Chapter 6

Make Tutorial

`Make` is a versatile tool which can run commands to read files, process these files in some way (such as compiling and linking them), and write out the processed files.[1] This can be quite useful whenever edits are made since `Make` will automate all of the reprocessing. For example, in software development, `Make` is used to compile source code into executable programs or libraries. `Make` can also be used to:

1. Run scripts on data files to get summaries or plots.

2. Parse and combine text files and plots to create papers.

Imagine writing a book with all of the above and having all your plots and tables seamlessly update when you just say change a number in a part of your code.

`Make` is designed so as to only update files when they need to be. Specifically, it tracks the dependencies between files it creates and files used to create them. If one of the original files (e.g. a data file) is changed, then `Make` knows to recreate, or update, the files that depend upon this file (e.g. a plot). `Make` is called a "build tool"—it builds data files, plots, papers, programs, etc. There are now many build tools available, all of which are based on the same concepts as `Make`.

6.1 Setup

`Make` is founded upon `Unix`, so you should have basic familiarity with `Unix` commands before learning `Make`.

[1] We've abridged the wonderful tutorial at Software Carpentry for our own purposes, https://swcarpentry.github.io/make-novice/, in accordance with their creative commons license, https://creativecommons.org/licenses/by/4.0/. An in-depth manual for `Make` is http://www.gnu.org/software/make/manual/make.html.

Download Make

Opening your Terminal and typing `make -v` will give the version of `Make` on your computer (or an error if it's not installed). `Linux` systems will generally have `Make` already installed. On `Macs`, downloading `Xcode` from the App Store will automatically install `Make`. Go to https://github.com/swcarpentry/windows-installer for a Windows version of `Make`.

Download Tutorial Code

This tutorial illustrates how to use `Make` in the context of a toy project, https://swcarpentry.github.io/make-novice/files/make-lesson.zip. (If you would like to follow along by running the code you will need Python, at https://www.anaconda.com/download/#macos.)

Set the downloaded project folder "make-lesson" as your current working directory. Here are the contents of our project:

```
|- books
|  |- abyss.txt
|  |- isles.txt
|  |- last.txt
|  |- LICENSE_TEXTS.md
|  |- sierra.txt
|- plotcount.py
|- wordcount.py
|- zipf_test.py
```

There are text files of various books in the "books" folder, as well as three Python programs to analyze them.

For instance, the Python program "wordcount.py" counts the frequency each word in a text file.

```
$ python wordcount.py books/isles.txt isles.dat
$ python wordcount.py books/abyss.txt abyss.dat
```

This outputted the frequency of words in "isles.txt" and "abyss.txt" to new data files "isles.dat" and "abyss.dat." (We will not use "plotcount.py" which makes count plots.)

The Python program "zipf_test.py" uses these data files to test "Zipf's law."

```
$ python zipf_test.py isles.dat abyss.dat > results.txt
```

The output of this analysis is in "results.txt."

The purpose of `Make` is to automate all these `Unix` commands in a special way. That is, when any file is altered, we want all of the files that depend on it to update—without unnecessarily updating any other files.

6.2 Makefile

Similar to a shell script, we can create a `Makefile` to automate processing of the texts. Let us create a new file called "Makefile" (no extension) with the following content:

```
# Count words.
isles.dat : books/isles.txt
        python wordcount.py books/isles.txt isles.dat
```

(Warning: `Make` is very particular about whitespace in the `Makefile`. Certain parts of the code have to be indented with a tab and not spaces. Further, copying and pasting code from pdfs can lead to incorrect newline characters.)

Let us go through each line of this short `Makefile`:

- The "#" denotes a comment.
- "isles.dat" is a *target*, a file to be created, or built.
- "books/isles.txt" is a *dependency*, a file that is needed to build or update the target. Targets can have zero or more dependencies.
- The colon ":" separates targets from dependencies.
- Here, "python wordcount.py books/isles.txt isles.dat" is an *action*. This builds or updates the target using the dependencies. Targets can have zero or more actions. These actions form a recipe to build the target from its dependencies and can be considered like a shell script.
- Actions are indented using a single TAB character, not 8 spaces. (If the difference between spaces and a TAB character isn't obvious in your editor, try moving your cursor from one side of the TAB to the other. It should jump four or more spaces.)
- Together, the target, dependencies, and actions form a *rule*.

