Reading Technical Documentation

Written by Tiffany Chan for the Digital Scholarship Commons, UVic Libraries

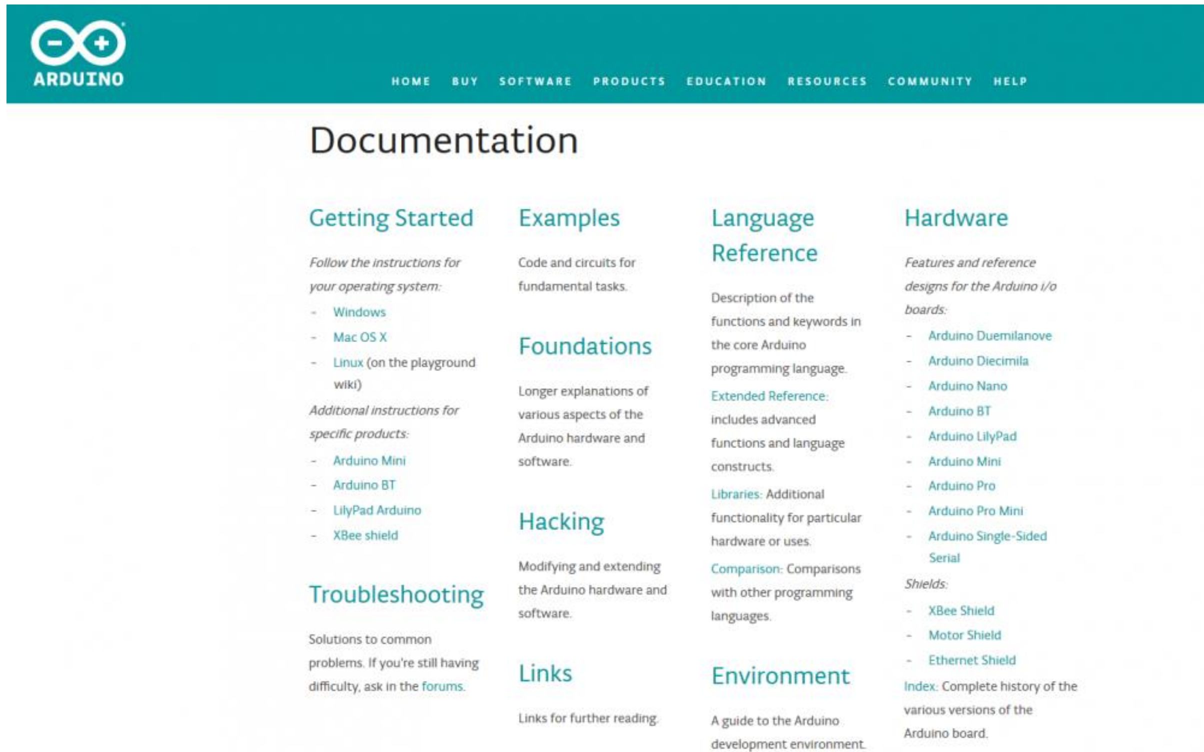
Technical documentation is any material meant to accompany a particular tool, software, or technology which can be descriptive (what problem does this solve?) or instructional (how-to). Although documentation ("docs") comes in various formats, this guide focuses on software documentation in textual format published on the web.

Here's the most common scenario in which I read documentation: I'm just starting a project and have a specific goal in mind, but I don't how to go about doing it. I may not even know what programming language, device, etc. to use. Can I do this with something I already have or should I install a dedicated application for it? If I want to make a sound-reactive light display, should I use Arduino or Raspberry Pi? What physical parts (LEDS, microphone, etc.) do I need? Which seems less complicated and time-consuming?

In short, I'm reading docs because I want to know what steps are involved and what the final product might look like—without investing too much time and resources upfront or reinventing the wheel. However, reading documentation is like trying to find a word in a dictionary without knowing how to spell it: how do I find what I'm looking for when I don't know what to look for?

This guide walks through types of documentation, how to parse them, and common conventions in web documentation (e.g. README files on Github). It provides advice and tips to help you navigate docs and sift the dizzying amount of information on the internet. Although much of this advice holds true for many types of software, I'll be using examples from Python, Machine Learning, Command Line, Arduino, JavaScript, and Twine.

Types of Documentation



The documentation page for Arduino, a physical computing technology.

Documentation can look very VERY different from source to source, and not all documentation is well-written (arguably, most is not). Docs vary in organization and level of detail, among many other ways. What follows is a brief, non-exhaustive typology of stuff on the web:

Tutorials

Usually, step-by-step instructions to accomplish a specific task. Even if it's not exactly what you're looking for, similar projects might still have useful code you can modify. See examples from Programming Historian.

