KATHOLIEKE UNIVERSITEIT LEUVEN

universiteit hasselt
KNOWLEDGE IN ACTION

VLAAMS SUPERCOMPUTER CENTRUM

Vlaanderen
is supercomputing

Make - introduction

Mag Selwa, Ehsan Moravveji

ICTS, Leuven

SCAN ME

Credit: Software Carpentry Foundation

24 November 2020

Build automation

- **Build automation** is the process of automating the creation of a software build and the associated processes including: compiling computer source code into binary code, packaging binary code, and running automated tests.
- Today, there are two general categories of tools:
  - Build automation utility (like Make, Rake, Cake, MS build, Ant, Gradle etc.)
    Primary purpose is to generate build artifacts through activities like compiling and linking source code.
  - Build automation servers
    These are general web based tools that execute build automation utilities on a scheduled or triggered basis; a continuous integration server is a type of build automation server.
- There are now a number of dependency-tracking build utilities, but Make is one of the most widespread, primarily due to its inclusion in Unix, starting with the PWB/UNIX 1.0, which featured a variety of tools targeting software development tasks.

Source: Wikipedia

Build automation

- Depending on the Makefile level of automation the following classification is possible:
  - Make-based tools (GNU make, make, mk, nmake, MPW make),
  - Non-Make-based tools (Apache Ant, Apache Moven, A-A-P, Bazel, Boot, Ninja, Scons, ….).
- Another group are build script generation tools (configure, Cmake, imake, OpenMake, Premake, qmake, GNU build system – autotools - a collection of tools for portable builds. These in particular include Autoconf and Automake, cross-platform tools that together generate appropriate localized makefiles).
- Other means of automation include
  - Continuous integration tools,
  - Configuration management tools,
  - Meta-build tools or package managers.

Source: Wikipedia

What is Make?

- **Make** was developed by Stuart Feldman in 1977 as a Bell Labs summer intern and remains in widespread use today.
- In software development, Make is a build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix and Unix-like operating systems.
- Besides building programs, Make can be used to manage any project where some files must be updated automatically from others whenever the others change.

What is Make?

- Make is a tool which can run commands to read files, process these files in some way, and write out the processed files. For example, in software development, Make is used to compile source code into executable programs or libraries
- Make can also be used to:
  - run analysis scripts on raw data files to get data files that summarize the raw data;
  - run visualization scripts on data files to produce plots;
  - parse and combine text files and plots to create papers.

Make and programming?

- Make is called a build tool - it builds data files, plots, papers, programs or libraries. It can also update existing files if desired.
- Make tracks the dependencies between the files it creates and the files used to create these. 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).
- There are now many build tools available, all of which are based on the same concepts as Make.

Make introduction

- **Questions**
  - How can I make my results easier to reproduce?
- **Objectives**
  - Explain what Make is for.
  - Explain why Make differs from shell scripts.
  - Name other popular build tools.

Make introduction

- Let's imagine that we're interested in testing Zipf's Law in some of our favorite books.
- **Zipf's Law:** The most frequently-occurring word occurs approximately twice as often as the second most frequent word.
- We've compiled our raw data i.e. the books we want to analyze and have prepared several Python scripts that together make up our analysis pipeline.
- Suppose that our directory has the Python scripts and data files we will be working with:

```
|- books
| |- abyss.txt
| |- isles.txt
| |- last.txt
| |- LICENSE_TEXTS.md
| |- sierra.txt
|- plotcounts.py
|- countwords.py
|- testzipf.py
```

- The first step is to count the frequency of each word in a book.
- Let's imagine that we're interested in testing Zipf's Law in some of our favorite books.

Hands-on 1

- Download make-lesson.zip from https://swcarpentry.github.io/make-novice/files/make-lesson.zip
- Move make-lesson.zip into a directory which you can access via your bash shell.
- Open a Bash shell window.
- Navigate to the directory where you downloaded the file.
- Unpack make-lesson.zip (`$ unzip make-lesson.zip`).
- Change into make-lesson directory (`$ cd make-lesson`).
- Let's take quick look at one of the books using the command head books/isles.txt (`$ head books/isles.txt`).
- Load Python/2.7 and Matplotlib modules for the whole exercises from now on.
- The first step is to count the frequency of each word in a book.

```
$ python countwords.py books/isles.txt isles.dat
```

- Take a quick peek at the result (`$ head -5 isles.dat`)
- We can see that the file consists of one row per word. Each row shows the word itself, the number of occurrences of that word, and the number of occurrences as a percentage of the total number of words in the text file.
- We can do the same thing for a different book (abyss)

```
$ python countwords.py books/abyss.txt abyss.dat
$ head -5 abyss.dat
```

Hands-on 1

- Let's visualize the results. The script plotcounts.py reads in a data file and plots the 10 most frequently occurring words as a text-based bar plot:

```
$ python plotcounts.py isles.dat ascii
```

- plotcounts.py can also show the plot graphically or create the plot as an image file (e.g. PNG image):

```
$ python plotcounts.py isles.dat show
$ python plotcounts.py isles.dat isles.png
```

- Finally, let's test Zipf's law for these books:

```
$ python testzipf.py abyss.dat isles.dat
```

- Conclusion: we are not too far from Zipf's law.

Make introduction

- The generic workflow is the following:
  - Read a data file.
  - Perform an analysis on this data file.
  - Write the analysis results to a new file.
  - Plot a graph of the analysis results.
  - Save the graph as an image, so we can put it in a paper.
  - Make a summary table of the analyses.
- Running `countwords.py` and `plotcounts.py` at the shell prompt, as we have been doing, is fine for one or two files. If, however, we had 5 or 10 or 20 text files, or if the number of steps in the pipeline were to expand, this could turn into a lot of work. Plus, no one wants to sit and wait for a command to finish, even just for 30 seconds.
- The most common solution to the tedium of data processing is to write a shell script that runs the whole pipeline from start to finish.

We start

- Using your text editor of choice (e.g. nano), add the following to a new file named `pipeline.sh`.

```
# USAGE: bash pipeline.sh
# to produce plots for isles and abyss
# and the summary table for the Zipf's law tests
python countwords.py books/isles.txt isles.dat
python countwords.py books/abyss.txt abyss.dat
python plotcounts.py isles.dat isles.png
python plotcounts.py abyss.dat abyss.png
# Generate summary table
python testzipf.py abyss.dat isles.dat > results.txt
```

- Run the script and check that the output is the same as before:

```
$ bash pipeline.sh
$ cat results.txt
```

We start – shell script?

- This **shell script** solves several problems in computational reproducibility:
  - It explicitly documents our pipeline, making communication with colleagues (and our future selves) more efficient.
  - It allows us to type a single command, bash pipeline.sh, to reproduce the full analysis.
  - It prevents us from repeating typos or mistakes. You might not get it right the first time, but once you fix something it'll stay fixed.
- Despite these benefits it has a few **shortcomings**.
- Let's adjust the width of the bars in our plot produced by plotcounts.py. Edit plotcounts.py so that the bars are 0.8 units wide instead of 1 unit. (Hint: replace `width = 1.0` with `width = 0.8` in the function `plot_word_counts`).
- Now we want to recreate our figures. We could just `bash pipeline.sh` again. That would work, but it could also be a big pain if counting words takes more than a few seconds. The word counting routine hasn't changed; we shouldn't need to recreate those files.

We start – shell script?

- Alternatively, we could manually rerun the plotting for each word-count file. (Experienced shell scripters can make this easier on themselves using a for-loop.)

```
for book in abyss isles; do
  python plotcounts.py $book.dat $book.png
done
```

- With this approach, however, we don't get many of the benefits of having a shell script in the first place.

We start – shell script?

- Another popular option is to comment out a subset of the lines in pipeline.sh:

```
# USAGE: bash pipeline.sh
# to produce plots for isles and abyss
# and the summary table for the Zipf's law tests
#python countwords.py books/isles.txt isles.dat
#python countwords.py books/abyss.txt abyss.dat
python plotcounts.py isles.dat isles.png
python plotcounts.py abyss.dat abyss.png
# Generate summary table
python testzipf.py abyss.dat isles.dat > results.txt
```

- Then, we would run our modified shell script using `bash pipeline.sh`.

We start – shell script?

- But commenting out these lines, and subsequently uncommenting them, can be a hassle and source of errors in complicated pipelines.
- What we really want is an **executable description** of our pipeline that allows software to do the tricky part for us: figuring out what steps need to be rerun.
- Make can execute the commands needed to run our analysis and plot our results. Like shell scripts it allows us to execute complex sequences of commands via a single shell command. Unlike shell scripts it explicitly records the dependencies between files - what files are needed to create what other files - and so can determine when to recreate our data files or image files, if our text files change.

We start

- Make can be used for any commands that follow the general pattern of processing files to create new files, for example:
  - Run analysis scripts on raw data files to get data files that summarize the raw data (e.g. creating files with word counts from book text).
  - Run visualization scripts on data files to produce plots (e.g. creating images of word counts).
  - Parse and combine text files and plots to create papers.
  - Compile source code into executable programs or libraries.
- There are now many build tools available, for example Apache ANT, doit, and nmake for Windows. There are also build tools that build scripts for use with these build tools and others e.g. GNU Autoconf and CMake. Which is best for you depends on your requirements, intended usage, and operating system. However, they all share the same fundamental concepts as Make.