Our rule above describes how to build the *target* "isles.dat" using the *action* "python wordcount.py" and the *dependency* "books/isles.txt."

6.3 Run Make

To build the first target in a `Makefile`, you need simply type `make` into your Terminal.

```
$ make
python wordcount.py books/isles.txt isles.dat
```

When the `Makefile` has multiple targets, you can specify which one to build.

```
$ make isles.dat
```

If the target "isles.dat" has been updated more recently than the dependency "books/isles.txt," then `Make` will not update the target.

```
$ make
make: `isles.dat' is up to date.
$ touch books/isles.txt
$ make
python wordcount.py books/isles.txt isles.dat
```

If we ask `Make` to build a file that already exists and is not a target, then it will tell us "nothing to be done."

```
$ make wordcount.py
make: Nothing to be done for `wordcount.py'.
```

`Make` will throw an error trying to build a file that does not exist and is not a target.

```
$ make Not_a_File
make: *** No rule to make target `Not_a_File'.  Stop.
```

When `Make` is called, it automatically searches the working directory for a file titled "Makefile." However, if we call it something else we need to tell `Make` where to find it. This we can do using the `-f` flag. For example, if our `Makefile` is named "MyOtherMakefile," we can build it with:

```
$ make -f MyOtherMakefile
```

Phony Targets

Suppose we want to remove all our data files *.dat. We can introduce a new target "clean," and associated cleaning rule, to the end of our `Makefile`.

```
.PHONY : clean
clean :
        rm -f *.dat
```

Here, the target "clean" is ".PHONY" in the sense that it doesn't refer to a file. (That's what this first line says.) Otherwise, if there were a file called "clean," `Make` would not execute the rule because the file would be up to date as it has no dependencies. We can now execute the cleaning rule:

```
$ make clean
rm -f *.dat
```

Multiple Targets

Let's add another rule to the middle of the Makefile:

```
abyss.dat : books/abyss.txt
        python wordcount.py books/abyss.txt abyss.dat
```

This outputs the frequency of words in "abyss.txt" to a new "abyss.dat" file. We can now build the target "abyss.dat:"

```
$ make abyss.dat
python wordcount.py books/abyss.txt abyss.dat
```

As the `Makefile` gets larger, there may be a variety of data file targets that we would like to be able to build all at once. One way of doing this would be with a phony target "dats" which depends on all of the targets we want to build. Let's put the following at the beginning of the `Makefile` so that it is the default target.

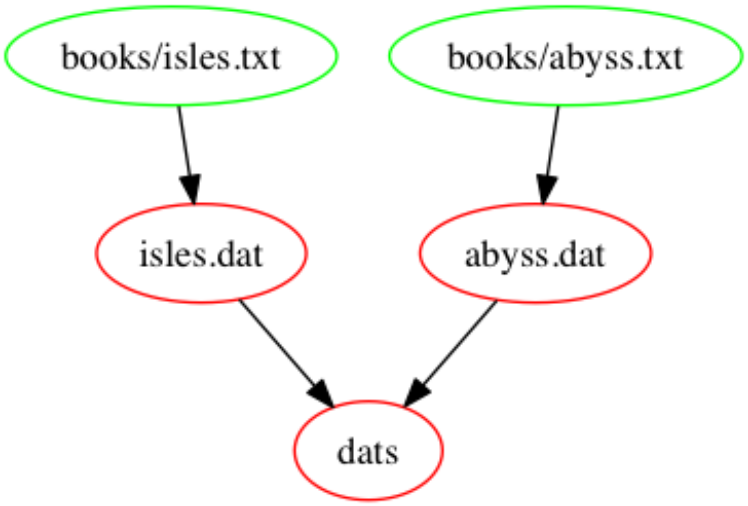
```
.PHONY : dats
dats : isles.dat abyss.dat
```

We can now build all the data files with either "make" or "make dats:"

```
$ make dats
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/abyss.txt abyss.dat
```

This is an example of a rule that has dependencies that are targets of other rules. When `Make` runs, it will check to see if the dependencies exist and, if not, will see if rules are available

that will create these. If such rules exist it will invoke these first, otherwise `Make` will raise an error.



