Chapter 21

INSTALLING SOFTWARE

Nearly everyone has installed software on their computer at some point. Many widely used programs come with double-clickable installers that make it simple to get a new program up and running. Installing more specialized software, including scientific command-line programs, can be a bit more involved. For many users, it is installing software, not using it, that becomes the most frustrating aspect of working within the command-line environment. In this chapter you will be introduced to basic installation procedures and cover background material that will help you understand how to address a range of installation scenarios.

Overview

In the simplest cases, installing command-line software that someone else wrote differs little from the steps you used to get your own programs running in earlier chapters. There are at least three things that have to happen in both cases. First, the program has to be in a language that your computer can understand. The programs you wrote in earlier chapters are in Python and `bash`, languages that can be interpreted by the software on your computer. Second, the program has to be placed in a folder where it can be found by the shell (i.e., in one of the folders specified by the `PATH` variable). Third, the program file has to have file permissions that make it executable. Satisfying these last two criteria should now be familiar; if not, review Chapter 6. In fact, if you have copied any of the scripts from the example folder to your own `~/scripts` folder you have already installed software written by someone else (us).

Installing tools for use at the command line sometimes is this simple. Usually, though, a bit more needs to happen. There are a variety of factors that can complicate the software installation process, which can be a source of frustration for beginners and a big time sink for experienced users. These complications include the following:

- Software isn't always provided in a language that can be understood as-is. Depending on the language, the program may need to be translated (compiled) to a language that can be understood by the computer. In other cases an interpreter may need to be installed.
- A new piece of software often needs other software packages, libraries, or modules to operate. These **dependencies** must be installed before you can use the new program. Dependencies often have their own dependencies, so it is not uncommon to need to install a dozen or more programs just to get the one you want to use running.
- Programs often need certain files and directories to record configurations, store intermediate files, and handle data input and output. Installation can include creating these folders.
- Software often needs to know things that are specific to your computer, such as the configuration of the Internet connection or the preferred program for handling certain file types. Installation scripts can either figure these things out automatically or ask the user to specify them.
- It is often necessary to modify and update existing files, such as `.bash_profile` and other configuration files, to use a new program. Again, the installation process can take care of this automatically, or you may be required to manually change some settings.

The rest of the chapter will explain some of these issues in more detail, and describe how they are typically approached. The skills you will develop can also enable you to use some software on computers the programmers didn't explicitly support. Specifically, you can often install basic Unix command-line programs under OS X simply by recompiling them on your own system.

Interpreted and compiled programs

The difference between interpreted and compiled programming languages was briefly mentioned in Chapter 7. At that point the distinction may have seemed relatively abstract, but it becomes relevant when installing a new program. Microprocessors, the brains of your computer, can only understand their own native machine language. In order to run, a program must either be interpreted (translated on the fly by another program such as Python or Perl) or already be in this native machine language.

Interpreted programs are stored and executed as text files containing a series of commands that can be read by both humans and the interpreter. The programs you wrote in earlier chapters were all built with interpreted languages, and the interpreting was done by the program indicated at the shebang line.

Programs in the native machine language, on the other hand, are stored and executed as binary files rather than as text files; for this reason, such programs are

often simply referred to as **binaries**. Machine languages are designed for the convenience of digital logic circuits rather than of people, and it is rare that anybody directly writes binary code. Usually such files are written in another, more human-friendly language, and then translated into a binary file for subsequent use. The original human-friendly program file is referred to as the **source code**, and this one-time translation to a binary is called **compiling**. We won't describe the computer languages that are commonly used to write source code, but we will describe in this chapter standard procedures for how to compile and install programs from source code. Common compiled computer languages include C, C++, and Fortran. Most GUIs are developed in this way and draw upon platform-specific libraries to handle things like graphics and user-interface elements. This process of compiling from source code is a focus of the present chapter.

A **compiler** is a program that translates source code into an executable binary. There are many compilers, including the widely used `gcc` program on Unix computers. For example, if you create a text file including the short program `rosetta.c` from Appendix 5 (available in the `pcfb/scripts` folder), and then type `gcc rosetta.c` at the command line, this will lead to `gcc` creating an executable file, called `a.out` by default. This `a.out` file can be run by typing `./a.out`. (The reason for the `./` is explained later in this chapter.) Using `gcc` directly is the most bare-bones way to create binaries. It does not readily take advantage of other libraries or user preferences during the compilation. To have a more flexible compilation method, programmers typically use the program `make`, which reads from a settings file and handles all of the housekeeping before calling `gcc` with all the dependencies and options specified.