We start

Why Use Make if it is Almost 40 Years Old?

- Today, researchers working with legacy codes in C or FORTRAN, which are very common in high-performance computing, will, very likely encounter Make.
- Researchers are also finding Make of use in implementing reproducible research workflows, automating data analysis and visualization (using Python or R) and combining tables and plots with text to produce reports and papers for publication.
- Make's fundamental concepts are common across build tools.
- GNU Make is a free, fast, well-documented, and very popular Make implementation. From now on, we will focus on it, and when we say Make, we mean GNU Make.
- Make is the most-widely know build system (which helps solving issues by asking around or Googling).
- **Key Points**: Make allows us to specify what depends on what and how to update things that are out of date.

We start - Makefile

- **Questions**
  - How do I write a simple Makefile?
- **Objectives**
  - Recognize the key parts of a Makefile, rules, targets, dependencies and actions.
  - Write a simple Makefile.
  - Run Make from the shell.
  - Explain when and why to mark targets as .PHONY.
  - Explain constraints on dependencies.

We start - Makefile

- Create a file, called `Makefile`, with the following content:

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

- This is a build file, which for Make is called a Makefile - a file executed by Make. Note how it resembles one of the lines from our shell script.

Makefile components

Components:

- `#` denotes a **comment**. Any text from # to the end of the line is ignored by Make.
- `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.
- A colon, `:`, **separates** targets from dependencies.
- `python countwords.py books/isles.txt isles.dat` is an **action**, a command to run to build or update 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 to be a shell script.
- Actions are **indented** using a single `TAB` character, not 8 spaces. This is a legacy of Make's 1970's origins. 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**.

Makefile components

- The rule above describes how to build the **target** isles.dat using the **action** python countwords.py and the **dependency** books/isles.txt.
- Information that was implicit in our shell script - that we are generating a file called isles.dat and that creating this file requires books/isles.txt - is now made explicit by Make's syntax.
- Before we start: first ensure we start from scratch and delete the .dat and .png files we created earlier:

```
$ rm *.dat *.png
```

Makefile components

- How to use:
  By default, Make looks for a Makefile, called Makefile, and we can run Make as follows:
  ```
  $ make
  ```
- By default, Make prints out the actions it executes:
  ```
  python countwords.py books/isles.txt isles.dat
  ```
- Common problem:
  If we see,
  ```
  Makefile:3: *** missing separator. Stop.
  ```
  then we have used a space instead of a TAB characters to indent one of our actions.
- Check if the result is as exected:
  ```
  $ head -5 isles.dat
  ```
  The first 5 lines of isles.dat should look exactly like before.

Makefile filename

- **Makefiles Do Not Have to be Called Makefile**
  We don't have to call our Makefile Makefile. However, if we call it something else we need to tell Make where to find it. This we can do using -f flag. For example, if our Makefile is named MyOtherMakefile:
  ```
  $ make -f MyOtherMakefile
  ```
- Sometimes, the suffix `.mk` will be used to identify Makefiles that are not called Makefile e.g. install.mk, common.mk etc.
- **Rerun**:
  When we re-run our Makefile, Make now informs us that:
  ```
  make: `isles.dat' is up to date.
  ```
  This is because our target, isles.dat, has now been created, and Make will not create it again.

We start - rebuild

Rerun for changes:

- Let's pretend to update one of the text files. Rather than opening the file in an editor, we can use the shell touch command to update its timestamp (which would happen if we did edit the file):
  ```
  $ touch books/isles.txt
  ```
- To make sure - compare the timestamps of books/isles.txt and isles.dat,
  ```
  $ ls -l books/isles.txt isles.dat
  ```
  we see that isles.dat, the target, is now older thanbooks/isles.txt, its dependency:

```
-rw-r--r-- 1 vsc30468 vsc30468 323972 Dec 1 10:35 books/isles.txt
-rw-r--r-- 1 vsc30468 vsc30468 182273 Dec 1 09:58 isles.dat
```

- If we run Make again, `$ make` then it recreates isles.dat:

```
python countwords.py books/isles.txt isles.dat
```

When it is asked to build a target, Make checks the '**last modification time**' of both the target and its dependencies. If any dependency has been updated since the target, then the actions are re-run to update the target. Using this approach, Make knows to only rebuild the files that, either directly or indirectly, depend on the file that changed. This is called an **incremental build**.

Makefiles as Documentation

- By explicitly recording the inputs to and outputs from steps in our analysis and the dependencies between files, Makefiles act as a type of documentation, reducing the number of things we have to remember.

- Let's add another rule to the end of Makefile:

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

- If we run Make,

```
$ make
```

- then we get:

```
make: `isles.dat' is up to date.
```

Makefiles: default target

- Nothing happens because Make attempts to build the **first target it finds** in the Makefile, the **default target**, which is isles.dat which is already up-to-date.

- To change the target we need to explicitly tell Make we want to build abyss.dat:
```
$ make abyss.dat
```

- We will get:
```
python countwords.py books/abyss.txt abyss.dat
```

“Up to Date” Versus “Nothing to be Done”

- If we ask Make to build a file that **already exists and is up to date**, then Make informs us that:

```
make: `isles.dat' is up to date.
```

- If we ask Make to build a file that **exists but for which there is no rule** in our Makefile, then we get message like:

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

- **up to date** means that the Makefile has a rule with one or more actions whose target is the name of a file (or directory) and the file is up to date.

- **Nothing to be done** means that the file exists but either :
  - the Makefile has no rule for it, or
  - the Makefile has a rule for it, but that rule has no actions.

Makefile: clean

- We may want to **remove all our data files** so we can explicitly recreate them all. We can introduce a **new target, and associated rule**, to do this. We will call it **clean**, as this is a common name for rules that delete auto-generated files, like our .dat files:

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

- This is an example of a rule that has **no dependencies**. clean has no dependencies on any .dat file as it makes no sense to create these just to remove them. We just want to remove the data files whether or not they exist. If we run Make and specify this target,

```
$ make clean
```

then we get:

```
rm -f *.dat
```

Clean – common problems

- There is no actual thing built called clean. Rather, it is a short-hand that we can use to execute a useful sequence of actions. Such targets, though very useful, can lead to problems. For example, let us recreate our data files, **create a directory called clean**, then run Make:

```
$ make isles.dat abyss.dat
$ mkdir clean
$ make clean
```

  We get:

```
make: `clean' is up to date.
```

- Make finds a file (or directory) called clean and, as its clean rule has no dependencies, assumes that clean has been built and is up-to-date and so does not execute the rule's actions.

Phony target

- As we are using clean as a short-hand, we need to tell Make to always execute this rule if we run make clean, by telling Make that this is a **phony target**, that it **does not build anything**. This we do by marking the target as `.PHONY`:

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

- If we run Make,

```
$ make clean
```

then we get:

```
rm -f *.dat
```

- In terms of Make, a phony target is simply a target that **is always out-of-date**, so whenever you ask `make <phony_target>`, it will run, independent from the state of the file system. Some common make targets that are often phony are: all, install, clean, distclean, info, check.

Phony target

- We can add a similar command to create all the data files. We can put this at the top of our Makefile so that it is the default target, which is executed by default if no target is given to the make command:

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

Makefile: dependencies

- The order of rebuilding dependencies is arbitrary. You should not assume that they will be built in the order in which they are listed.

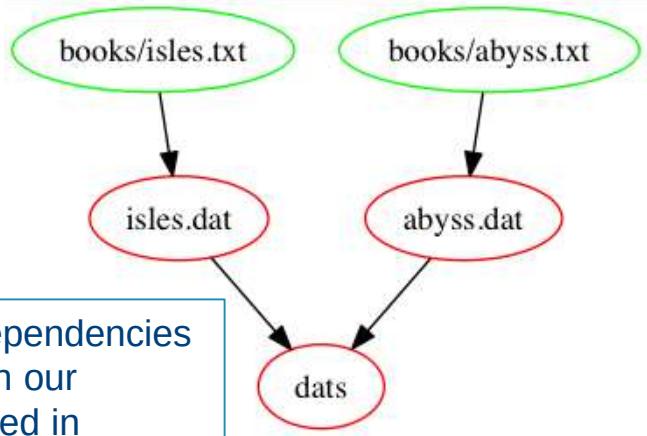
- Dependencies must form a directed acyclic graph. A target cannot depend on a dependency which itself, or one of its dependencies, depends on that target.

- This rule is also an example of a rule that has no actions. It is used purely to trigger the build of its dependencies, if needed.
  If we run,
  ```
  $ make dats
  ```
  then Make creates the data files:
  ```
  python countwords.py books/isles.txt isles.dat
  python countwords.py books/abyss.txt abyss.dat
  ```

Makefile: dependencies

- If we run dats again, then Make will see that the dependencies (isles.dat and abyss.dat) are already up to date. Given the target dats has no actions, there is nothing to be done:

```
$ make dats
make: Nothing to be done for `dats'.
```



Graph of the dependencies embodied within our Makefile, involved in building the dats target.