Duplication

Our Makefile currently looks like:

```
# Count words.
.PHONY : dats
dats : isles.dat abyss.dat

isles.dat : books/isles.txt
        python wordcount.py books/isles.txt isles.dat

abyss.dat : books/abyss.txt
        python wordcount.py books/abyss.txt abyss.dat

.PHONY : clean
clean :
        rm -f *.dat
```

This has some duplication. For example, the names of text files and data files are repeated in many places throughout. `Makefiles` are a form of code and, in any code, repetition can lead to problems, for instance, we might rename a data file in one part of the Makefile but forget to rename it elsewhere.

In the following sections we'll reduce duplication with *automatic variables* and *pattern rules*.

Automatic Variables

Automatic variables are reserved expressions for targets and dependencies to be used within rules. (For all such variables: see https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html.) The most common ones are:

- `$@` is the target of the current rule.
- `$<` is the first dependency of the current rule.
- `$^` is all of the dependencies of the current rule.

This lets us eliminate some duplication in the actions of the data file rules.

```
isles.dat : books/isles.txt
        python wordcount.py $< $@
abyss.dat : books/abyss.txt
        python wordcount.py $< $@
```

Here, we replaced books with "$<," and the data files with "$@." These automatic variables get replaced when `Make` is run:

```
$ make isles.dat
python wordcount.py books/isles.txt isles.dat
```

Pattern Rules

Our `Makefile` still has repeated content. Specifically, the rules for each data file are identical except for the book name. We can replace these rules with a single *pattern rule*. This rule specifies how to build any data file from a text file in the books directory:

```
%.dat : books/%.txt
        python wordcount.py $< $*.dat
```

The `%` is a `Make` wildcard, it is placeholder for any data file name. The `$*` is an automatic variable that matches the "stem" of the target. (Here, $* will match %.) Now we can build the data file for any book, for instance:

```
$ make last.dat
python wordcount.py books/last.txt last.dat
```

Multiple Dependencies

Our data files are a product of not only the text files, but of the script "wordcount.py" that processes them. We should explicitly list the processing code as a dependency.

```
%.dat : books/%.txt wordcount.py
        python wordcount.py $< $*.dat
```

This way, when the `Python code` is changed, `Make` will rebuild the data files.

```
$ make isles.dat
make: `isles.dat' is up to date.
$ touch wordcount.py
$ make isles.dat
python wordcount.py books/isles.txt isles.dat
```

Build Results

We can add a build for our "results.txt" to the beginning of the `Makefile`.

```
# Generate summary table.
results.txt : zipf_test.py isles.dat abyss.dat last.dat
        python $< *.dat > $@
```

In the process of building "results.txt," `Make` will make sure that all of its data files dependencies are up to date. Here is the output,

```
$ make
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/abyss.txt abyss.dat
python wordcount.py books/last.txt last.dat
python zipf_test.py isles.dat abyss.dat last.dat > results.txt
```

6.4 Conclusion

Beyond automation, Makefiles act as a type of documentation. They explicitly record the inputs to and outputs from steps in our analysis and the dependencies between files. Thus, reducing the number of things we have to remember.

The following is our final `Makefile` code:

```
# Generate summary table.
results.txt : zipf_test.py isles.dat abyss.dat last.dat
        python $< *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat

%.dat : books/%.txt wordcount.py
        python wordcount.py $< $*.dat

.PHONY : clean
clean :
        rm -f *.dat
```

The following picture illustrates all of the dependencies: