Sake

A self-documenting build automation tool
Version 0.9.7 (2016-03-29)

Tony Fischetti

This manual is for sake, version 0.9.7

Table of Contents

1 Introduction

1.1 What is it?

Sake is a way to easily design, share, build, and visualize workflows with intricate interdependencies. Sake is self-documenting because the instructions for building a project also serve as the documentation of the project's workflow. The first time it's run, sake will build all of the components of a project in an order that automatically satisfies all dependencies. For all subsequent runs, sake will only rebuild the parts of the project that depend on changed files. This cuts down on unnecessary re-building and lets the user concentrate on their work rather than memorizing the order in which commands have to be run.

Sake is free, open source cross-platform software under a very permissive license (MIT Expat) and is written in Python.

1.2 Who is it for?

Sake's insistence on clean formatting, explicit statement of intentions, capacity for visualization, ability to work on various platforms, and ability to rebuild only what is needed make it a great choice for

- Scientists that want to share their scientific workflow with other researchers. Sake helps facilitate open science and reproducibility.
- Data analysts with steps in their pipeline that take hours or days to finish running.
- Business teams that want to share and visualize workflows amongst its members, even if they are using different computing platforms.

1.3 Nomenclature

To avoid confusion, it would be helpful to expound on some of the terms that this documentation will be using.

`workflow` a series of steps performed in a particular order to complete a piece of work.

`scientific workflow`
: a specific type of workflow where many or all of the steps are performed by computers. This is a growing trend in science. Data science, bioinformatics, cognitive science, and computational linguistics, in particular, make heavy use of this paradigm.

`pipeline` Another name for a workflow, especially one whose steps produce outputs that are fed, as input, into subsequent steps.

`dependency`
: An input or step that is required for another step to execute is said to be a `dependency` of the later step.

`graph` a construct in mathematics that represents a set of objects (called 'nodes' or 'vertices') and the connections between them (called 'links' or 'edges')

`directed graph`
: A type of graph whose connections are unidirectional and can be visually represented as arrows.

`directed acyclic graph`
: A type of directed graph that does not contain any loops or cycles. A directed graph has the property of having at least one ordering of nodes such that no node in the ordering 'points to' a node earlier in the ordering.

`dependency graph`
: If the nodes of a directed acyclic graph are viewed as steps in a workflow or pipeline, and the directional links are viewed as outputs that become another step's input, this construct of mathematics is an abstraction of a workflow that can be run in an order that satisfies all dependencies. A visual representation of a dependency graph is an easy and intuitive way to view a workflow and the steps involved.

`YAML`
: A data serialization format that is extremely easy to read.

`Sakefile`
: A file written in a subset of YAML that describes a workflow. This is read and can be executed or visualized by `sake`.

`target`
: This is the abstraction of a step in a workflow used by the `sakefile`. A `sakefile` is made of one or more targets. This name is borrowed from the `make` software.

`build`
: (Used as a verb). The execution of the commands of a step. A whole project can be built, or just a specific step. The 'build' can be the execution of a script, the execution of arbitrary shell commands, the removal of intermediate files, the compilation of source files, etc... A build can take microseconds or days to complete.

1.4 Differences from GNU Make or other `make` software

The ideas behind `sake` and `sakefiles` are borrowed heavily from `make` software, which is used primarily to build executables from various source files. A `makefile` can, more generally, be used to describe a process that brings one or more source files to some 'completed' state. Probably nothing is better than `make` to compile executables, but there are several reasons why using `sake` may be a better choice for some tasks.

- As a consequence of being so powerful for source code compilation, `Makefiles` can sometimes be very hard to read and write, particularly to the unfamiliar.
- Most `make` software assumes that if the timestamp of a file changes, than all subsequent steps that depend on that file need to be run in order to remain up-to-date. In many cases, though, the time-stamp of a file can change but the contents remain the same and, thus, a rebuild isn't necessary.
- The syntax of `makefiles` makes it difficult to intuit the flow of a pipeline. Additionally, outside tools have to be used in order to visualize the flow.
- Steps like displaying help and cleaning intermediate files are not handled automatically by `make` and are prone to errors.

`Sake` seeks to rectify these limitations of `make`. To wit,

- Sakefiles are written in a very easy-to-read-and-write markup language.

- `Sake` was also not designed with a particular application in mind, so the use of `sake` is just as applicable to one domain as it is to another. This cuts down on the number of tools a user must learn to do certain specialized tasks.
- Originally borne out of the frustration of rebuilding targets whenever a timestamp of a file changes and, therefore, being difficult to use for data analysis with very long time-consuming analytics, `sake` actually *reads* the file to determine if a re-building is really necessary.
- The clean nature of the `sakefile` makes it much easier to intuit the flow of a pipeline. Additionally, a visualization mechanism is built right in which produces an image of the dependency graph that is easy to study, to share, and is aesthetically pleasing.
- Sake handles some 'administrative' tasks for the user. This cuts down on hard-to-track-down errors.

Perhaps the most fundamental difference between `sake` and the `make` system is that `sake` seeks not only to be a system that builds a project from start to completion (or part of a project) resolving all dependencies along the way; `sake` seeks to simultaneously *document* the project.

There are some more technical differences between the two systems that may confuse a user that is familiar with `make`. For example, all 'targets' in a `sakefile` are what GNU Make considers 'phony' targets. Targets in `sakefiles` are not filenames, but instead more human-readable short descriptions of the step. Other differences will become clearer later in the document.

2 Installation

Here we will go over obtaining the `sake` software, go over generic installation instructions, and discuss detailed installation instructions for various platforms.

Note: Some of the commands in this section may require root or administrator privileges depending on your system

2.1 Getting Sake

Since sake is a set of python modules and a python driver, there are no binary executables that can be downloaded. Although, in most cases, *manually* obtaining the source code for sake is unecessary for installation, obtaining the sake source can be done in several ways:

- A tarball can be downloaded from http://pypi.python.org/pypi/master-sake (`http://pypi.python.org/pypi/master-sake`). Only the most stable releases get uploaded to the Python Package Index, so this is a great option for users that are looking for stability.
- A tarball or zip file of sake is available for download on this project's webpage at http://tonyfischetti.github.io/sake/ (`http://tonyfischetti.github.io/sake/`). This is the latest snapshot of the master branch of sake from its git repository, where it is developed. This is the second-most stable version of the sake source. This is perhaps more feature-rich than the source on PyPI, but it may contain bugs that would hopefully be found and fixed before upload to PyPI.
- Finally, the bleeding-edge sake source can be cloned directly from the git repository thusly: (assuming you have git installed)

```
git clone https://github.com/tonyfischetti/sake.git
```

For the vast majority of cases, the best way to obtain sake is during the installation step, with the python package manager `pip`.

2.2 Dependencies

There are four pieces of open source software that must be installed to ensure the proper functioning of sake:

- Python. Sake is Python3 compatible and is tested using Python 2.7, 3.2, 3.3, 3.4, and 3.5
- The python module `NetworkX`
- The python module `PyYAML`
- The graph visualization tool `Graphviz`

Future versions of sake may shed some of these dependencies.

2.3 Generic Installation Instructions

Note: performing all of these steps are usually unnecessary depending on what system you are using. If you do not care to know how sake can be installed generically, you should read the installation instructions for your specific platform (if available) below.

If you do not already have it, Python can be installed by following the links on the Python.org website: http://www.python.org/download/ (`http://www.python.org/download/`). There are executable installers for Windows and Mac OS X. There are also download links to the Python source code for compiling on Unix and Unix-alike systems though, for these systems, there is very often a package manager available that will facilitate a much easier python installation process. If the platform-specific instructions are not available below, a quick search of the web for

```
<your system here> python install
```

can save you a lot of trouble.

The two python packages and sake itself can be installed in at least two ways.

- One uses the python package manager `pip`. After `pip` is installed, these python packages can be installed from the command-line thusly:

  ```
  pip install master-sake
  ```

  `pip` installing `master-sake` should automatically install dependencies `NetworkX` and `PyYAML`. If there is a problem and it doesn't do this automatically, the dependencies can be easily installed by issuing these commands:

  ```
  pip install networkx
  pip install pyyaml
  ```

  `pip` itself can be installed via:

  ```
  easy_install pip
  ```

  Finally, `easy_install` can be installed with instructions from this website: https://pypi.python.org/pypi/setuptools (`https://pypi.python.org/pypi/setuptools`)
- Another way to install these python packages is by downloading a tarball of the source of these packages, extracting the archive and running the `setup.py` script inside the extracted archive with

  ```
  python setup.py install
  ```

Finally, Graphviz can be installed easily by following the directions on the Graphviz download page: http://www.graphviz.org/Download..php (`http://www.graphviz.org/Download..php`). Easy-to-install packages are available for Mac OS X, Windows, GNU/Linux, and Solaris.