- Suppose the Makefile looks like this:

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

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

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

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

Makefile: components

Key Points

- Use `#` for comments in Makefiles.
- Write rules as `target: dependencies`.
- Specify update actions in a tab-indented block under the rule.
- Use `.PHONY` to mark targets that don't correspond to files.

Hands-on 2

Write Two New Rules

1. Write a new rule for `last.dat`, created from books/last.txt.
2. Update the `dats` rule with this target.
3. Write a new rule for `results.txt`, which creates the summary table. The rule needs to:
   - Depend upon each of the three .dat files.
   - Invoke the action `python testzipf.py abyss.dat isles.dat last.dat > results.txt`.
4. Put this rule at the top of the Makefile so that it is the default target.
5. Update clean so that it removes results.txt.

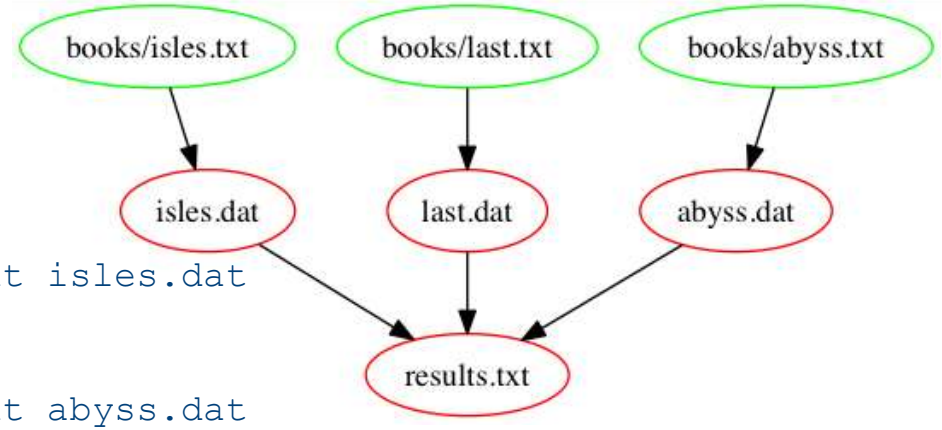
Hands-on 2

The starting Makefile:

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

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

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

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

Hands-on 2 (after the changes)

```
# Generate summary table.
results.txt : isles.dat abyss.dat last.dat
	python testzipf.py abyss.dat isles.dat last.dat > results.txt

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

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

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

last.dat : books/last.txt
	python countwords.py books/last.txt last.dat

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



Makefile: automation

- **Questions**
  - How can I abbreviate the rules in my Makefiles?
- **Objectives**
  - Use Make automatic variables to remove duplication in a Makefile.
  - Explain why shell wildcards in dependencies can cause problems.

Makefile: repetition?

- Our Makefile has a lot of **duplication**. For example, the names of text files and data files are repeated in many places throughout the Makefile.

- Makefiles are a form of code and, in any code, **repeated code** can lead to problems e.g. we rename a data file in one part of the Makefile but **forget to rename** it elsewhere.

- D.R.Y. (**Don't Repeat Yourself**)
- In many programming languages, the bulk of the language features are there to allow the programmer to describe long-winded computational routines as short, expressive, beautiful code. Features in Python or R or Java, such as user-defined variables and functions are useful in part because they mean we don't have to write out (or think about) all of the details over and over again. This good habit of writing things out only once is known as the "Don't Repeat Yourself" principle or D.R.Y.

Makefile: automation

- Let us remove some of the repetition from our Makefile.
- In `results.txt` rule we duplicate the data file names and the name of the results file name:

```
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py abyss.dat isles.dat last.dat > results.txt
```

- Looking at the results file name first, we can replace it in the action with `$@`:

```
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py abyss.dat isles.dat last.dat > $@
```

- `$@` is a Make **automatic variable** which means '**the target of the current rule**'. When Make is run it will replace this variable with the target name.

Makefile: automation

- We can replace the dependencies in the action with `$^`:

```
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py $^ > $@
```

- **`$^`** is another automatic variable which means '**all the dependencies of the current rule**'. Again, when Make is run it will replace this variable with the dependencies.
- If we update our text files and re-run our rule:

```
$ touch books/*.txt
$ make results.txt
```

- We get:

```
python countwords.py books/isles.txt isles.dat
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/last.txt last.dat
python testzipf.py isles.dat abyss.dat last.dat > results.txt
```

Makefile: automation

- What will happen if you now execute:

```
$ touch *.dat
$ make results.txt
```

1. nothing
2. all files recreated
3. only .dat files recreated
4. only results.txt recreated

```
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py $^ > $@
isles.dat : books/isles.txt
    python countwords.py books/isles.txt isles.dat
abyss.dat : books/abyss.txt
    python countwords.py books/abyss.txt abyss.dat
last.dat : books/last.txt
    python countwords.py books/last.txt last.dat
```

Solution

- 4. Only results.txt recreated.
- The rules for *.dat are not executed because their corresponding .txt files haven't been modified.
- If you run:

```
$ touch books/*.txt
$ make results.txt
```

- you will find that the .dat files as well as results.txt are recreated.

Makefile: automation

- As we saw, **$^** means '**all the dependencies of the current rule**'. This works well for results.txt as its action treats all the dependencies the same - as the input for the `testzipf.py` script.
- However, for some rules, we may want to **treat the first dependency differently**. For example, our rules for `.dat` use their first (and only) dependency specifically as the input file to `countwords.py`. If we add additional dependencies (as we will soon do) then we don't want these being passed as input files to `countwords.py` as it expects only one input file to be named when it is invoked.
- Make provides an automatic variable for this, **$<** which means '**the first dependency of the current rule**'.

Makefile: automation

Key Points

- Use **$@** to refer to the target of the current rule.
- Use **$^** to refer to all dependencies of the current rule.
- Use **$<** to refer to the first dependency of the current rule.

Hands-on 3

- Rewrite each `.dat` rule to use the automatic variables `$@` (‘the target of the current rule’) and `$<` (‘the first dependency of the current rule’).
- Makefile before the challenge.

Hands-on 3 (original)

```
# Generate summary table.
results.txt : *.dat
	python testzipf.py $^ > $@

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

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

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

last.dat : books/last.txt
	python countwords.py books/last.txt last.dat

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

Hands-on 3 (after the changes)

```
# Generate summary table.
results.txt : isles.dat abyss.dat last.dat
	python testzipf.py $^ > $@

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

isles.dat : books/isles.txt
	python countwords.py $< $@

abyss.dat : books/abyss.txt
	python countwords.py $< $@

last.dat : books/last.txt
	python countwords.py $< $@

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

Makefile: data and the code

- **Questions**
  - How can I write a Makefile to update things when my scripts have changed rather than my input files?
- **Objectives**
  - Output files are a product not only of input files but of the scripts or code that created the output files.
  - Recognize and avoid false dependencies.

Makefile: data and the code

```
# Generate summary table.
results.txt : isles.dat abyss.dat last.dat
    python testzipf.py $^ > $@
# Count words.
.PHONY : dats
    dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
    python countwords.py $< $@
abyss.dat : books/abyss.txt
    python countwords.py $< $@
last.dat : books/last.txt
    python countwords.py $< $@
.PHONY : clean
clean :
    rm -f *.dat
    rm -f results.txt
```

Makefile: data and the code

- Our data files are a product not only of our text files but the script, `countwords.py`, that processes the text files and creates the data files. A change to `countwords.py` (e.g. to add a new column of summary data or remove an existing one) results in changes to the `.dat` files it outputs.
- Let's pretend to edit `countwords.py`, using touch, and re-run Make:

```
$ make dats
$ touch countwords.py
$ make dats
```

- Nothing happens! Though we've **updated** `countwords.py` our data files are not updated because our rules for creating `.dat` files **don't record any dependencies** on `countwords.py`.

Makefile: data and the code

- We need to add `countwords.py` as a dependency of each of our data files also:

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

- If we pretend to edit `countwords.py` and re-run Make

```
$ touch countwords.py
$ make dats
```

- Then we get:

```
python countwords.py books/isles.txt isles.dat
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/last.txt last.dat
```

Make: dry run

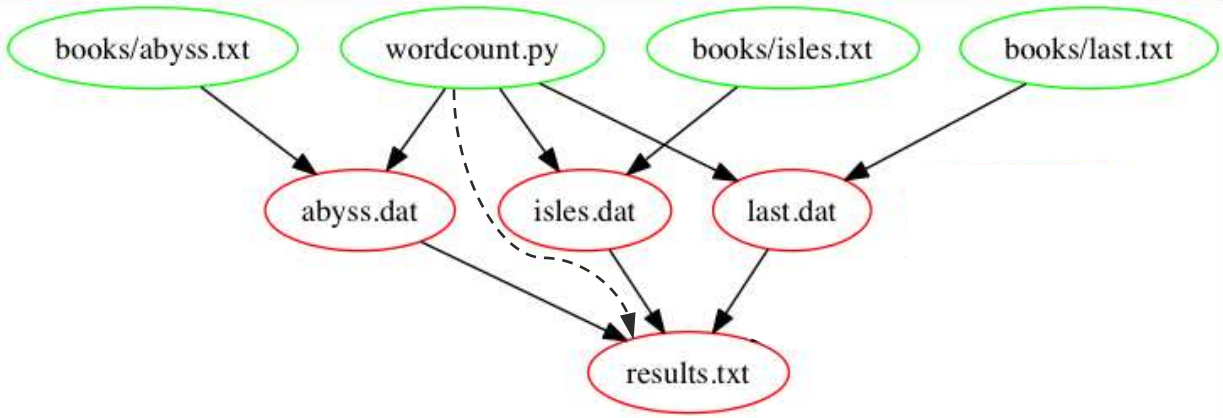
- make can show the commands it will execute without actually running them if we pass the **`-n`** flag:

