Lesson 3: Automating Analyses with Make

How to deal with multiple files and dependencies in a project

Reference:
Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson
https://merely-useful.tech/py-rse/automate.html

Make and Build Management

- Make is a **build management tool** used to automate the process of compiling and linking software projects.
- Helps **manage and automate tasks in a workflow**, especially when dealing with multiple files and dependencies.
- Make uses a file called **'Makefile'** to define **how to compile and link** a program.

Key Concepts in Make

Build rule: A specification for a build manager that describes how some files depend on others and what to do if those files are out-of-date. A build rule has targets, prerequisites, and a recipe.

Build targets: The file(s) that a build rule will update if they are out-of-date compared to their dependencies.

Default target: The build target that is used when none is specified explicitly.

Prerequisite: Something that a build target depends on.

Recipe: The part of a build rule that describes how to update something that has fallen out-of-date.

How Make Works:

1. Every time the **operating system** creates, reads, or changes a file, it updates a **timestamp** on the file to show when the operation took place. Make can compare these timestamps to figure out whether files are newer or older than one another.
2. A user can describe which files depend on each other by writing **rules** in a **Makefile.**
   a. For example, one rule could say that `results/moby_dick.csv` depends on `data/moby_dick.txt`, while another could say that the plot `results/comparison.png` depends on all of the CSV files in the `results` directory.
3. Each rule also tells Make how to update an out-of-date file.
   a. For example, the rule for *Moby Dick* could tell Make to run `bin/countwords.py` if the result file is older than either the raw data file or the program.
4. When the user runs Make, the program checks all of the rules in the Makefile and runs the commands needed to update any that are out of date.
   a. If there are **transitive dependencies**—i.e., if A depends on B and B depends on C—then Make will trace them through and run all of the commands it needs to in the right order.

Installation: https://www.gnu.org/software/make/

Clone Repo: [instructor repo]

Testing it out

Run this command in the shell: `$ make`

- Make automatically looks for a file called `Makefile`, follows the rules it contains, and prints the commands that were executed
- Make indents a rule with spaces rather than tabs!

Using variables to avoid repetition

- Reduces cognitive load
- Easier to edit the Makefile if a filename changes

```
.PHONY : all clean

COUNT=bin/countwords.py
RUN_COUNT=python $(COUNT)

# Regenerate all results.
all : results/moby_dick.csv results/jane_eyre.csv

# Regenerate results for "Moby Dick"
results/moby_dick.csv : data/moby_dick.txt $(COUNT)
    $(RUN_COUNT) data/moby_dick.txt > results/moby_dick.csv

# Regenerate results for "Jane Eyre"
results/jane_eyre.csv : data/jane_eyre.txt $(COUNT)
    $(RUN_COUNT) data/jane_eyre.txt > results/jane_eyre.csv

# Remove all generated files.
clean :
    rm -f results/*.csv
```

Automatic Variables: Simplifying even further

```
.PHONY : all clean

COUNT=bin/countwords.py
RUN_COUNT=python $(COUNT)

# Regenerate all results.
all : results/moby_dick.csv results/jane_eyre.csv

# Regenerate results for "Moby Dick"
results/moby_dick.csv : data/moby_dick.txt $(COUNT)
    $(RUN_COUNT) data/moby_dick.txt > results/moby_dick.csv

# Regenerate results for "Jane Eyre"
results/jane_eyre.csv : data/jane_eyre.txt $(COUNT)
    $(RUN_COUNT) data/jane_eyre.txt > results/jane_eyre.csv

# Remove all generated files.
clean :
    rm -f results/*.csv
```

```
.PHONY: all clean

COUNT=bin/countwords.py
RUN_COUNT=python $(COUNT)

# Regenerate all results.
all : results/moby_dick.csv results/jane_eyre.csv \
  results/time_machine.csv

# Regenerate result for any book.
results/%.csv : data/%.txt $(COUNT)
    $(RUN_COUNT) $< > $@

# Remove all generated files.
clean :
    rm -f results/*.csv
```