2.4 Installation for Mac OS X

Python and `easy_install` are already installed and ready to use on OS X. Open `Terminal.app` (usually in /Applications/Utilities) and issue the following commands:

```
sudo easy_install pip
sudo pip install master-sake
```

The use of `sudo` will prompt you for your login password, if applicable.

Finally, Mac-specific Graphviz installers are available at the Graphviz site at http://www.graphviz.org/Download_macos.php (`http://www.graphviz.org/Download_macos.php`). The Mountain Lion installer will work with Mavericks and Yosemite.

If you are using a version of Python that is not the version that ships with OS X, you should follow the 'Generic Installation Instructions'.

2.5 Installation for GNU/Linux (Debian)

Debian and some Debian-based GNU/Linux distributions make it very easy to install sake and all of its dependencies. Issue the following commands in a terminal emulator:

```
sudo apt-get install python-networkx
sudo apt-get install python-pip
sudo pip install master-sake
```

The `python-networkx` will automatically install PyYAML, and Graphviz.

If for some strange reason, Python is missing on your Debian system, you can install it using `apt-get`, as well.

For other GNU/Linux distributions, check to see if your system has a package management tool and whether these (or analogous) packages are available for that tool. `yum` and `pacman` are some other popular package managers, although there are many others.

2.6 Installation on Windows

Note: These instructions will assume that Python 2.7 is used. The same installation instructions will work with Python 3.*, if the appropriate version numbers are changed. These installation instructions also assume the use of Windows 7. The instructions should be the same, or very similar for other versions of Windows

1. Download the Windows Python installer at http://www.python.org/download/ (`http://www.python.org/download/`)
2. Double-click the installer. A default install will install Python in

   ```
   C:\Python27\  or
   C:\Python33\
   ```

3. Open the Control Panel. In the search bar, type

   ```
   environment
   ```

   and click on the entry that reads

   ```
   Edit environment variables for your account
   ```

4. In the top-most list box, find the entry that reads `Path`. (If it doesn't exist, you have to add it.) Click on the `Edit...` button and append the following to the `Variable value` field

   ```
   ;C:\Python27;C:\Python27\Scripts
   ```

   of course, changing the Python version number if necessary.

5. Download the Python script that bootstraps `easy_install` at

   ```
   https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py (https://
   bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py)
   ```

   Place this file somewhere you can find with the command prompt. This will assume you named it `ez_setup.py`.

6. Open a command prompt window (Windows key + r, type "cmd", press <Enter>)

7. Navigate to the directory where ez_setup.py resides and run it with Python. As an example,

   ```
   cd C:\Users\me\Desktop
   python ez_setup.py
   ```

8. After the last command finishes, close the command prompt window, open a new one, and issue the following commands:

   ```
   easy_install pip
   pip install master-sake
   ```

9. Open up a command prompt with Administrator privileges. You can press the Windows key, type `cmd`, right click the Program "cmd" at the top of the menu and choose "Run as administrator". In the same command prompt window navigate to the Python scripts directory:

   ```
   cd C:\Python27\Scripts
   ```

   and issue the command

   ```
   mklink sake.py sake
   ```

10. Download and install a graphviz windows intaller from their website: http://www.graphviz.org/Download_windows.php (`http://www.graphviz.org/Download_windows.php`).

11. Find the location of the bin directory of the Graphviz install. Add it to the `Path` variable just like in steps 3 and 4. This time, append the location of the bin directory of Graphviz. It should look something like this.

    ```
    ;C:\Program Files (x86)\Graphviz2.34\bin
    ```

12. Open the Control Panel. In the search bar, type

    ```
    environment
    ```

    and this time click on the entry that reads

    ```
    Edit the system environment variables
    ```

13. Click the button (on the bottom) that reads "Environment Variables...". In the bottom-most list box, find the entry that reads `PATHEXT`. Click on the `Edit...` button and append the following to the `Variable value` field

    ```
    ;.PY
    ```

14. Close all command prompt windows, start a new one and issue the command

    ```
    sake -V
    ```

    to make sure that everything worked.

2.7 Updating sake

If sake is installed using `pip`, updating is as easy as running the following command:

```
pip install --upgrade master-sake
```

If sake was installed through the tarball and `python setup.py install`, you can just download the latest tarball and run the installation command again.

3 Using Sake

3.1 Running Sake

Sake is a command-line tool. After proper installation, running sake is just a matter of typing `sake` into your terminal. To see all the options, you can type:

```
sake -h
```

Sake expects certain things of the directory you run it from. Without any options, it looks for files called `"Sakefile"`, `"Sakefile.yaml"`, `"Sakefile.yml"` (in that order) and uses the first one it finds. The `Sakefile` is the instruction manual for building a project, and is the subject of the next section. Almost all of the command-line flags and arguments passed into the `sake` executable just modify how it treats and interacts with the Sakefile instructions.