  ```
  $ touch countwords.py $ make -n dats
  ```

- This gives the same output to the screen as without the `-n` flag, but the commands are not actually run. Using this ‘dry-run’ mode is a good way to check that you have set up your Makefile properly before actually running the commands in it.

Make: false dependency

- `*.txt` files are input files and have no dependencies. To make these depend on `countwords.py` would introduce a **false dependency**.
- Intuitively, we should also add `countwords.py` as dependency for `results.txt`, as the final table should be rebuilt as we remake the `.dat` files. However, it turns out we don't have to!



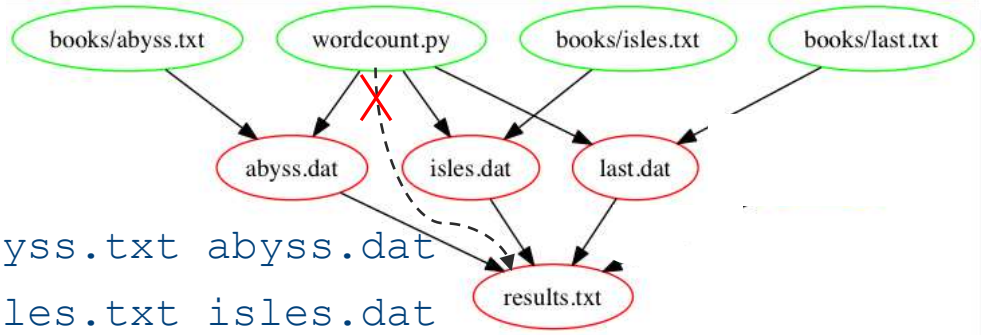
Make: updating pipeline

- Let's see what happens to results.txt when we update countwords.py:

```
$ touch countwords.py
$ make results.txt
```

we get:

```
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/isles.txt isles.dat
python countwords.py books/last.txt last.dat
python testzipf.py abyss.dat isles.dat last.dat > results.txt
```

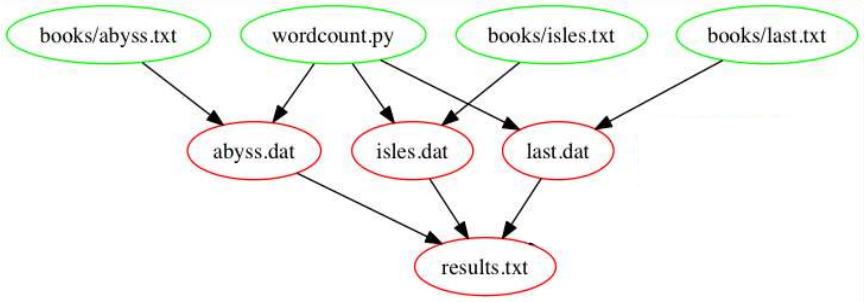


- The whole pipeline is triggered, even the creation of the `results.txt` file! To understand this, note that according to the dependency figure, `results.txt` depends on the `.dat` files. The update of `countwords.py` triggers an update of the `*.dat` files. Thus, make sees that the **dependencies** (the `.dat` files) are **newer than the target file** (`results.txt`) and thus it recreates `results.txt`.
- This is an example of the power of make: **updating a subset of the files** in the pipeline triggers **rerunning** the appropriate downstream steps.

Make: updating pipeline

- Updating One Input File: what will happen if we now execute:

```
$ touch books/last.txt
$ make results.txt
```



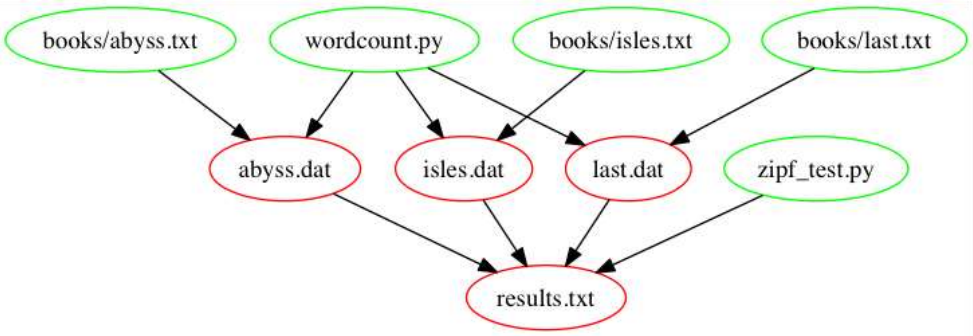
1. only `last.dat` is recreated
2. all `.dat` files are recreated
3. only `last.dat` and `results.txt` are recreated
4. all `.dat` and `results.txt` are recreated

- **Solution**

  3. only `last.dat` and `results.txt` are recreated.

Make: updating pipeline

- `testzipf.py` as a Dependency of `results.txt`. What would happen if you added `testzipf.py` as dependency of `results.txt`, and why?



- **Solution**

  If you change the rule for the results.txt file like this:

```
results.txt : isles.dat abyss.dat last.dat testzipf.py
    python testzipf.py $^ > $@
```

  `testzipf.py` becomes a part of `$^`, thus the command becomes

```
python testzipf.py abyss.dat isles.dat last.dat testzipf.py
> results.txt
```

Make: updating pipeline

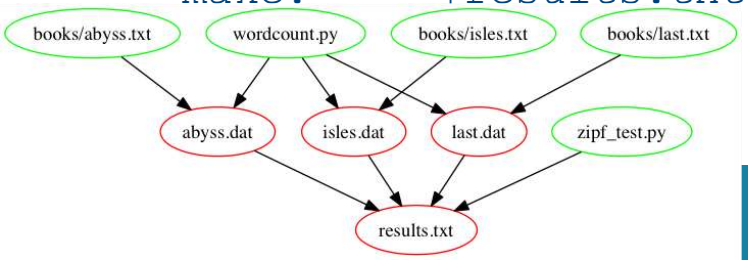
```
python testzipf.py abyss.dat isles.dat last.dat testzipf.py > results.txt
```

- This results in an error from `testzipf.py` as it tries to parse the script as if it were a `.dat` file. Try this by running:

  ```
  $ make results.txt
  ```

  You'll get

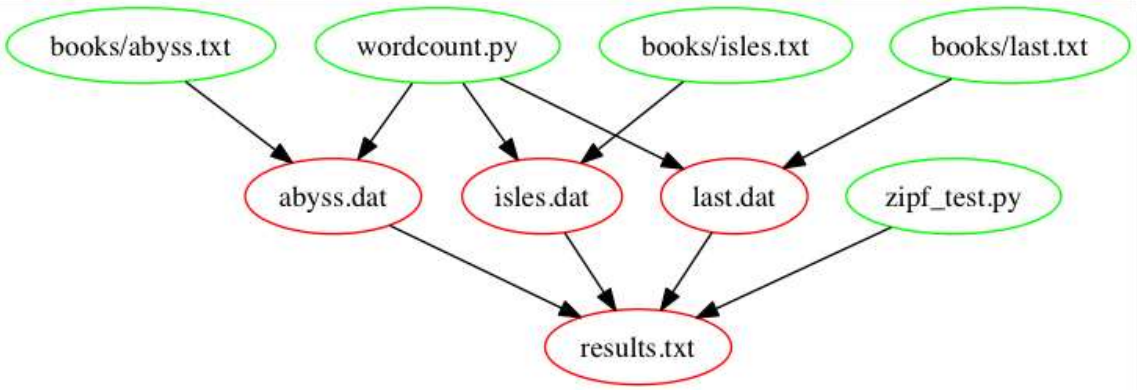
  ```
  python testzipf.py abyss.dat isles.dat last.dat testzipf.py >
  results.txt
  Traceback (most recent call last):
    File "testzipf.py", line 19, in <module>
      counts = load_word_counts(input_file)
    File "path/to/testzipf.py", line 39, in load_word_counts
     counts.append((fields[0], int(fields[1]), float(fields[2])))
  IndexError: list index out of range
  make: *** [results.txt] Error 1
  ```



Make: updating pipeline

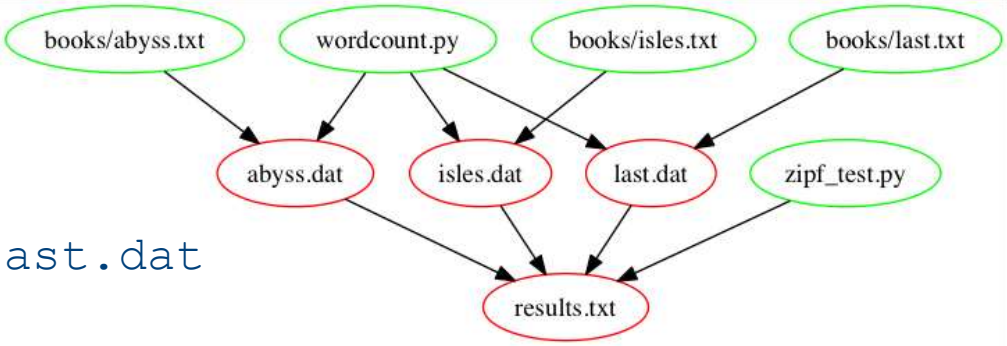
- Way around: we still have to add the `testzipf.py` script as dependency to `results.txt`. BUT we cannot use `$^` in the rule.
- We can however move `testzipf.py` to be the first dependency and then use `$<` to refer to it. In order to refer to the `.dat` files, we can just use `*.dat` for now (we will cover a better solution later on).

