty Ruby
array as `[]`. Similarly, an empty Ruby hash is written using curly braces:
`{}`. As with arrays, the best way to learn about hashes is to play with them
in an `irb` session:

```irb
>> h = {}
>> h['foo'] = 'bar'
>> h['baz'] = 'quux'
>> h
=> {"foo"=>"bar", "baz"=>"quux"}
```

Note that Ruby stores the hash elements in the same order that they were
defined. This ordering is maintained by design in Ruby 2 or later, but it's a
bad idea to rely on this fact; hash element order is not guaranteed by earlier
versions of Ruby, nor is it generally guaranteed in other languages.
```

```
.
.
.

#### Exercises
\label{sec:execises_hashes}

1. Define a hash with keys equal to `"foo"`, `"bar"`, and `"baz"`, with values
equal to those same strings reversed.
2. By reading the Ruby API on hashes, learn how to iterate through a hash.
Apply this to the hash created in the previous exercise by printing out each
key/value pair.
.
.
.
```

Comparing Listing 3 with Listing 4, we see that the latter expands the discussion considerably while taking care to ground the discussion of hashes in previous material, in this case referring to a previous section on arrays. Note that we've also added transitions between sections, such as "We saw…" and "Recall from…", which help ensure a smooth flow from one subject to another.

In order to examine the changes more closely, it's helpful to look at specific changes made between Listing 3 and Listing 4. For example, Listing 4 expands

```
Hashes are like arrays, but where the index doesn't have to be an integer.
Very common in Ruby web frameworks like Sinatra and Rails, will use
frequently
```

from Listing 3 to the much longer

```
We saw in Section~\ref{sec:arrays} that Ruby arrays can be modeled as a
sequence of elements indexed by integers, so that `a[0]` is the first element
of the array, `a[1]` is the second element, and so on. We can think of hashes
as being arrays where the index doesn't have to be an integer. (Indeed, in
some languages, notably Perl, hashes are called "associative arrays" for this
reason.) Hashes are commonly used, among other places, in Ruby web frameworks
such as Sinatra and Rails, and understanding them is essential to using these
frameworks.
```

An even more dramatic expansion happens to the throwdown content

```
start with empty hash {}
```

which expands to

```
Recall from Listing~\ref{code:empty_array} that we can write an empty Ruby
array as `[]`. Similarly, an empty Ruby hash is written using curly braces:
`{}`. As with arrays, the best way to learn about hashes is to play with them
in an `irb` session:
```

Such expansions of 1–3 lines into full paragraphs are not uncommon during the main exposition, which is one reason why I consider it to be the hardest step to complete.

It's worth noting that Listing 4 also adds some exercises at the end of the (sub)section:

```
#### Exercises
\label{sec:execises_hashes}

1. Define a hash with keys equal to `"foo"`, `"bar"`, and `"baz"`, with values
equal to those same strings reversed.
2. By reading the Ruby API on hashes, learn how to iterate through a hash.
Apply this to the hash created in the previous exercise by printing out each
key/value pair.
```

Adding exercises is optional, but experience shows that they are usually appreciated by the readers of the tutorial. (Placing them immediately after the corresponding section helps to reinforce the material, although collecting them at the end of the chapter is also a possibility.)

The result of finishing the main exposition is a complete rough draft of our tutorial document (article or chapter). There's still a lot of work left, but at this point I usually feel like I'm "writing downhill".

3.4 Polishing

After the main exposition comes one or more *polishing passes*. In my experience, polishing is time-consuming but relatively easy, requiring a relatively low cognitive load and a correspondingly low activation energy.

An example of polishing appears in Listing 5, which represents a single polishing run through Listing 4. Don't worry about all the details; I suggest skimming Listing 5 and then moving on to the breakdown of specific differences that appears immediately after.

Listing 5: Polishing the material on hashes.

```
### Hashes
\label{sec:hashes}

We saw in Section~\ref{sec:arrays} that Ruby arrays can be modeled as a
sequence of elements indexed by integers, so that `a[0]` is the first element
of the array, `a[1]` is the second element, and so on. We can think of a hash
as being like an array where the index doesn't have to be an integer. (Indeed,
in some languages, notably Perl, hashes are called "associative arrays" for
this reason.) Hashes are commonly used, among other places, in Ruby web
frameworks such as Sinatra and Rails, and understanding them is essential to
using these frameworks.

Recall from Listing~\ref{code:empty_array} that we can write an empty Ruby
array using square brackets as `[]`. Similarly, we can write an empty Ruby
hash using curly braces: `{}`. As with arrays, the best way to learn about
hashes is to play with them in an `irb` session, as shown in
Listing~\ref{code:hash_irb}.

\begin{codelisting}
\label{code:hash_irb}
\codecaption{Defining a hash in \kode{irb}.}
```irb
>> h = {}
>> h['foo'] = 'bar'
>> h['baz'] = 'quux'
>> h
=> {"foo"=>"bar", "baz"=>"quux"}
```
\end{codelisting}

\noindent We see from Listing~\ref{code:hash_irb} that hashes consist of
key/value pairs, which can be written in terms of the "hashrocket" operator
`=>`:

```ruby
{ "foo" => "bar" }
```

\noindent (Here we've included additional spaces to follow a common Ruby
convention, but such spaces are ignored by the Ruby interpreter.) The key is
the string `"foo"` and the value is the string `"bar"`.
```

```
Note that `irb` prints the hash elements out in the same order that they were
defined. For example, the last line in Listing~\ref{code:hash_irb} displays

```irb
=> {"foo"=>"bar", "baz"=>"quux"}
```

\noindent instead of

```irb
=> {"baz"=>"quux", "foo"=>"bar"}
```

\noindent This is because the `"foo" => "bar"` key/value pair was defined
first and the `"baz" => "quux"` key/value pair was defined second. This order
is maintained by design in Ruby 2 or later, but it's a bad idea to rely on
this fact; hash element order is not guaranteed by earlier versions of Ruby,
nor is the order generally preserved in other languages.

The hash in Listing~\ref{code:hash_irb} uses strings (Section~\ref{sec:strings})
for both keys and values, but it's more idiomatic Ruby to use symbols
instead, as shown in Listing~\ref{code:hash_symbol_keys}. This usage is
especially common in Ruby on Rails.

\begin{codelisting}
\label{code:hash_symbol_keys}
\codecaption{Defining a hash using symbol keys.}
```irb
>> h = {}
>> h[:foo] = 'bar'
>> h[:baz] = 'quux'
>> h
=> {:foo=>"bar", :baz=>"quux"}
```
\end{codelisting}
.
.
.
```

Compared to Listing 4, we see that Listing 5 polishes the language a bit, changing, e.g.,

```
We can think of hashes as being arrays where the index doesn't have to be an
integer.
```

to

```
We can think of a hash as being like an array where the index doesn't have to
be an integer.
```

and

```
Note that the hash elements are in the same order as defined. This is true in
Ruby 2 or later, but it's a bad idea to rely on this fact, as generally
speaking hashes (also called *dictionaries* or *maps*) don't always guarantee
the order of their elements.
```

to

```
Note that Ruby stores the hash elements in the same order that they were
defined. This ordering is maintained by design in Ruby 2 or later, but it's a
bad idea to rely on this fact; hash element order is not guaranteed by earlier
versions of Ruby, nor is it generally guaranteed in other languages.
```

Listing 5 also remedies an important omission from Listing 4, namely, the definition of hashes as key/value pairs:

```
\noindent We see from Listing~\ref{code:hash_irb} that hashes consist of
*key/value pairs*, which can be written in terms of the "hashrocket" operator
`=>`:

```ruby
{ "foo" => "bar" }
```

\noindent (Here we've included additional spaces to follow a common Ruby
convention, but such spaces are ignored by the Ruby interpreter.) The key is
the string `"foo"` and the value is the string `"bar"`.
```

Note that this change also includes two occurrences of the LaTeX command `\noindent`, which suppresses new-paragraph indentation in PDF output.

Listing 5 also adds material on using Ruby *symbols* as keys, which tightens up the motivation since such keys are commonly used in Ruby web frameworks (Listing 6).

Listing 6: Adding material on Ruby symbols.