Topical Guides

Information about a specific subject or feature. See example about Items in the Internet Archives library and API.

Reference Guides

This type of doc most resembles a literal dictionary. It typically provides a bare-bones description of a specific function or command. Especially useful if you know vaguely what you

need but you forget the syntax (what you actually type in). See example: Arduino's reference guide for functions.

Cookbooks

A collection of code examples or recipes for a specific software. I've never found this organically online (examples are often rolled into other types of docs), but it seems pretty common in e-book or book format. You can find a lot of them through UVic Libraries and probably other library search portals too.

Help Forums

Stack Overflow is the most (in)famous and often most helpful, although you can also find stuff in GitHub discussions or sometimes even reddit.

README Files

This is what you'll find on GitHub. See example: torch-rnn docs by Justin Johnson. Most README files are split into sections:

- *Installation:* How to install something step-by-step.
- *Dependencies or Requirements:* Other things that need to be pre-installed for it to work (e.g. code libraries or other software).
- *Support:* Whether the software requires a specific operating system (e.g. Windows) or version (e.g. Python 3 vs. Python 2).
- *Getting Started, Quickstart, etc.:* Probably the most useful part of the doc. Usually contains simple examples to demonstrate a software's capabilities.
- *Examples:* This may be dispersed across sections or silently packaged in one folder named "examples" or "samples" or something similar.
- *Features:* Things the software can do.
- *License:* Any conditions or rules for distribution of the software.
- *Updates or Release History:* Notes about new releases, fixes, and the like. If a piece of software seems outdated, it may be depreciated and non-functional.

Note: Most documentation incorporates aspects of more than one type. In fact, it's rare to find a doc that doesn't.

Reading and Using Documentation

```
#Define function for combining sounds, synthesizing into one chord
#freq = pitch, coef = amplitude, datasize = length, fname = name of wav file
def synthComplex(freq=[], coef=[], datalength=10000):
    #returns a tone for each number in datasize and appends it to a list
    for x in range(datalength):
        samp = 0
        for k in range(len(freq)):
            samp = samp + coef[k] * math.sin(2*math.pi*freq[k]*(x/frate)) #the equation f
        sine_list.append(samp)

#Synthesize the sounds
synthComplex([800, 300], [1, 1], 8000)
synthComplex([600], [1], 20000)

#Write the sounds into a single .wav file
def synthMelody():
    wav_file=wave.open("tone1.wav","w")
    nchannels = 1
    sampwidth = 2
```

Python code written in the Sublime Text Editor.

1. Narrowing Your Search Terms

This is half the battle, since knowing an accurate search term will yield better results. For example, say I want to make a light display with Arduino that flashes different colours. Googling “arduino light display sound” might give me a wide range of results, some of which are unrelated to my project. But based on that, I might find that “sound-reactive LED arduino” or “arduino music visualizer” is closer to what I’m looking for. Additionally, consider if you can refine your search using boolean operators or other methods.

2. Skim It

No one reads documentation for the plot. Like other kinds of research, you’ll likely skim docs and slow down if something catches your eye. It’s often helpful to look at an overview or table of contents section first. You might even stumble across a solution to another problem in the process.

3. Code and the Command Line

In case you haven’t guessed already:

```
Text formatted in code blocks like this are commands
to be typed in as is.
```

Code can also be `formatted inline`. Often, the code blocks contain commands for Linux/Bash Command Line. You can find this in Terminal on a Mac or [Bash on Ubuntu on Windows] in Windows 10. For older Windows operating systems, you can try [Git BASH]. (For more on the Command Line and what it's for, see this [excellent explanation] by Jojo Karlin, Jonathan Reeve, Patrick Smyth, Steven Zweibel. You can also play around with the Command Line in [DH Box] without needing to set it up or accidentally deleting something.

How can you tell if some code is meant for the Command Line? You might see things like this:

```
#This command installs some software named some_program
sudo pip install some_program

#This command clones/downloads a Github repository
git clone https://github.com/somerepo

#This command executes code (named some_script.py) written in Python 3
python3 some_script.py

#This navigates into a directory named some_dir
cd some_dir
```

With most code blocks, the documentation writer likely won't specify what language it's in. Readers are expected to know from the syntax of the commands and this can be confusing for beginners.

Let's look at an IRL example from the [torch-rnn documentation] written by Justin Johnson:

```
# Install most things using luarocks
luarocks install torch
luarocks install nn
luarocks install optim
luarocks install lua-cjson

# We need to install torch-hdf5 from GitHub
git clone https://github.com/deepmind/torch-hdf5
cd torch-hdf5
luarocks make hdf5-0-0.rockspec
```

This is a series of commands you would type into Command Line (having installed Lua). Note that you have to press Enter to run something in Command Line, so make sure you do that

after every line. The words `#after the hash` are comments (we'll talk about them later), so you don't need to type them into the Command Line console.