```
results.txt : testzipf.py isles.dat abyss.dat last.dat
    python $< *.dat > $@
```



Make: updating pipeline

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

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

isles.dat : books/isles.txt countwords.py
	python countwords.py $< $@
abyss.dat : books/abyss.txt countwords.py
	python countwords.py $< $@
last.dat : books/last.txt countwords.py
	python countwords.py $< $@
.PHONY : clean
clean :
	rm -f *.dat
	rm -f results.txt
```



Makefile: data and the code

Key Points

- Make results depend on processing **scripts** as well as **data** files.
- Dependencies are **transitive**: if A depends on B and B depends on C, a change to C will indirectly trigger an update to A.

Makefile: pattern rules

- **Questions**
  - How can I define rules to operate on similar files?
- **Objectives**
  - Write Make pattern rules.

Makefile: pattern rules

- Our Makefile still has **repeated content**. The rules for each `.dat` file are identical apart from the text and data file names.
- We can replace these rules with a **single pattern rule** which can be used to build any `.dat` file from a `.txt` file in `books/`:

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

- `%` is a Make wildcard. `$*` is a special variable which gets replaced by the stem with which the rule matched.
- This rule can be interpreted as: “In order to build a file named [something].dat (the target) find a file named books/[that same something].txt (the dependency) and run countwords.py [the dependency] [the target].”
- If we re-run Make,

```
$ make clean
$ make dats
```

then we get:

```
python countwords.py books/isles.txt isles.dat
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/last.txt last.dat
```

Makefile: pattern rules

- Note that we can still use Make to build individual `.dat` targets as before, and that our **new rule will work no matter what stem** is being matched.
  `$ make sierra.dat`
  which gives the output below:
  `python countwords.py books/sierra.txt sierra.dat`

Makefile: make wildcards

- The Make `%` wildcard can only be used in a **target and in its dependencies**. It cannot be used in actions.
- In **actions**, you may however use **`$*`**, which will be replaced by the **stem** with which the rule matched.
- Our example Makefile is now much shorter and cleaner:

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

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

%.dat : books/%.txt countwords.py
	python countwords.py $< $*.dat
.PHONY : clean
clean :
	rm -f *.dat
	rm -f results.txt
```

Makefile: make wildcards

- We introduced pattern rules, and used the `$*` variable in the `dat` rule in order to explain how to use it. Arguably, a neater solution would have been to use `$@` to refer to the target of the current rule, but then we wouldn't have learned about `$*`.

```
%.dat : books/%.txt countwords.py
    python countwords.py $< $@

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

Makefile: make wildcards

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

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

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

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

Makefile: pattern rules

Key Points

- Use the wildcard **%** as a placeholder in targets and dependencies.
- Use the special variable **$*** to refer to matching sets of files in actions.

Makefile: variables

- **Questions**
  - How can I eliminate redundancy in my Makefiles?
- **Objectives**
  - Use variables in a Makefile.
  - Explain the benefits of decoupling configuration from computation

Makefile: variables

- Despite our efforts, our Makefile still has **repeated content**, namely the name of our script, `countwords.py`. If we renamed our script we'd have to update our Makefile in multiple places.

- We can introduce a Make **variable** (called a **macro** in some versions of Make) to hold our script name:
  `COUNT_SRC=countwords.py`

- This is a variable assignment - `COUNT_SRC` is assigned the value `countwords.py`.

- `countwords.py` is our script and it is invoked by passing it to python.

Makefile: variables

- We can introduce another variable to represent this execution:
  `COUNT_EXE=python $(COUNT_SRC)`

- `$(...)` tells Make to **replace a variable with its value** when Make is run. This is a variable **reference**. At any place where we want to use the value of a variable we have to write it, or reference it, in this way.

- Here we reference the variable `COUNT_SRC`. This tells Make to replace the variable `COUNT_SRC` with its value `countwords.py`. When Make is run it will assign to `COUNT_EXE` the value python `countwords.py`.

- Defining the variable `COUNT_EXE` in this way allows us to easily change how our script is run (if, for example, we changed the language used to implement our script from Python to R).

Makefile: variables

Key Points

- Define variables by assigning values to names.
- Reference variables using `$(...)` or `${...}`.

Hands-on 4

Use Variables

- Update `Makefile` so that the `%.dat` rule references the variables `COUNT_SRC` and `COUNT_EXE`.
- Then do the same for the `zipf-test.py` script and the `results.txt` rule, using `ZIPF_SRC` and `ZIPF_EXE` as variable names

Hands-on 4 (original)

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

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

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

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

Hands-on 4 (after the changes)

```
COUNT_SRC=countwords.py
COUNT_EXE=python $(COUNT_SRC)
ZIPF_SRC=testzipf.py
ZIPF_EXE=python $(ZIPF_SRC)

# Generate summary table.
results.txt : $(ZIPF_SRC) isles.dat abyss.dat last.dat
    $(ZIPF_EXE) *.dat > $@

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

%.dat : books/%.txt $(COUNT_SRC)
    $(COUNT_EXE) $< $*.dat

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

Makefile: variable definitions

- We place variables at the top of a Makefile so they are easy to find and modify.
- Alternatively, we can pull them out into a new file that just holds variable definitions (i.e. delete them from the original makefile). Let us create `config.mk`:

```
# Count words script.
COUNT_SRC=countwords.py
COUNT_EXE=python $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=python $(ZIPF_SRC)
```

- We can then **import** `config.mk` into Makefile using:

```
include config.mk
```

- We can re-run Make to see that everything still works:

```
$ make clean
$ make dats
$ make results.txt
```

Makefile: variable definitions

- We have **separated** the **configuration** of our Makefile **from its rules**, the parts that do all the work.

- If we want to change our script name or how it is executed we just need to edit our configuration file, not our source code in Makefile.

- **Decoupling** the code from configuration in this way is good programming practice, as it promotes more **modular, flexible and reusable** code.

Makefile: functions

- **Questions**
  - How *else* can I eliminate redundancy in my Makefiles?
- **Objectives**
  - Write Makefiles that use built-in functions to match and transform sets of files.

Makefile: functions

At this point, we have the following Makefile:

```
include config.mk

# Generate summary table.
results.txt : $(ZIPF_SRC) isles.dat abyss.dat last.dat
	$(ZIPF_EXE) *.dat > $@

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

%.dat : books/%.txt $(COUNT_SRC)
	$(COUNT_EXE) $< $*.dat

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

Makefile: functions

- Make has many built-in functions which can be used to write more complex rules. One example is **`wildcard`**.
- **`wildcard`** gets a list of files matching some pattern, which we can then save in a variable.
- For example, we can get a list of all our text files (files ending in `.txt`) and save these in a variable by adding this at the beginning of our makefile:
  ```
  TXT_FILES=$(wildcard books/*.txt)
  ```
- We can add a `.PHONY` target and rule to show the variable's value:
  ```
  .PHONY : variables
  variables:
      @echo TXT_FILES: $(TXT_FILES)
  ```

Makefile: functions

@echo

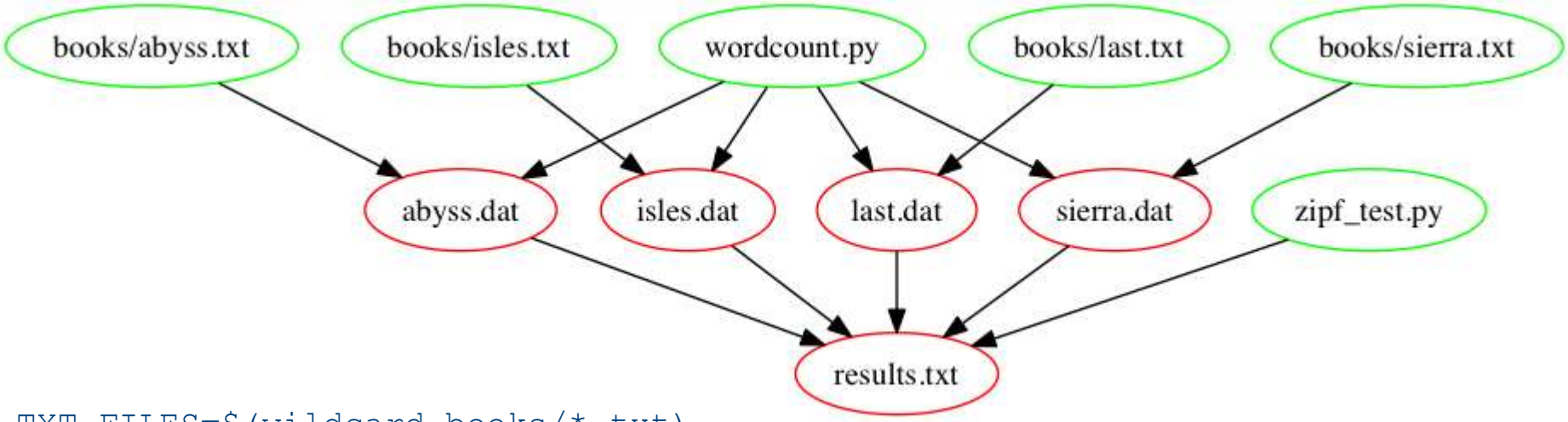
- Make prints actions as it executes them. Using `@` at the **start of an action** tells Make **not to print** this action. So, by using @echo instead of echo, we can see the result of echo (the variable's value being printed) but not the echo command itself.