Dependencies and platform-specific aspects are not limited to compiled binary files. Interpreted programs have similar requirements—specifically, a pre-compiled interpreter for the language that the program is written in, and any libraries or modules needed. OS X and Linux both come with interpreters for common languages, but if you are running Windows, you will have to install such interpreters yourself.

OPEN SOURCE SOFTWARE There is a way to reap the advantages that come with the speed of compiled binaries and the transparency that comes with human-readable computer code—namely, by distributing source code in addition to or instead of binaries. This is the **open source** model, in which users can inspect the source code to see what it is doing, as well as compile the binary executable themselves so that it is optimized for their own computer. Open source programs are also usually licensed in such a way that anybody can use and modify them, as long as they apply the same license when they pass on the software to others. Open source licenses are often applied to interpreted programs as well, even though the program file is the source file. This just makes it clear that it is okay to distribute and modify the program.

Increasingly, software written by scientists is distributed according to the open source model. To the uninitiated, it sometimes comes as a shock and even an outright offense that they are expected to compile their own binary from the source code. It is actually a courtesy.

Some of the most popular repositories for open source software are `sourceforge.net`, `github.com`, and `code.google.com`. Python programs and modules, some of which have special installation requirements, can be found at `pypi.python.org`, and Perl programs are archived at `www.cpan.org`. When you download and install software, you are exposing the guts of your computer to the wilds of the world—do not take security lightly, and only install software from trusted sources.

Approaches to installing software

This chapter applies mostly to installing binaries of command-line programs that operate in the shell. There are three basic approaches to installing binaries on your computer. In order of increasing complexity, these are: using binaries that have already been compiled; using a package manager system; or compiling the binaries yourself. Usually you will start with the simplest method available for your platform.

`Readme.txt` *and* `Install.txt`

Software is often distributed with a file called `Readme.txt` or `README`. This file usually has a general explanation of what the program does, who wrote it, what kind of license it is distributed under, its history, and, most importantly, any out-of-the-ordinary details you need to install it. It is easy to get blasé about `Readme.txt` and proceed with installation without opening it, but taking a quick look can save a lot of time later. It is also a good place for a beginner to start, since it often gives a step-by-step guide to the installation process, even in cases where nothing is out of the ordinary.

Installation details are sometimes stored separately from `Readme.txt` in a file called `Install.txt` or `INSTALL`. Often, though, `Install.txt` is a configuration file for installation scripts rather than directions for the would-be user. Open the files in a text editor or view them using the `less` command to find out.

Installing programs from precompiled binaries

Just because a program is open source doesn't necessarily mean that you have to install it from source code; it just means that the source code is available. Many authors of open source software provide compiled binaries for a variety of computers, right alongside the source code. If you don't need to inspect or modify the source code and the binary is available for your operating system, you might as well just download and install the binary. Sometimes the binary is available as a separate download. In many cases the source code is distributed as a compressed folder, and when you expand the file it includes some binaries with the source code files.

From the user perspective there is often little difference between an executable binary and an executable text file. They can be located in the same places in the file system and called from the command line in the exact same way. Installing an executable binary file can be as simple as installing an executable text file. Although the binary does have to be compatible with your computer type, you don't need to have an interpreter installed since it is in machine language that can be directly understood by the microprocessor. The file does, however, need to have executable permissions and be in your path. Refer to Chapter 6 if you need a refresher on this.

Automated installation tools

The second method of installing software uses a variety of tools known as **package management systems**, which can install open source software with as few as two

clicks. These programs not only download and compile the source code (or download a binary for your system if it is available), they also install any dependencies and take care of system configuration if need be. This can save a lot of time, and doesn't require much expertise. If you find that you need an open source program that isn't already installed on your computer, it is usually a good idea first to see if it is available via one of these tools rather than do all the work yourself.

On OS X, one of the most widely used systems for making open source programs available is Fink. You can download Fink from the project Web site at `www.finkproject.org`. The first time Fink is launched, it will update the list of available open source programs and you can select which of these you would like to install. Fink can be used from the command line, but also has a convenient GUI called Fink Commander.