```
The hash in Listing~\ref{code:hash_irb} uses strings (Section~\ref{sec:strings})
for both keys and values, but it's more idiomatic Ruby to use symbols
instead, as shown in Listing~\ref{code:hash_symbol_keys}. This usage is
especially common in Ruby on Rails.

\begin{codelisting}
\label{code:hash_symbol_keys}
\codecaption{Defining a hash using symbol keys.}
```irb
>> h = {}
>> h[:foo] = 'bar'
>> h[:baz] = 'quux'
>> h
=> {:foo=>"bar", :baz=>"quux"}
```
\end{codelisting}
```

Finally, Listing 5 includes two code *listings* (as opposed to mere code samples). The first, on defining a hash in **`irb`**, converts a previous code sample from Listing 4 into a listing, while the second, on symbol keys (Listing 6), is a new addition. In both cases, the reason for the change is to make it easier to refer to the particular listing later on. This is especially useful if the listing is important enough to justify a cross-reference at some other place in the tutorial. (For example, Listing 6 was a regular code listing after the main exposition of the present meta-tutorial, but was promoted to a code listing during the second polishing pass when I realized I wanted to refer to it.)

The above examples are designed to give you the general idea of what a polishing pass might accomplish. You'll have to use your own judgment based on the particulars of the tutorial you're working on.

3.5 Shipping

In the words of Steve Jobs: *Real artists ship.* As tutorial-artists, our job isn't done until we've *shipped*—that is, until we've released our work to the world. Once we've gone through a couple of polishing passes, it's time to nudge our fledgling tutorial out of the nest and get it to fly.

How you ship depends on your style, your current audience (if any), and your tools, but in my case shipping mainly means (a) deploying the tutorial to a custom URL on Softcover (e.g., www.railstutorial.org) and (b) sending an announcement out via my email list, Twitter & Facebook accounts, and Hacker News.

When I first ship a product, even though it has typically gone through a couple of polishing passes (Section 3.4) and is generally in good shape, I still like to describe it as a "draft". This is mainly in order to manage expectations—better for the document to be more polished than people expect than for them to be disappointed at bugs or typos—and it also sets a good frame for soliciting feedback.[16] In addition, I typically ship the first public draft as a free online version of the product, which maximizes readership (and SEO). This effectively lets me outsource the copy- and technical editing to my readers. (I find that many readers enjoy being involved in this way, and my gratitude for their help is genuine.)

After incorporating reader feedback for a while (which often involves another polishing round or two), it's time for the "official" launch. This gets promoted via the same channels mentioned before (email list, Twitter, etc.), and typically coincides with the official start of product sales. If, as is often the case, I also make screencasts based on the written tutorial, that's a third launch event. And after *that*, it's usually time for a break. (I hear French Polynesia is nice this time of year… (Figure 14).[17])

4 Tricks of the trade

Having discussed some general principles (Section 2) and the main steps of successive refinement (Section 3), in this final section we'll discuss some miscellaneous tricks of the trade to keep in mind when writing narrative tutorials.