- If we run Make:
  ```
  $ make variables
  ```
  We get:
  ```
  TXT_FILES: books/abyss.txt books/isles.txt books/last.txt
  books/sierra.txt
  ```

- Note how `sierra.txt` is now included too.

Makefile: functions

Dependencies embodied within our Makefile, involved in building the results.txt target, once we have introduced our function:




```
TXT_FILES=$(wildcard books/*.txt)

.PHONY : variables
variables:
	@echo TXT_FILES: $(TXT_FILES)

$ make variables
TXT_FILES: books/abyss.txt books/isles.txt books/last.txt books/sierra.txt
```

Makefile: pattern substitution

- `patsubst` (pattern substitution) takes a **pattern, a replacement string and a list of names** in that order; each name in the list that matches the pattern is replaced by the replacement string. Again, we can save the result in a variable. So, for example, we can rewrite our list of text files into a list of data files (files ending in .dat) and save these in a variable:

```
DAT_FILES=$(patsubst books/%.txt,%.dat, $(TXT_FILES))
```

- We can extend variables to show the value of `DAT_FILES` too:

```
.PHONY : variables
variables:
	@echo TXT_FILES: $(TXT_FILES)
	@echo DAT_FILES: $(DAT_FILES)
```

- If we run Make,

```
$ make variables
```

then we get:

```
TXT_FILES: books/abyss.txt books/isles.txt books/last.txt books/sierra.txt
DAT_FILES: abyss.dat isles.dat last.dat sierra.dat
```

- Now, `sierra.txt` is processed too.

Makefile: pattern substitution

- With these we can rewrite clean and dats:

```
.PHONY : dats
dats : $(DAT_FILES)
.PHONY : clean
clean :
    rm -f $(DAT_FILES)
    rm -f results.txt
```

- Let's also tidy up the `%.dat` rule by using the automatic variable `$@` instead of `$*.dat`:

```
%.dat : books/%.txt $(COUNT_SRC)
    $(COUNT_EXE) $< $@
```

- Let's check:

```
$ make clean
$ make dats
```

We get:

```
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/isles.txt isles.dat
python countwords.py books/last.txt last.dat
python countwords.py books/sierra.txt sierra.dat
```

Makefile: pattern substitution

- We can also rewrite `results.txt`:

```
results.txt : $(DAT_FILES) $(ZIPF_SRC)
    $(ZIPF_EXE) $(DAT_FILES) > $@
```

- If we re-run Make:

```
$ make clean
$ make results.txt
```

We get:

```
python countwords.py books/abyss.txt abyss.dat
python countwords.py books/isles.txt isles.dat
python countwords.py books/last.txt last.dat
python countwords.py books/sierra.txt sierra.dat
python testzipf.py last.dat isles.dat abyss.dat sierra.dat > results.txt
```

- Let's check the results.txt file:

```
$ cat results.txt
  Book    First   Second  Ratio
  abyss   4044    2807    1.44
  isles   3822    2460    1.55
  last    12244   5566    2.20
  sierra  4242    2469    1.72
```

So the range of the ratios of occurrences of the two most frequent words in our books is indeed around 2, as predicted by Zipf's Law, i.e., the most frequently-occurring word occurs approximately twice as often as the second most frequent word.

Makefile: pattern substitution

Final Makefile:

```
include config.mk

TXT_FILES=$(wildcard books/*.txt)
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))

# Generate summary table.
results.txt : $(DAT_FILES) $(ZIPF_SRC)
	$(ZIPF_EXE) $(DAT_FILES) > $@

# Count words.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
	$(COUNT_EXE) $< $@

.PHONY : clean
clean :
	rm -f $(DAT_FILES)
	rm -f results.txt

.PHONY : variables
variables:
	@echo TXT_FILES: $(TXT_FILES)
	@echo DAT_FILES: $(DAT_FILES)
```

Makefile: pattern substitution

`config.mk` file contains:

```
# Count words script.
COUNT_SRC=countwords.py
COUNT_EXE=python $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=python $(ZIPF_SRC)
```

Makefile: functions

Key Points

- Make is actually a small programming language with many built-in functions.
- Use `wildcard` function to get lists of files matching a pattern.
- Use `patsubst` function to rewrite file names.

Hands-on 5

Adding more books

- We can now do a better job at testing Zipf's rule by adding more books. The books we have used come from the Project Gutenberg website (http://www.gutenberg.org/). Project Gutenberg offers thousands of free ebooks to download.

Exercise instructions:

- go to Project Gutenberg and use the search box to find another book, for example *'The Picture of Dorian Gray'* from Oscar Wilde.
- download the 'Plain Text UTF-8' version and save it to the books folder; choose a short name for the file (that doesn't include spaces) e.g. "dorian_gray.txt" because the filename is going to be used in the results.txt file
- optionally, open the file in a text editor and remove extraneous text at the beginning and end (look for the phrase End of Project Gutenberg's [title], by [author])
- run make and check that the correct commands are run, given the dependency tree
- check the results.txt file to see how this book compares to the others

Self-Documenting Makefiles

- **Questions**
  - How should I document a Makefile?
- **Objectives**
  - Write self-documenting Makefiles with built-in help.

Self-Documenting Makefiles

- Many bash commands, and programs that people have written that can be run from within bash, support a `--help` flag to display more information on how to use the commands or programs.

- In this spirit, it can be useful, both for ourselves and for others, to provide a `help` target in our Makefiles. This can provide a summary of the names of the key targets and what they do, so we don't need to look at the Makefile itself unless we want to.

- For our Makefile, running a help target might print:

```
$ make help
  results.txt : Generate Zipf summary table.
  dats        : Count words in text files.
  clean       : Remove auto-generated files.
```

Self-Documenting Makefiles

- How would we implement this? We could write a rule like:

```
.PHONY : help
help :
    @echo "results.txt : Generate Zipf summary table."
    @echo "dats        : Count words in text files."
    @echo "clean       : Remove auto-generated files."
```

- But **every time** we add or remove a rule, or change the description of a rule, we would have to **update this rule too**. It would be better if we could keep the descriptions of the rules by the rules themselves and **extract these descriptions automatically**.

- The bash shell can help us here. It provides a command called **sed** which stands for ‘stream editor’. Sed reads in some text, does some filtering, and writes out the filtered text.

Self-Documenting Makefiles

- We could write comments for our rules, and mark them up in a way which `sed` can detect.
- Since Make uses `#` for comments, we can use **`##`** for comments that describe what a rule does and that we want `sed` to detect. For example:

```
## results.txt : Generate Zipf summary table.
results.txt : $(DAT_FILES) $(ZIPF_SRC)
	$(ZIPF_EXE) $(DAT_FILES) > $@

## dats        : Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
	$(COUNT_EXE) $< $@

## clean       : Remove auto-generated files.
.PHONY : clean
clean :
	rm -f $(DAT_FILES)
	rm -f results.txt

## variables   : Print variables.
.PHONY : variables
variables:
	@echo TXT_FILES: $(TXT_FILES)
	@echo DAT_FILES: $(DAT_FILES)
```

Self-Documenting Makefiles

- We use **##** so we can distinguish between comments that we want sed to automatically filter, and other comments that may describe what other rules do, or that describe variables.
- We can then write a help target that applies sed to our Makefile:

```
.PHONY : help
help : Makefile
	@sed -n 's/^##//p' $<
```

- This rule depends upon the Makefile itself. It runs sed on the first dependency of the rule, which is our Makefile, and tells `sed` to get all the lines that begin with `##`, which sed then prints for us.
- If we now run

```
$ make help
```

we get:

```
results.txt : Generate Zipf summary table.
dats        : Count words in text files.
clean       : Remove auto-generated files.
variables   : Print variables.
```

Self-Documenting Makefiles

- If we **add, change or remove a target or rule**, we now only need to remember to **add, update or remove a comment next to the rule**.
- So long as we respect our convention of using **##** for such comments, then our **help** rule will take care of detecting these comments and printing them for us.

Self-Documenting Makefiles

Key Points

- Document Makefiles by adding specially-formatted comments and a target to extract and format them.