After running `sake` for the first time, a file is deposited in the current directory named `".shastore"`. This hidden file is another YAML document that stores the SHA1 hashes of all the dependencies and output files indicated in the Sakefile. This is how sake determines which dependencies have changed and what targets need to be rebuilt.

3.2 The Sakefile

The sakefile, which serves as the instruction manual for sake builds, is 100% parse-able YAML. The structure of the YAML document enforces simplicity and readability.

The Sakefile format as `sake` requires it, is a subset of the full YAML specification. There are only a few structures that can appear in the document:

- targets
- meta-targets
- the "all" special target
- macro definitions
- an arbitrary number of comments

All valid Sakefiles must contain at least one target. All other elements are, strictly speaking, optional.

3.3 Anatomy of a target

The "target" is an isolated collection of key-value pairs that tells sake how to build one element that makes up the whole project. Before we get into the details of all the necessary components of a target, it would behoove us to see a complete example of one:

```
format raw data:
    help: format raw (copy and pasted) data using perl
    dependencies:
        - raw-data.txt
    formula: >
        perl -pe 's/^(\D+)\s+([\d,]+)\s+([\d,]+)\s*/\1\t\2\t\3\n/'
        raw-data.txt | sed 's/,//g' > cleaned-data.tsv;
    output:
        - cleaned-data.tsv
```

3.3.1 Target title

The target names serve two purposes in sake projects.

- If you are looking to build only one part of a project (which includes all of that part's dependencies) the name of the target is the string you pass into the `sake` executable to denote that.
- When sake produces a dependency diagram, the target name is the name of the node in the visualization.

There are very few restrictions on what can be used as a valid target name. The only requirements are

1. that the YAML parser used by sake unambiguiously identify the name as a string
2. it is not named any of the following reserved names:
    - `help`
    - `visual`
    - `clean`, or
    - `all`
3. it does not contain parentheses. Parentheticals in target names have special meaning.

Since the target name can also serve as a (very) short description of what the target is for, the target name should aim to be descriptive. The target title in the example above is, or course, `format raw data`.

3.3.2 Help

The `help` value in the target is an arbitrarily long description of what the target is for. This is automatically parsed and output when the `sake help` command is run.

Since sake seeks to be a documentation tool, as well as an automated build system, the `help` element is mandatory.

3.3.3 Dependencies

The `dependencies` element is a list of all the files (by filename) that are required for the target to build. In the example above, the target only requires the input of file `raw-data.txt`, so that is the only dependency. The filenames may be written as an absolute path, or as a relative path. All relative paths are relative to the directory where `sake` is run. So the dependency entry may have been written like

```
- ./input/raw-data.txt
```

if it were stored in a directory under the current directory named `input`, or

```
- ../raw-data.txt
```

if it were in a directory above the current directory. Even though you can specify a file above the current working directory, it is strongly discouraged and runs contra to sake's aim for simplicity and being intuitive. Ideally, the Sakefile should appear in the top level directory of any given project and only reference files below it.

A dependency entry may also contain Unix-style 'wildcard' characters. For example, a dependency that looks like this:

```
- ./input/*
```

will tell sake that the target described depends on all the files in the directory named `input`. Even though sake supports the use of wildcards, for most use cases, wildcard usage should be discouraged because it obscures clarity and makes reading the Sakefile difficult.

If a target has no dependencies, the dependency field may be left out. A good example of this would be if there is a target that starts off a data pipeline and just downloads a CSV from the internet.

Sake determines which targets have to be run by whether any of its dependencies' content has changed. If a target has no dependencies, `sake` has to assume that the target needs to be rebuilt every time it is run. To stop this behavior, you can create a dummy dependency for the target that never changes, or changes only when you want to explicitly re-run the target. Alternatively, you can include the dependency field but leave it blank, like so

```
dependencies:
formula: >
....
```

This will also stop sake from running the target every time `sake` is called.

3.3.4 Formula

The `formula` field stores the command that carries out the building of a target. It is a string that gets executed as a system command. Anything that can be done in the shell from a terminal emulator can be used as a target's formula. In this way, the formula is like an arbitrarily long shell script. It can call other scripts, run programs, and send commands to specific interpreters. You can even get fancy with Applescript, VBScript, or Powershell and communicate with running applications or control menu-driven interfaces.