Another option for OS X is MacPorts. This system has many advantages, including a large repository of software and automatic handling of dependencies. However, it also can be annoying, due to its insistence on installing its own versions of software that may already be available on your computer. For example, if you ask it to install a Python package, MacPorts will try to install its own instance of Python from scratch, a process that can take many hours. Once you have Python up and running, however, it is simple to add more packages. To install MacPorts, download the appropriate binary files from `www.macports.org`. Operations are done using the `port` command. To search for packages containing a keyword or part of the name, type `port search` *`keyword`*. Once you find a package, you must install it using the exact name of the package listed, with the command:

```
sudo port install packagename
```

For example, if you search for `biopython`, you will come up with versions that correspond to different versions of Python, such as Python 2.5 versus Python 2.6. This means you cannot just say `port install biopython`. Instead, you must specify `py25-biopython` or `py26-biopython` as the target.

Most Linux distributions come with their own easy-to-use software installation managers and GUIs, some of which include an extensive assortment of scientific applications. In Ubuntu Linux, you can add programs using the Add/Remove item of the Applications menu. You can also try the package manager called Synaptic, available via System ▸ Administration. From the command line, you can also use `sudo apt-get install` to install packages that have been prepared for Ubuntu. For other Linux distributions, you will want to investigate other package managers, including `yum` and `rpm`.

Installing command-line programs from source code

Sometimes, you may discover that a program you want isn't available as a compiled binary or via a package manager such as Fink. Other times, you may discover that you need to modify the program to make it work on your system. In all

such cases, the solution is to install from the source code. Because of their Unix underpinnings, both OS X and Linux give you access to many kinds of programs written for general Unix platforms. For command-line programs, you will often actually prefer downloading software written for Unix even when there is an OS X version available. This is because the Unix version is most likely to be updated first and is fully usable when compiled for your OS X computer.

Getting your computer ready

At a later date you may want to check the version of a particular program, modify it, redistribute it, or recompile it, so it is a good idea to hold on to source code. Make a folder called `src` in your home directory to store and build all the software you compile:

```
mkdir ~/src
```

Next, you need to make sure that you have all the software necessary to compile source code. At a minimum, this requires a compiler. Nearly all versions of Linux are distributed with a compiler, but it must be installed separately on OS X and Windows. OS X does come with a set of compiler tools, but they are not installed by default (at least not at the time of writing this book). They are distributed under the name XCode in the Optional Installs section of the installation DVD that came with your Mac, and they can be installed for free from there. If your install disk isn't handy, they can also be downloaded and installed from Apple's Web site at `developer.apple.com/mac/`; free registration is required.

Unarchiving the source code

Almost always, source code is a collection of files rather than a single file. These files are usually distributed as an archive that you can then expand on your own system. To save Internet bandwidth and hard drive space, this archive is usually compressed. Files will typically be a compressed archive—called a tarball—with the extension `.tar.gz` or `.tar.Z`; or they may come compressed in a `.zip` archive. These compression and archiving schemes were discussed in Chapter 20 in the context of remote servers.

We will walk through the process using ExifTool discussed in Chapter 19 as the first example file. With your web browser, download the source archive for the Unix version using the shortcut URL we have created at `tinyurl.com/pcfb-exif`. When you download the `Image-Exif` archive, your web browser may leave the file as a compressed archive; it may automatically uncompress the tarball (leaving you with a `.tar` file); or it may even go on to untar the `.tar` file, leaving you with the `.tar` file plus a folder representing the file contents.

In a Linux GUI, when you double-click a compressed file or archive, it will open a window that looks as through it is displaying the contents of the uncompressed archive. This is actually a preview of the archive's contents, and you have

to click the Extract button at the top of the window to complete the operation. If the archive is automatically expanded after downloading, move the folder containing the source code into your `~/src` folder. Otherwise, move the archive into your `~/src` folder. Then `cd` into `~/src`, uncompress with `gzip -d` or `gunzip`, and use `tar -xf` to separate out the individual files from the archive:[1]

```
mv ~/Downloads/Image-Exif* ~/src ← Your download location may vary
cd ~/src
guzip Image-Exif* ← Uncompress
tar -xf Image-Exif*.tar ← Unarchive
```

Compiling and installing binaries

Now that you have a folder with the source code on your computer in a convenient location, you would typically `cd` into the source folder and look around with `ls`. In some cases, the software developer will have distributed several precompiled binaries with the source code. If that is the case and there is one for your particular system, you can just `cp` it to `~/scripts` and you are ready to go. (Remember, programs have to be somewhere listed in the variable `PATH` for the shell to find them, and we already set up `~/scripts` as a place to put executables and added it to `PATH`. Alternatively, you could create a folder called `~/bin` for compiled binaries and add it to your `PATH`).