Makefiles: Pros & Cons

- **Questions**
  - What are the advantages and disadvantages of using tools like Make?
- **Objectives**
  - Understand advantages of automated build tools such as Make.

Makefiles: Pros & Cons

- Automated build tools such as Make can help us in a number of ways. They help us to **automate repetitive commands**, hence **saving us time** and **reducing the likelihood of errors** compared with running these commands manually.

- They can also save time by ensuring that automatically-generated artifacts (such as data files or plots) are **only recreated when** the files that were used to create these have **changed in some way**.

- Through their notion of targets, dependencies, and actions, they serve as a **form of documentation**, recording dependencies between code, scripts, tools, configurations, raw data, derived data, plots, and papers.

Hands-on 6

Creating PNGs

1. Add new rules, update existing rules, and add new variables to:
   - Create `.png` files from `.dat` files using `plotcounts.py`.
   - Remove all auto-generated files (`.dat, .png, results.txt`).
2. Finally, many Makefiles define a default phony target called `all` as first target, that will build what the Makefile has been written to build (e.g. in our case, the `.png` files and the `results.txt` file). As others may assume your Makefile conforms to convention and supports an `all` target, add an **`all`** target to your Makefile (Hint: this rule has the `results.txt` file and the `.png` files as dependencies, but no actions). With that in place, instead of running `make results.txt`, you should now run `make all`, or just simply `make`. By default, make runs the first target it finds in the Makefile, in this case your new `all` target.

Hands-on 6 (original)

```
include config.mk

TXT_FILES=$(wildcard books/*.txt)
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))

# Generate summary table.
results.txt : $(DAT_FILES) $(ZIPF_SRC)
	$(ZIPF_EXE) $(DAT_FILES) > $@

# Count words.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
	$(COUNT_EXE) $< $@

.PHONY : clean
clean :
	rm -f $(DAT_FILES)
	rm -f results.txt

.PHONY : variables
variables:
	@echo TXT_FILES: $(TXT_FILES)
	@echo DAT_FILES: $(DAT_FILES)
```

config.mk:

```
# Count words script.
COUNT_SRC=countwords.py
COUNT_EXE=python $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=python $(ZIPF_SRC)
```

Hands-on 6 (after the changes)

config.mk:

```
# Count words script.
COUNT_SRC=countwords.py
COUNT_EXE=python $(COUNT_SRC)

# Plot word counts script.
PLOT_SRC=plotcounts.py
PLOT_EXE=python $(PLOT_SRC)

# Test Zipf's rule
ZIPF_SRC=testzipf.py
ZIPF_EXE=python $(ZIPF_SRC)
```

Hands-on 6 (after the changes)

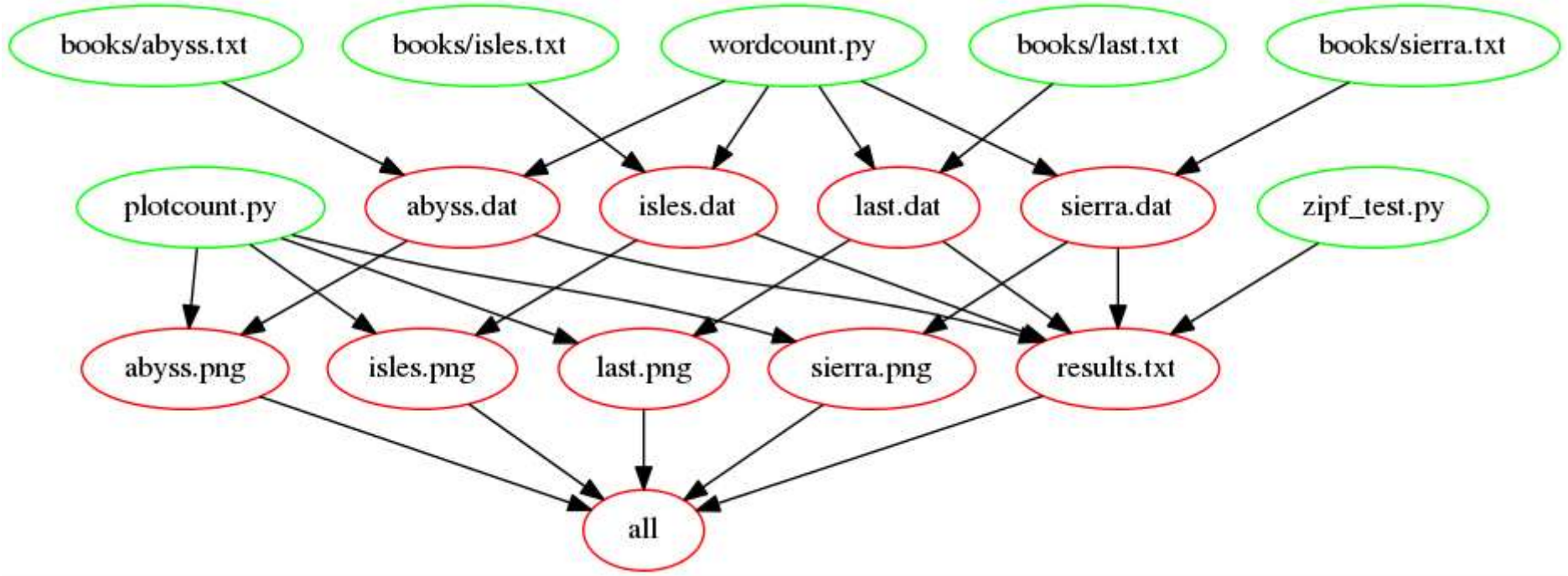
```
include config.mk

TXT_FILES=$(wildcard books/*.txt)
DAT_FILES=$(patsubst books/%.txt,%.dat, $(TXT_FILES))
PNG_FILES=$(patsubst books/%.txt,%.png, $(TXT_FILES))

## all          : Generate Zipf summary table and plots of word counts.
.PHONY : all
all : results.txt $(PNG_FILES)

## results.txt : Generate Zipf summary table.
results.txt : $(DAT_FILES) $(ZIPF_SRC)
	$(ZIPF_EXE) $(DAT_FILES) > $@

## dats         : Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
	$(COUNT_EXE) $< $@

## pngs         : Plot word counts.
.PHONY : pngs
pngs : $(PNG_FILES)
```

Hands-on 6 (after the changes)

```
%.png : %.dat $(PLOT_SRC)
	$(PLOT_EXE) $< $@

## clean       : Remove auto-generated files.
.PHONY : clean
clean :
	rm -f $(DAT_FILES)
	rm -f $(PNG_FILES)
	rm -f results.txt

## variables   : Print variables.
.PHONY : variables
variables:
	@echo TXT_FILES: $(TXT_FILES)
	@echo DAT_FILES: $(DAT_FILES)
	@echo PNG_FILES: $(PNG_FILES)

.PHONY : help
help : Makefile
	@sed -n 's/^##//p' $<
```

Makefiles: Pros & Cons

The dependencies involved in building the `all` target with support for images added:



Makefiles: Creating an Archive

- Often it is useful to **create an archive file of your project** that includes all data, code and results. An archive file can package many files into a single file that can easily be downloaded and shared with collaborators. We can add steps to create the archive file inside the Makefile itself so it's easy to update our archive file as the project changes.
- Edit the Makefile to create an archive file of your project. Add new rules, update existing rules and add new variables to:
  - Create a new directory called `zipf_analysis` in the project directory.
  - Copy all our code, data, plots and the Zipf summary table to this directory. The `cp -r` command can be used to copy files and directories into the new zipf_analysis directory:

```
$ cp -r [files and directories to copy] zipf_analysis/
```

- Hint: create a new variable for the `books` directory so that it can be copied to the new `zipf_analysis` directory

Hands-on 7

- Create an archive, `zipf_analysis.tar.gz`, of this directory. The bash command `tar` can be used, as follows:

  ```
  $ tar -czf zipf_analysis.tar.gz zipf_analysis
  ```

- Update all to create zipf_analysis.tar.gz.
- Remove zipf_analysis.tar.gz when make clean is called.
- Print the values of any additional variables you have defined when make variables is called.