4. Copy/Paste

When I talk to people in my home discipline (English Literature), some are reluctant to copy/paste and use code they find on the internet in their own programs. In the Humanities, we're taught that using other people's work without giving credit, formatted according to specific conventions, is plagiarism and should be avoided.

For better or worse, this is generally not the way coders operate. If anything, copy/pasting code is encouraged. Not only does it save you time and effort (and possibly a lot of frustration), but you can be reasonably confident that it will work and that you haven't introduced any errors by mistyping something. Of course, this should be balanced with opportunities to learn by writing your own code (even manually type out something you've found can be a good learning experience).

However, chances are that you'll write a bit of original anyway, since you'll probably have to modify anything you find to suit the situation at hand. At the very least, you should change variable names (what's a variable?) to something more descriptive for your project.

5. Useful Options, Commands, or Keyboard Shortcuts

Here are some tips for working with code or text editors such as Notepad++, Sublime, or Atom.

Note: The exact keys vary depending on the editor and operating system (the below is meant for Windows PC). If you're using a Mac, replace Cntrl with the Command button.

a. Word Wrap

This is usually under the View dropdown menu in your text editor. Basically, it shifts words to a new line when it would otherwise continue outside the window—it's similar to how Microsoft Word automatically jumps to a new line once you've reached the maximum width of your document. Even if you have word wrap enabled, the code will still execute as if it were all written on the same line (this is very important in coding!).

b. Find and Replace: Cntrl + h

This is especially helpful for renaming variables, which can happen if you discover that a name conflicts with something else in your code, or for other reasons. Many text editors have extra options such as matching case.

c. Indentation: Cntrl + [(increase indent) or Cntrl +] (decrease indent)

Indentation is meaningful in most, if not all, coding languages. For example, code inside conditionals or loops often needs to be indented. As a visual cue, it's also more human-readable that way!

d. Commenting Out and Uncommenting: Cntrl + /

To "comment out" something means to turn a block of code into a comment, thereby making it inert or dormant (the computer won't run it). More specifically, changing that code into a comment signals the computer to skip those lines when running the program. "Uncommenting" is the reverse: your comment becomes executable code.

Some examples of comment format/syntax:

```
#This is a comment in Python.

//This is a comment in Arduino.

//This is a comment in JavaScript.

<!--This is a comment in Twine's Harlowe format. (same as HTML)-->
```

This is a handy and non-destructive way of "erasing" code without actually deleting it. Sometimes, once I've progressed to a certain stage, I create a duplicate or back up version of some code and comment it out. That way, if I play around with the code further and get stuck, I can always return to a clean copy that I know still works.

You can also use this technique to alternate between two options by commenting out the option you don't need and potentially uncommenting it later when you need it. For example, say I want a Python script that I can apply either to a list of specific files or to every file in a specific directory. I could switch between the two like this:

```
#If you only want a specific set of files in your source folder:
# listOfFiles = [] #list file names (separated by commas) in the square brackets

# for file in listOfFiles:
        #OR
#Iterate over each file in the directory
for file in os.listdir(sourcePath):
    fileName = file
    #print(fileName)
    #Open the stripped .txt file & read it
    contents = open(sourcePath + fileName, 'r+')
    fulltext = contents.read()

    '''Split the file into lines according to line continues
    (as they appear in the raw text). Returns a list of lines.'''
    listLines = fulltext.splitlines()
```

I could uncomment `listOfFiles` *and the first* `for` *statement, then comment out the second* `for`*.*

Additionally, you can print variables or values (see below) to the screen to help with debugging and comment them out after.

6. If you think, "There must be an easier way to do this," then there probably is

As I mentioned at the beginning of this guide, avoid reinventing the wheel wherever possible. In practical terms, this might mean searching for a code library that does what you're looking for, rather than assuming you need to write something completely from scratch. If one library doesn't have a built-in option for your specific use case, a similar one might. Similarly, if a would-be solution is hard to get working, don't feel the need to make it work with brute force. There may be a different, less troublesome solution elsewhere.

When reading documentation, look for examples, screenshots, or videos that show you exactly what the end result of some code or process looks like. This will not only help with debugging by comparing what you expect to what you get; it'll also help you decide if what you get is close enough what you're looking for—or if you want to look elsewhere.