In other cases, you will usually find a file called `README` or `INSTALL` with specific instructions on how to compile and install the software. The other important file is called `Makefile`, which is a text file with instructions for the compiler on how to build a binary from the source code files.

In the case of ExifTool, you will have to build the `Makefile`, and then compile the executable files. After you execute the `tar -xf` command, you have a folder containing the source files and the configuration files needed to guide its installation. Move into the program's directory and begin the process of configuring, then building (with `make`), and finally installing (with `sudo make install`):

```
cd Image-ExifTool-8.23 ← Move into the newly created folder; version number may differ
perl Makefile.PL ← This program uses a Perl script to write the Makefile; some others don't
make ← Build (that is, compile) the program using the Makefile generated above
sudo make install ← Using your special privileges as superuser, install files in their locations
```

By default, the `make` command expects that the settings file is called `Makefile` (and remember, character case matters). If the file with instructions is called some-

[1]There are many variations on the commands used to uncompress and unarchive files. Some are designed as one-liners using shell commands piped together. You may also be able to uncompress and unarchive in one step, with `tar xfz archive.tar.gz`. In this case, there is no dash before `xfz` as there is in `tar –xf`.

thing else, such as `Makefile.OSX`, you need to rename the file, or tell make the name of the file using the `-f` argument:

```
make -f Makefile.OSX
```

The most common reason for the make file to have a name other than `Makefile` is that there are separate versions for different operating systems and computer architectures. If everything goes well, `make` will parse the `Makefile` and from it determine which compilers to use for which files and how to stitch everything together into a single executable binary.

In many installations, including this one, the `Makefile` includes information not only on how to build the binaries, but where to install them on the system and how to make other modifications necessary for the installation. These post-build processes are executed with the command:

```
sudo make install
```

This command usually takes much less time than the building of the binaries themselves. If this step succeeds, then the binaries are copied from the `~/src` directory into special system folders and each time you log back in you will have the new program available for use.

Variation 1: Off-the-shelf `Makefile`

A very simple compilation and installation procedure is to build the `agrep` tool, discussed in Chapter 16. Download the Unix source from `ftp://ftp.cs.arizona.edu/agrep` (there are several files available, but at the time of printing the relevant one is `agrep-2.04.tar`),[2] then move it to your `~/src` folder and unarchive the file. In this case, a `Makefile` is supplied off-the-shelf and doesn't need to be created with a script. This is common for simple programs that don't require computer-specific configurations. Within the unarchived directory, simply type `make`, and it should build the executable for you.

The supplied `Makefile` doesn't include any information on installing the program after it is compiled, so `make install` won't do anything. You need to manually place the executable file in the appropriate location on your system.[3] Move the `agrep` executable from the source folder to `~/scripts` or some other place in your `PATH`:

```
mv ./agrep ~/scripts
```

---

[2]Note that this is an FTP URL, not a Web URL. It will probably download using a different program than your web browser.

[3]Within OS X, to install the `man` file for `agrep`, copy it from the source folder with this command: `sudo cp agrep.1 /usr/share/man/man1/`.

Variation 2: Generating a `Makefile` *with* `./configure`

In the ExifTool example earlier, the command `perl Makefile.PL` generated your settings file for you, but this is not the most common case. If there is no `Makefile` in the source folder that you download, but there *is* a file called `configure`, then that is a separate script which will generate a customized `Makefile` to match your system capabilities. You will first run this command before proceeding with the `make` process. The `README.txt` file will typically explain how and whether the `configure` command is to be used. Sometimes there are options for what features will be enabled in the compiled program which can be specified at the time that you run the `configure` command.

This next installation example compiles the program ImageMagick, which takes advantage of the `configure` command. However, because it has many dependencies, this command may or may not work on your system as specified. If you cannot get it to work with these instructions, you might use this as a chance to test out Fink, MacPorts, or another package manager as the installation mechanism.