[16] Some people like to use publishing platforms that allow comments right on the document, but I don't like this for error reports because they're hard to process as a queue, and as soon as you fix the error the comment is out of date. My preferred feedback channel is good ol' email.

[17] Image copyright © 2014 Michael Hartl, hereby released under a Creative Commons Attribution license.

Figure 14: French Polynesia is a nice place for a break.

4.1 Floats

Floats are objects whose position is not fixed in a print document or PDF, but rather "float" for the purpose of achieving a clean layout. Even though the position *is* fixed in HTML and HTML-based formats like EPUB and MOBI, for consistency we use the term even in these contexts. Floats are mainly added during the content throwdown (Section 3.2) and main exposition (Section 3.3), although it's not unusual to add a few floats during the polishing step (Section 3.4) as well.

4.1.1 Figures

Figures are probably the most common kind of float. Although figures can consist of ordinary text, the vast majority of the time they involve embedded images, as seen in Figure 15. As with sections and code listings, figures can be cross-referenced; indeed, they *should* be referenced, and the rule is that figures should be referenced in the order they appear.

Figure 6: Goldilocks chooses examples that are just right.

Choosing such examples takes pratice and judgment, but one useful guideline is, per Section 1.1, to defer that which can be deferred. Thus, the signup page from the *Rails Tutorial* shown in Figure 3 is challenging but not *too* challenging because it defers the more advanced material to a future section. When done well, such Goldilocks-style examples can induce a state of *flow* that makes learning immersive and fun (Box 1).[8]

Figure 15: A floating figure in a PDF document.

Figure 16: Do these doughnuts count as edible or dangerous?

Although figures are frequently used to illustrate key conclusions (data plots, graphs, etc.) in academic-style articles and books, one trick of the trade when writing tutorials is to include pictures of things that are cute, edible, dangerous, etc. (Figure 16).[18] This is a neat neurological hack that helps reset the reader's attention span by hooking into humans' natural interest in cute, edible, and dangerous things.[19] It's a good idea to ground such figures in the discussion, but they can still be effective even if the connection is quite tangential (e.g., Figure 2).

4.1.2 Tables

Tables are useful for collecting and summarizing information found elsewhere in the text, effectively combining the merits of a narrative tutorial with those of a reference work. For example, each section of *Learn Enough Command Line to Be Dangerous* ends with a table like the one shown in Table 1.

More information on producing tables with Softcover can be found in "Tabular and tables" in *The Softcover Book*.

[18] Image retrieved from https://www.flickr.com/photos/arndog/3536129634 on 2015-08-12

[19] This approach was pioneered by the "Head First" series. I learned about the technique from Giles Bowkett.

Command	Description	Example
`curl`	Interact with URLs	`$ curl -O example.com`
`which`	Locate a program on the path	`$ which curl`
`head <file>`	Display first part of file	`$ head foo`
`tail <file>`	Display last part of file	`$ tail bar`
`wc <file>`	Count lines, words, bytes	`$ wc foo`
`cmd1 | cmd2`	Pipe `cmd1` to `cmd2`	`$ head foo | wc`
`less <file>`	View file contents interactively	`$ less foo`
`grep <string> <file>`	Find string in file	`$ grep foo bar.txt`
`grep -i <string> <file>`	Find case-insensitively	`$ grep -i foo bar.txt`

Table 1: An example table.

4.1.3 Aside boxes