Automatic Variables: Simplifying even further

- **%:** Also called **wildcard,** is a placeholder for pattern matching in rules
  - % cannot be used in recipes, which is why $< and $@ are needed.
- **$@:** Represents the **target of the rule**.
  - In this case, it will be replaced with results/%.csv where % matches the stem of the target file.
- **$<:** Represents the **first prerequisite**.
  - In this rule, it will be replaced with data/%.txt, corresponding to the source data file for each book.

```
.PHONY: all clean

COUNT=bin/countwords.py
RUN_COUNT=python $(COUNT)

# Regenerate all results.
all : results/moby_dick.csv results/jane_eyre.csv \
  results/time_machine.csv

# Regenerate result for any book.
results/%.csv : data/%.txt $(COUNT)
    $(RUN_COUNT) $< > $@

# Remove all generated files.
clean :
    rm -f results/*.csv
```

Automatic Variables

```
.PHONY: all clean

COUNT=bin/countwords.py
RUN_COUNT=python $(COUNT)

# Regenerate all results.
all : results/moby_dick.csv results/jane_eyre.csv \
  results/time_machine.csv

# Regenerate result for any book.
results/%.csv : data/%.txt $(COUNT)
    $(RUN_COUNT) $< > $@

# Remove all generated files.
clean :
    rm -f results/*.csv
```

```
python bin/countwords.py data/moby_dick.txt >
  results/moby_dick.csv
python bin/countwords.py data/jane_eyre.txt >
  results/jane_eyre.csv
python bin/countwords.py data/time_machine.txt >
  results/time_machine.csv
```

```
$ make  # Same as `make all` as "all" is the first target
```

Automating even further: Defining sets of files

But what if we add another book to the data folder?

```
DATA=$(wildcard data/*.txt)
```

Here, we are creating a **list** of all the text files in the data directory.

```
DATA: data/dracula.txt data/frankenstein.txt
  data/jane_eyre.txt data/moby_dick.txt
  data/sense_and_sensibility.txt data/sherlock_holmes
  data/time_machine.txt
```

Automating even further: Defining sets of files

Making a **phony target** called settings that uses the shell command `echo` allows us to print the names and values of our variables:

```
.PHONY: all clean settings

# ...everything else...

# Show variables' values.
settings :
    echo COUNT: $(COUNT)
    echo DATA: $(DATA)
```

```
$ make settings
```

```
echo COUNT: bin/countwords.py
COUNT: bin/countwords.py
echo DATA: data/dracula.txt data/frankenstein.txt
  data/jane_eyre.txt data/moby_dick.txt
  data/sense_and_sensibility.txt
  data/sherlock_holmes.txt data/time_machine.txt
DATA: data/dracula.txt data/frankenstein.txt
  data/jane_eyre.txt data/moby_dick.txt
  data/sense_and_sensibility.txt data/sherlock_holmes.txt
  data/time_machine.txt
```

Automating even further: Documenting Makefiles

- Now that we've defined each set of files, our code shrinks even more
- `$ make help` will display all the comments, to clarify targets and explain the workflow
- making comments as you go will keep the help function continually updated

```
.PHONY: all clean help settings

COUNT=bin/countwords.py
DATA=$(wildcard data/*.txt)
RESULTS=$(patsubst data/%.txt,results/%.csv,$(DATA))

## all : regenerate all results.
all : $(RESULTS)

## results/%.csv : regenerate result for any book.
results/%.csv : data/%.txt $(COUNT)
    python $(COUNT) $< > $@

## clean : remove all generated files.
clean :
    rm -f results/*.csv

## settings : show variables' values.
settings :
    @echo COUNT: $(COUNT)
    @echo DATA: $(DATA)
    @echo RESULTS: $(RESULTS)

## help : show this message.
help :
    @grep '^##' ./Makefile
```

```
$ make all
```