Hands-on 7

include config.mk

unchanged

```
TXT_DIR=books
TXT_FILES=$(wildcard $(TXT_DIR)/*.txt)
DAT_FILES=$(patsubst $(TXT_DIR)/%.txt, %.dat, $(TXT_FILES))
PNG_FILES=$(patsubst $(TXT_DIR)/%.txt, %.png, $(TXT_FILES))
RESULTS_FILE=results.txt
ZIPF_DIR=zipf_analysis
ZIPF_ARCHIVE=$(ZIPF_DIR).tar.gz

## all        : Generate archive of code, data, plots and Zipf summary table.
.PHONY : all
all : $(ZIPF_ARCHIVE)

$(ZIPF_ARCHIVE) : $(ZIPF_DIR)
	tar -czf $@ $<

$(ZIPF_DIR): Makefile config.mk $(RESULTS_FILE) \
	$(DAT_FILES) $(PNG_FILES) $(TXT_DIR) \
	$(COUNT_SRC) $(PLOT_SRC) $(ZIPF_SRC)
	mkdir -p $@
	cp -r $^ $@
	touch $@
```

Hands-on 7

```
 ## results.txt : Generate Zipf summary table.
$(RESULTS_FILE) : $(DAT_FILES) $(ZIPF_SRC)
		$(ZIPF_EXE) $(DAT_FILES) > $@

## dats		: Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : $(TXT_DIR)/%.txt $(COUNT_SRC)
		$(COUNT_EXE) $< $@

## pngs		: Plot word counts.
.PHONY : pngs
pngs : $(PNG_FILES)

%.png : %.dat $(PLOT_SRC)
		$(PLOT_EXE) $< $@
```

Hands-on 7

```
## clean      : Remove auto-generated files.
.PHONY : clean
clean :
		rm -f $(DAT_FILES)
		rm -f $(PNG_FILES)
		rm -f $(RESULTS_FILE)
		rm -rf $(ZIPF_DIR)
		rm -f $(ZIPF_ARCHIVE)

## variables   : Print variables.
.PHONY : variables
variables:
		@echo TXT_DIR: $(TXT_DIR)
		@echo TXT_FILES: $(TXT_FILES)
		@echo DAT_FILES: $(DAT_FILES)
		@echo PNG_FILES: $(PNG_FILES)
		@echo ZIPF_DIR: $(ZIPF_DIR)
		@echo ZIPF_ARCHIVE: $(ZIPF_ARCHIVE)

.PHONY : help
help : Makefile
		@sed -n 's/^##//p' $<
```

Makefiles: Creating an Archive

Archiving the Makefile

- Why does the Makefile rule for the archive directory add the Makefile to our archive of code, data, plots and Zipf summary table?

- **Solution**
  - Our code files (`countwords.py`, `plotcounts.py`, `testzipf.py`) implement the individual parts of our workflow. They allow us to create `.dat` files from `.txt` files, `.png` files from `.dat` files and `results.txt`.
  - Our Makefile, however, documents dependencies between our code, raw data, derived data, and plots, as well as implementing our workflow as a whole. `config.mk` contains configuration information for our Makefile, so it must be archived too.

Makefiles: Creating an Archive

touch the Archive Directory

- Why does the Makefile rule for the archive directory `touch` the archive directory after moving our code, data, plots and summary table into it?

- **Solution**
  - A directory's timestamp is not automatically updated when files are copied into it. If the code, data, plots, and summary table are updated and copied into the archive directory, the archive directory's timestamp must be updated with `touch` so that the rule that makes `zipf_analysis.tar.gz` knows to run again;
  - without this `touch`, `zipf_analysis.tar.gz` will only be created the first time the rule is run and will not be updated on subsequent runs even if the contents of the archive directory have changed.

Makefiles: Pros & Cons

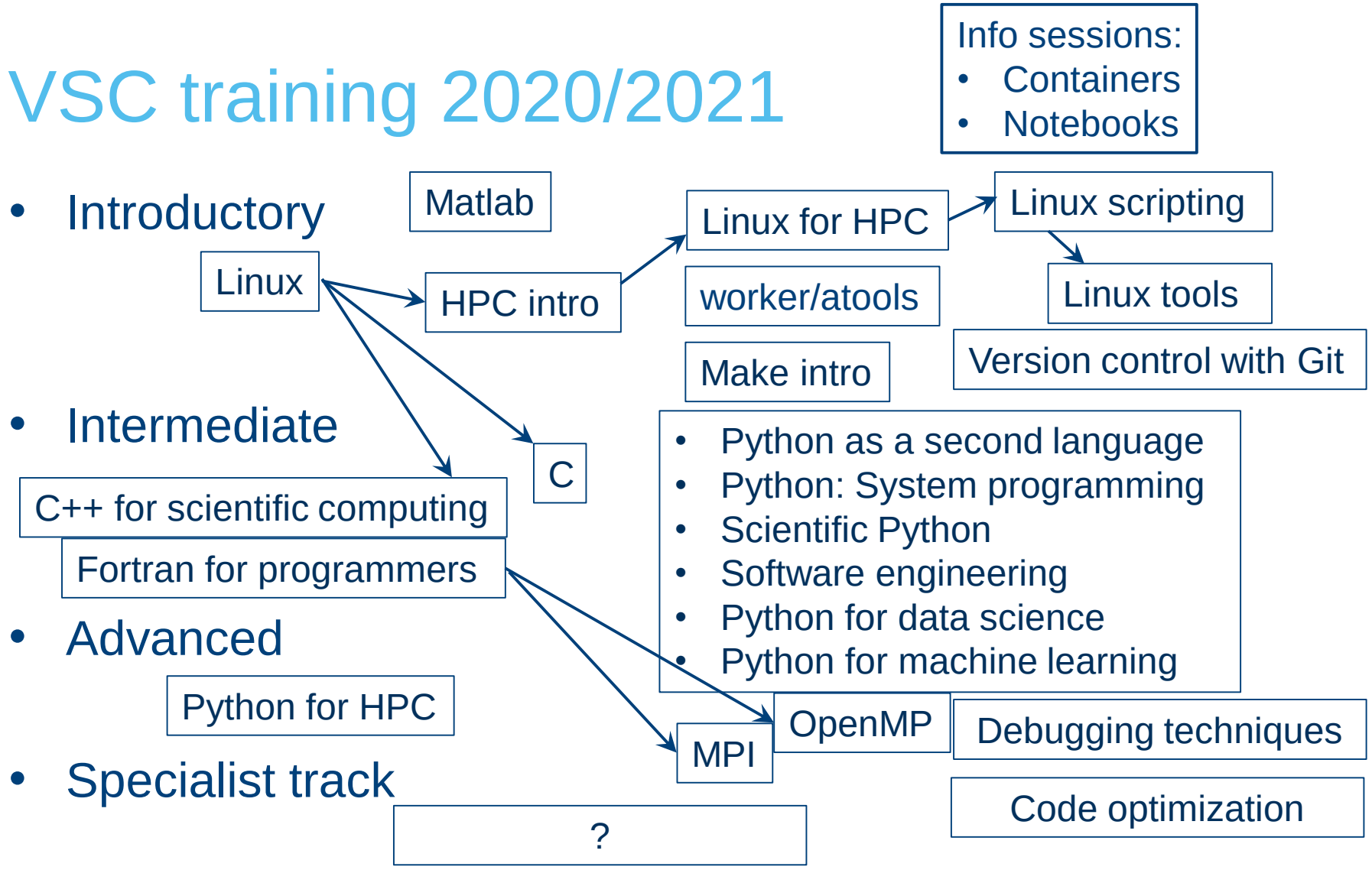
Key Points

- Makefiles save time by automating repetitive work, and save thinking by documenting how to reproduce results.

Questions

- Now
- Helpdesk:
  hpcinfo@kuleuven.be or
  https://admin.kuleuven.be/icts/HPCinfo_form/HPC-info-formulier
- VSC web site:
  http://www.vscentrum.be/
  - VSC documentation: https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/
    VSC agenda: training sessions, events
- Systems status page:
  http://status.kuleuven.be/hpc

VSC training 2020/2021



Stay up-to-date https://www.vscentrum.be/training

Hands-on 8

Compiling LAPACK as a shared library in linux

- Download LAPACK (e.g. version 3.3.0) from http://www.netlib.org/lapack/

- Modify the files

```
(1) {path}/make.inc
(2) {path}/SRC/Makefile
(3) {path}/BLAS/SRC/Makefile
(4) {path}/Makefile
```

Hands-on 8

- (1) For `{path}/make.inc`:
  add "`-fPIC`" to the configuration variables
  `FORTRAN`
  `OPTS`
  `LOADER`

  I also appended "-02" for optimization and speedier compilation on to `FORTRAN` and `LOADER`

  For ease of installation, add a new variable, `PREFIX`, which designates the installation path:

```
#INSTALLATION PATH
PREFIX=/data/leuven/304/vsc30468/shared_libraries/lapack/3
.3.0/
```

Hands-on 8

- (2) For `{path}/SRC/Makefile`:
  Update the main target:

```
all: ../$(LAPACKLIB) liblapack.so
```

Add the following build target and rule

```
liblapack.so: $(ALLOBJ)
    gfortran -shared -Wl,-soname,$@ -o $@ $(ALLOBJ)
```

Remove some redundancy from the object files list:

```
#DSLASRC = spotrs.o sgetrs.o spotrf.o sgetrf.o
DSLASRC = spotrs.o sgetrs.o sgetrf.o

#ZCLASRC = cpotrs.o cgetrs.o cpotrf.o cgetrf.o
ZCLASRC =
```

Hands-on 8

- (3) For `{path}/BLAS/SRC/Makefile`:
  Update the main target to read as follows

```
all: $(BLASLIB) libblas.so
```

You can create an alias in the `{path}/make.inc` for `BLASLIBSO = libblas.so`

Add the new target:

```
libblas.so: $(ALLOBJ)
    gfortran -shared -Wl,-soname,$@ -o $@ $(ALLOBJ)
```

Hands-on 8

- (4) For `{path}/Makefile`
  Added an install rule which depends on the `PREFIX` setting from `make.inc`:

```
install: all
cp BLAS/SRC/libblas.so $(PREFIX)/lib
cp SRC/liblapack.so $(PREFIX)/lib
cp *.a $(PREFIX)/lib
```