You may remember from Chapter 19 that ImageMagick is a powerful image processing system. It is actually installed by default on some systems (type `which convert` to see if it is already present on yours). In such cases, the procedure described here can be used to update your installation and to install the Unix manual pages.

For OS X or Linux, download the source files for the Unix version of ImageMagick from the project page at `http://sourceforge.net/projects/imagemagick/files`.

The appropriate file will be either a `.zip` or `.tar.gz` file without `windows` in its name. Move the file to your `~/src` folder, uncompress and unarchive as before, and `cd` into the folder that is produced.

As you know, a script will not run unless it can be found. We have talked about adding locations to your `PATH`; however, there are likely to be many configure files on your system, and you don't want to have to move them all into your `~/scripts` folder just to get them to run. Another way to run a script which is not in a folder in your `PATH` is to specify its full absolute path when typing the command. If the script is located in your current working directory, then you can use a shortcut to indicate its absolute location.

Recall that `cd ..` moves you to the folder enclosing the present folder, because `..` is a shortcut describing "the folder containing this one." Similarly, a single dot indicates the current folder of residence. Thus to specify the full path to the `configure` script in your current working directory when you try to run it, you can type the command preceded by `./`:

```
host:ImageMagick lucy$ ./configure
```

This `./` path specifier is similar to the `~/` shortcut, which inserts the full path to your home folder into the command line, but instead it inserts the full path of the working directory.

From within the ImageMagick source folder, run `./configure`, and it should print out a long list of status messages, ending with some details about what compiler it will use for building the executables. Upon completion, it has written that information into a customized `Makefile`. After it is done, you can proceed with the `make` command to compile the binaries of the required programs. This may take a long time!

For the ImageMagick installation, the instructions are found in the `Install-unix.txt` file. In this case, that file tells you how to enable or disable many optional features. If your installation fails because some of the dependencies cannot be installed or found, you may be able to get through the installation of basic functions by disabling those options, in particular, certain fonts, X11, or PostScript support (achieved with a package called Ghostscript). The choice to disable components is made at the time you run `./configure`, so that your `Makefile` is generated appropriately. If you have to go back and rerun `configure`, first follow the cleanup steps described later in this chapter. Other troubleshooting tips are included in the `Install-unix.txt` file.

Once ImageMagick is installed, you will most commonly use it via the `convert` command, so type `man convert` to read about the program's features and functions.

Installing Python modules

Python modules are often supplied with their own installation scripts. Such a script is usually contained in the source code folder and is called `setup.py`. To install a Python library, run the installation script by changing into the source directory, then typing:

```
sudo python setup.py install
```

This will perform whatever compilation and configuration steps are necessary to get the library working on your system. You don't have to specify `./configure` before `setup.py` because the executable that is being called is the `python` program, which is in your `PATH`.

As an example, try installing the `pyserial` library used for serial communications. Download the source archive `pyserial-2.5-rc2.tar.gz` (or the most recent version) from `pypi.python.org/pypi/pyserial/`. Move the archive file to your `~/src` folder, then uncompress and unarchive it; remember you can press tab to complete filenames at the command line:

```
host:~ lucy$ mv ~/Downloads/pyserial-2.5-rc2.tar.gz ~/src
host:~ lucy$ cd ~/src
host:src lucy$ ls
Image-ExifTool-8.23.tar.gz    agrep-2.04.tar
ImageMagick-6.6.2-8.tar.gz    pyserial-2.5-rc2.tar.gz
host:src lucy$ tar xfz pyserial-2.5-rc2.tar.gz ← Uncompress and unarchive
host:src lucy$ cd pyserial-2.5-rc2 ← Move into the newly created folder
host:pyserial-2.5-rc2 lucy$ ls
CHANGES.txt MANIFEST.in README.txt  examples    setup.py ← The installer file
LICENSE.txt PKG-INFO    documentation   serial      test
host:pyserial-2.5-rc2 lucy$ sudo python setup.py install ← Required for privileges
Password:
running install
running build
running build_py
...
changing mode of /usr/local/bin/miniterm.py to 755
running install_egg_info
Writing /Library/Python/2.5/site-packages/pyserial-2.5_rc2-py2.5.egg- info
host:pyserial-2.5-rc2 lucy$
```

That's all there is to it (when it works). You can see by the status updates that in addition to building the program, the `setup.py` script also copies the files to a central installation folder where they will be accessible to Python programs that you write. In our experience, installing Python libraries using the `setup.py` procedure is generally easier and gives better results even than using a package manager.