One optional but useful trick of the trade is using *aside boxes* for discussions important enough to include in the document but too extensive or tangential to include in the main body of the narrative. For example, Box 1 contains an aside on the concept of *flow*. One benefit of aside boxes (as opposed to merely shaded callout areas) is that they can be referenced later, as described in Box 3.

I usually find that aside boxes emerge naturally as stubs (temporary placeholders) at the throwdown stage (Section 3.2) and then get filled in during the main exposition (Section 3.3) and polishing passes (Section 3.4), but your mileage may vary.

4.2 Writing style

Every author has a *voice*—the unique syntax, diction, example types, etc., that characterize that author's writing. Based on my experience writing tutorials, I can offer some suggestions on developing a personal style conducive to effective teaching.[20]

My suggestion is to maintain an even, instructional tone when writing a narrative tutorial. There's room for variation, though, and the particular tone

[20]This is especially relevant to authors interested in collaborating directly with me; it is convenient if such authors emulate my writing voice as closely as possible.

should be calibrated to the type of document. For example, the *Ruby on Rails Tutorial*, which teaches a challenging subject (web development) to a general audience, is written exclusively in a friendly, helpful tone, whereas *The Tau Manifesto* is written in a more aggressive, polemical tone, appropriate for its role as an attack on the number π.

One advantage of maintaining a neutral tone is that it makes it easy to drop in the occasional bit of dry humor. For example, many of the figures in this tutorial (such as Figure 7 and Figure 10) are humorous in nature, and most of the humor derives from the contrast with the rest of the content. This sort of humor only works when it's the exception, though, and not the rule—one common newbie error is making light of practically *everything*, which can be intensely frustrating for the reader. When you're struggling with a hard subject, the last thing you want is a tutorial that makes a joke every other sentence.

Don't be afraid to show some personality in your writing, but don't let it overwhelm the instructional purpose of the document. The responsibility of the tutorial author is to convey useful information. Entertaining the reader is welcome to the extent that it supports this goal, but (in the context of an instructional document) such entertainment is not an end in itself.

4.3 Navigating and editing text

One challenge when writing narrative tutorials is navigating and editing potentially large text files. For example, as of this writing the *Rails Tutorial* has over 134,000 words. In my experience, finding your way around large documents is best done by using the search feature of your text editor. I can usually remember a few words related to the subject I want to find, so I'll hit `Cmd-F` in my editor and type a word or two to find the relevant place in the source.

This search-for-a-word trick is especially useful when editing the document using an output format such PDF or HTML. For example, when viewing this tutorial using Softcover's HTML previewer, I noticed that the output in Section 4.2 showed pi instead of π, as seen in Figure 17. Wanting to find the corresponding place in the document source, I looked for a word or word combination likely to be unique, and seized upon "polemical". A quick search for this word yielded the right location (Figure 18) and allowed me to make the

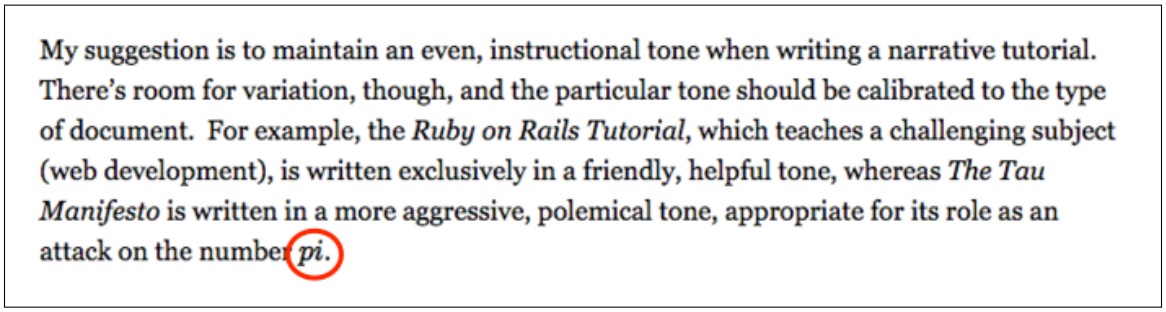


Figure 17: Noticing an error in the HTML preview.

```
My suggestion is to maintain an even, instructional tone when writing a
narrative tutorial. There's room for variation, though, and the particular
tone should be calibrated to the type of document. For example, the \emph{Ruby
on Rails Tutorial}, which teaches a challenging subject (web development), is
written exclusively in a friendly, helpful tone, whereas \emph{The Tau
Manifesto} is written in a more aggressive, polemical tone, appropriate for
its role as an attack on the number~$pi$.
```

Figure 18: Finding the Figure 17 error in the document source.

desired correction (changing `$pi$` to `$\pi$`), as shown in Figure 19. (Incidentally, you might notice that, in the time since I took the screenshots, the text in this document has diverged from that shown in Figure 19. This is the result of the second and third polishing passes (Section 3.4).)

4.3.1 To-dos

Another useful trick is to add "to-dos" to the document when you think of something to add but don't want to take the time to write the full content right at that moment. My favorite technique is to use "triple stars" (three asterisks) followed a brief note-to-self regarding the material to add. For example, this paragraph itself was a "triple star" addition, and originally looked like this:

```
***triple stars
```

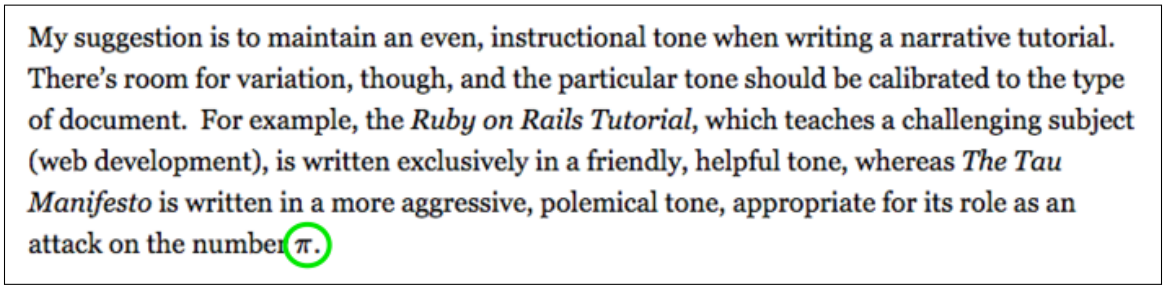


Figure 19: Correcting the error in Figure 17.

The main virtue of using triple stars is that `***` rarely appears in any other context, so it's easy to find them using a text editor's search function, which in turn makes it easy to expand them during one of the polishing passes (Section 3.4). *Note*: Triple stars work well in LaTeX, but in Markdown the asterisk character introduces a beginning *italicized* section, so in this case I recommend "triple plus" (e.g., `+++add note about triple stars/triple plus technique`).

4.3.2 Two panes

Finally, a helpful complement to searching and editing is the old "two pane" trick achievable in any good text editor. When searching through the document for whatever reason (to fix an error, look up a label for a cross-reference, find a particular string, etc.), it's usually inconvenient to move the cursor and hence lose our place. In this context, it's useful to have the same