7. Print Early, Print Often

Printing things to the screen or console is usually one of the first command you learn in programming. Examples:

```
#In Python:
print(something)

//In Arduino (print to Serial Monitor):
Serial.print(something)

//In JavaScript:
console.log(something) //print to console
window.alert(something) //print to an alert box

<!--In Twine:-->
<<print $something>>
```

As mentioned above, you can print variables or values. This is helpful for checking if the output of a function is what you expected it to be. If you have a complex function with several discrete steps, or you pass the output of one function into the input of another, printing the output of each step/function can save you a lot of frustration. Otherwise, if something breaks, you'll have a harder time figuring out where it goes wrong!

Yes, there will probably be an error message pointing to a specific line or spot in your code anyway, but I still prefer to know earlier where possible. A decision you make in solving the error may have consequences in other code elsewhere (e.g. choosing to ditch a specific library).

For example, I like to leave a print command, commented out, floating around at the end of some code like this:

```
#Iterate over each file in the directory
for file in os.listdir(sourcePath):
        fileName = file
        #print(fileName)
        #Open the stripped .txt file & read it
        contents = open(sourcePath + fileName, 'r+')
        fulltext = contents.read()

        '''Split the file into lines according to line continues
        (as they appear in the raw text). Returns a list of lines.'''
        listLines = fulltext.splitlines()

#print(listLines)
```

8. Start Small and Scale Up

Another way to avoid frustration is to start with the smallest scope possible (in software development circles, called a Minimum Viable Product or MVP), and then increase the complexity gradually. I recommend saving the MVP separately before messing with it further. That way, you always have a workable version at hand.

Writing or testing code is an iterative or recursive process. It's been described as pushing a broken car down a hill, tinkering under the hood, and then pushing it back up and down the hill again. Put simply, coders run their programs over and over and over again before they work satisfactorily. This might seem odd to humanities scholars when we obsess over just the right words or turn of phrase. Although writing is also an iterative or recursive process, we often go through several stages of revision before "testing" it on anyone, especially if we're perfectionists.

However, in coding, writing big chunks of code before or without testing can end up working against you since we wind up in the same conundrum as in number 6: you know there's an error, but you don't quite know what it is.

9. Understanding Error Messages

If you've spent time copy/pasting solutions from the web, you may have noticed that said solutions seem to introduce their own errors. Here are some frequent examples:

Error	Sample Error Message	Description	Possible Fix
Missing library	*In Arduino:* error: 'FastLED' was not declared in this scope	Happens when you try to use a library that you don't have installed or forgot to include/import it.	Go to Sketch > Include Library > Manage Libraries and install the library you need (here, it's FastLED). If you installed the library but it's still not working, you may have forgotten to include it with #include<libraryName.h> (where libraryName is name of the library).
Missing variable	*In Twine:* Error: <<print>>: bad evaluation: myVar is not defined	_____ is not defined is a classic case of the missing variable. It happens when you call a variable without having defined or declared it (i.e. assigned it a value) beforehand. If you copy/paste something from a help	Declare the variable somewhere in the code ahead (in an earlier line) than where you need it. Note that this will also depend on the scope you want your variable to have. Do you need to use the variable in multiple functions? Do you want the its value to be the same for each of the functions or change as the program runs?

		forum, this can happen because whoever provided the solution used arbitrary variable names for the sake of demonstration. There could also be a scope issue.	It's helpful to understand global vs. local variables too, but be aware that different programming languages or applications might handle scope differently.
You're not my type	*In Python:* `TypeError: Can't convert 'int' object to str implicitly`	For many coding languages, variables and values can be sorted into several data types (e.g. integer, string, list). When you try to perform an operation on one type of data that is meant for another type, you get this error. In this example, Python is telling you that you're trying to treat an integer (a whole number) as if it were a string (of letters or characters or, in other words, text). This error occurs if you try to drop an integer into a prose statement.	Many programming languages have built-in ways of converting one data type to another. Here, you'd probably want the str() method. You might also want to double-check what data type something is to see if you've declared it correctly. If you're not sure what type a variable is in Python, you can use the type() method, which returns the type of a specific variable/value.
Syntax error	*In JavaScript:* `SyntaxError: missing ; before statement`	This is maybe the most annoying problem of the bunch but also the easiest to fix. Chances are, you missed a punctuation mark somewhere. (A classic one is a missing semicolon at the end of a line.)	Pretty straightforward: find and correct the error. Sometimes copy/pasting the search error into Google will bring up a Stack Overflow question from someone who made a similar misstep.

10. Read More Documentation

Hopefully these tips help, but there's no real substitute for reading lots of documentation yourself and experimenting with code. Given how much documentation varies from source to source, it's impossible to anticipate every potential situation you could encounter.

Over time, you'll get a feel for the scope and complexity of different projects. Where possible, try to come up with a project driven by your own interests rather than an arbitrary exercise. Like learning any language, even a programming language, navigating documentation is a skill earned through practice and the motivation to make/say something meaningful to you.