Troubleshooting

What to do when software won't compile or installations don't work

Installation is not always as easy as it could be. Most often, the culprit is one or more dependencies—supporting files that are used by the program. Because they deal with graphics, communications, and reading and writing to files, dependencies tend to be more platform-specific than a program designed just to do mathematical calculations. As a result, getting feature-rich graphical programs to install may require a bit of experimentation and Web searching. Here are a few approaches to troubleshooting an installation.

Look inside the `Makefile` If you look inside the `Makefile`, you may find that some lines particular to your operating system need either to be uncommented (remove the # sign), commented out (add a # at the beginning of the line), or otherwise edited. Often—but not always—guiding comments inside the file tell you where these modifications need to be made.

If you do try editing your `Makefile`, save the original version under a different name, so that you can go back to it if necessary. (The reason you have to save the original under a new name, rather than the edited file that you intend to experiment with, is because `Makefile` is a special name that the `make` program looks for to find its settings.) After editing the file, and before attempting to compile again with `make`, you will want to clear out any files generated by your previous compilation attempt. To do this, either try the `make clean` command, or manually delete all the files whose names end with the extension `.o`. You could also delete the entire source code folder for the program and then create a new one from the original archive file you downloaded. This ensures that previous attempts at making the program don't leave files that interfere with your troubleshooting attempts, although you would want to save the `Makefile` that you were editing in the original folder.

Warnings There are important differences between a warning message and an error. Errors are typically fatal to an installation. Many compilation procedures will issue a long stream of `Warning` lines, flagging places where the developer might have taken a few shortcuts or followed a procedure that is not strictly proper for your system. If your compiled program runs in the end, these warnings can usually be ignored without consequence; often they are a matter of style, completeness, and programming rigor. If your binary fails to run, however, the warnings may hold clues to libraries that failed to get linked up with your program. Pay special heed to any lines that talk about `lib` or `dylib` files not being found.

Permissions If you are lucky, then something will fail just because of incorrect permissions. This is relatively easily corrected using the `chmod u+x` command, or by invoking the `make` command with superuser privileges using `sudo make` or especially `sudo make install`.

Troubleshooting dependencies If your program is failing because some required files or libraries are not installing, try to get those to install separately first. This will narrow down the problem to the actual portion of the installation that is causing trouble. Conversely, if you installed the dependencies separately before trying the whole installation, the installer might not be finding your special library files. This can be a problem when using a combination of MacPorts for certain files, and manual installation for others. In these cases, you can try re-installing the dependencies as part of the process of installing your desired package. ImageMagick is an example of where this might happen, since it is dependent upon so many platform-specific graphics libraries.

Technical details of your platform If a binary is compiled for the wrong platform (e.g., Linux versus OS X), when you try to run it, you may get the error message `Cannot execute binary file`. A similar error may occur if a program has been compiled to use multiple processors. Sometimes a program will

be compiled for 64-bit operation—this has to do with how blocks of memory are handled during execution—and will fail with a cryptic error when you try to run it on an older system. In any of these cases, you will have to go back and recompile or download the appropriate executable, making sure that you have chosen 32-bit mode. If you are given the choice and you are not sure what your system supports, you can probably just use 32-bit mode without noticeable effect.

Search the Web If your installation continues to fail, there may be a bug or an error in the `Makefile`. These bugs are often not the programmer's fault, so don't complain too vehemently online. Just because a program works as written on one system does not mean that it will work on yours. Chances are that someone else has probably come across this problem before, so search the Web for the exact error you are receiving (with quotes around it so it is searched as a phrase) and see what solutions others have recommended.

Installing software from source code can be difficult, but it opens up a wide world of programs that you may find useful and which may not otherwise be available to you.

SUMMARY

You have learned:

- Three methods for installing software:
  - Using precompiled binaries for your system
  - Using package managers such as Synaptic, `apt-get`, Fink, and MacPorts
  - Compiling from source
- Compiling steps:
  - Uncompressing and unarchiving with `gunzip` and `tar -xf`
  - Generating the `Makefile` with `./configure`
  - Compiling with `make`
  - Installing with `sudo make install`

Moving forward

There are many programs specific to subdisciplines in biology that are available only as source code. Download, compile, and install tools that will be useful for your own work.