file open in two different text editor windows, as shown in Figure 20. This way, we can use one pane as the main writing area and the other pane as a sort of "random access" window for moving around in the document. This isn't an essential technique, but experience shows that it is extraordinarily useful, especially during the main exposition (Section 3.3) and polishing passes (Section 3.4).

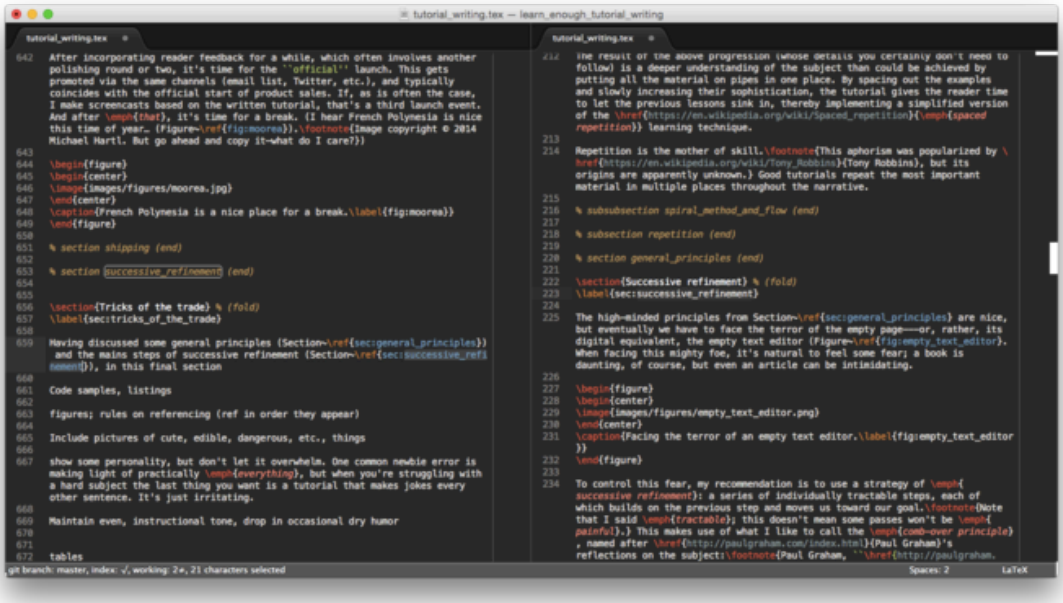


Figure 20: Opening the same file in two different panes.

Figure 21: A nice tweet about the Ruby on Rails Tutorial.

5 Conclusion

Writing narrative tutorials is challenging but can be highly rewarding. With hard work and a little luck, you might get tweets like the one shown in Figure 21. With a *lot* of luck, you might get an endorsement like the one shown in Figure 22. Although I'm a proponent of getting paid for your work, and indeed the Ruby on Rails Tutorial is a successful product business, at the end of the day knowing you've helped people learn a difficult and valuable skill is the best reward. (Admittedly, funding the occasional trip to French Polynesia (Figure 14) isn't bad, either.)

As an aspiring tutorial creator, you will ultimately need to develop techniques that fit your particular style and goals, but my hope is that this meta-tutorial will give you a good head start. By following the general principles of *motivation* and *structure*, incorporating techniques such as deferring material (Section 2.1), using concrete examples (Section 2.2), and harnessing the power of repetition (Section 2.3), you'll be able to produce an engaging and instructive narrative. By following the steps of successive refinement (Section 3)—free association, content throwdown, main exposition, polishing pass(es), and shipping—you'll make and ship an informative and helpful narrative tutorial. Finally, by applying some tricks of the trade (Section 4), you'll ease your authorial burden while livening up the presentation, thereby keeping the reader's

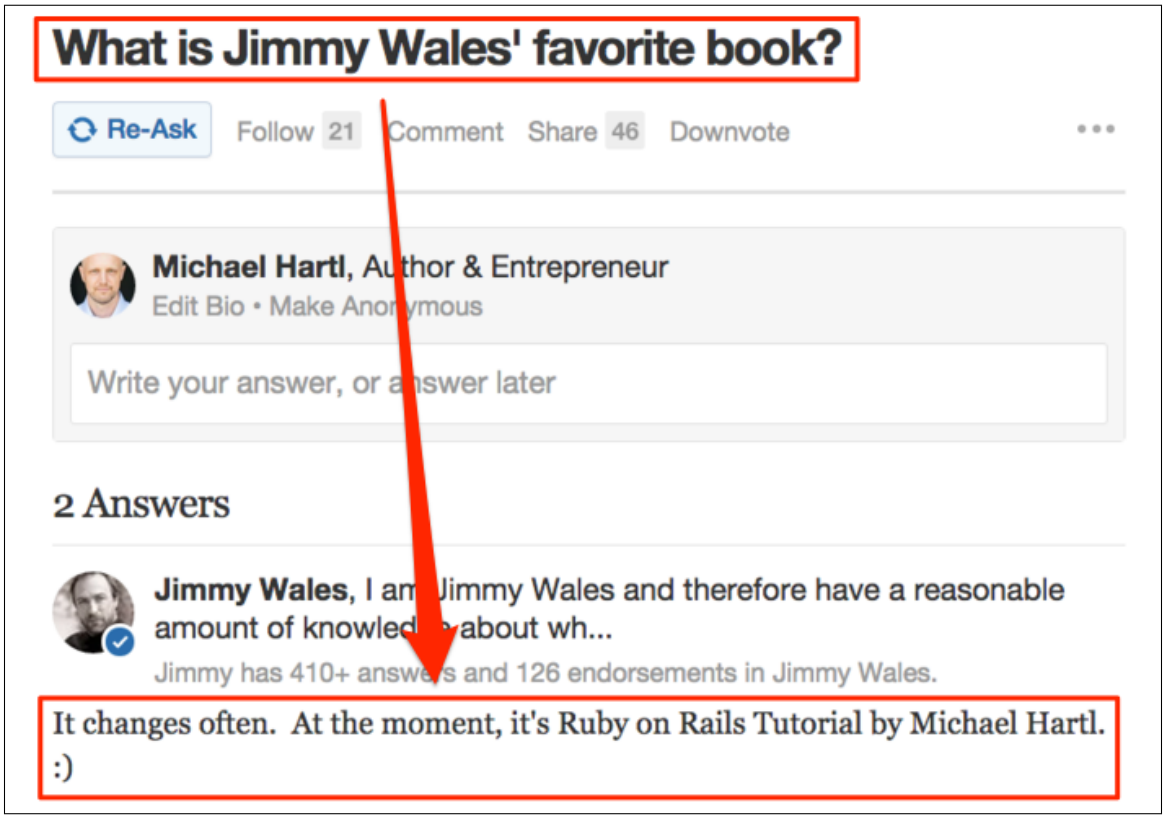


Figure 22: A surprise endorsement from Wikipedia founder Jimmy Wales.

interest while instructing gracefully and effectively.

Writing a quality tutorial is rarely easy, but the results can be deeply satisfying. Good luck!