Remember that any scripts called from the formula should appear in the dependency list, or the formula will not be rerun if the script changes.

Every "target-proper" must have a formula. If a formula is missing from a target, `sake` inteprets it as a meta-target, which is the subject of a future section.

3.3.5 Output

This field is similar to the `dependency` field; it lists all of the files that are created as a result of the target's formula being run. This section serves three very important roles in sake:

1. It is used internally by sake to determine which targets have to be run and updated before others. If another target uses the output from the above target (namely, `cleaned-data.tsv`), that target needs to know that if the dependency of the target whose output it relies on has changed, its output has to be brought up to date in order for it's own output to be up to date.

2. It is recorded by sake so that when the `sake clean` command is run, the output files are automatically removed from the project's file structure.
3. It provides useful information to someone (a human) reading the Sakefile so they can tell where a file came from and why it's there.

As with the `dependencies` entries, output entries may contain wildcard characters but, again, this may obscure clarity. It is also potentially dangerous, as `sake clean` will remove files that match the wildcard that may be unintended.

3.3.6 Ignored attributes

An arbitrary number of other key-value attributes may be added to a target– sake will just ignore them. Even so, adding other attributes may be helpful for other reasons... consider the following addendum to the example target from above:

```
format raw data:
    help: format raw (copy and pasted) data using perl
    WARNING: the regular expression need to be rewritten for readibility
    dependencies:
        - raw-data.txt
    formula: >
        perl -pe 's/^(\D+)\s+([\d,]+)\s+([\d,]+)\s*/\1\t\2\t\3\n/'
        raw-data.txt | sed 's/,//g' > cleaned-data.tsv;
    output:
        - cleaned-data.tsv
```

This added `"WARNING"` atrribute may contribute to the documentation quality and improve understanding of the project.

Added non-standard attributes may also be helpful components of a non-sake tool that processes Sakefiles to perform some other function, such as to generate markup.

Sake will issue a warning if it sees an attribute name that it doesn't recognize because it is possible that it was a mispelling of a reserved attribute. If the warning is unwanted, it is possible to suppress it by adding `"(ignore)"` in front of the attribute name, as in this example:

```
format raw data:
    help: format raw (copy and pasted) data using perl
    (ignore) WARNING: the regular expression need to be rewritten...
    ...
```

3.4 Meta-targets

Meta-targets allow one or more targets to be treated as one. This can greatly improve readability and make a Sakefile more managable to maintain. Here's an example:

```
foobar:
    help: does "foo" and "bar"
    foo:
        help: does "foo"
        dependencies:
            - baz.dat
```

```
            - foo.sed
        formula: >
            ./foo.sed baz.dat > foo.dat;
        output:
            - foo.dat
    bar:
        help: does "bar"
        dependencies:
            - baz.dat
            - bar.awk
        formula: >
            ./bar.awk baz.dat > bar.dat;
        output:
            - bar.dat
```

In this example, both atomic targets `"foo"` and `"bar"` use shell scripts to transform the file `"baz.dat"` to `"foo.dat"` and `"bar.dat"`, respectively. These targets are similar in function so it's plausible that it would make sense to group them together, while also keeping them functionally separated so that one may be run independently of the other.

With a Sakefile containing the meta-target above, `sake` may be run on `foobar`, `foo`, or `bar` as in this command:

```
sake foobar
```

This grouping can be readily seen in the output of the `sake help` command:

```
You can 'sake' one of the following...

foobar:
  - does "foo" and "bar"
    foo:
      -  does "foo"
    bar:
      -  does "bar"

clean:
  -  remove all targets' outputs and start from scratch

visual:
  -  output visual representation of project's dependencies
```

A meta-target must have a help field, and at least one target. A meta-target may not contain another meta-target, either.

3.5 Macro definitions

In addition to targets, a Sakefile may also contain "macro definitions".

A macro definition can be used to replace often-repeated paths and files with shorter "nick-names" or aliases. More generally, it replaces text elsewhere in the Sakefile document with the value specified in the macro definition.

To use a macro, you first have to define one, like in these examples:

```
#! A_FILE=areallyreallyreallylongfilename.jpg
#! THE_PATH = /Applications/Xcode.app/Contents/Developer/Library/
#!LS = ls -al
```

The macro definition must start with `"#!"`. Then, the macro name must start with a letter and contain no spaces. Then an equals sign (`=`), and then the macro's value. Anything after the first non-whitespace character after the equals sign and until the end of the line is considered to be part of the macros value, including spaces (as with the `"LS"` example.)

Any amount of spacing is allowed after the `"#!"` and preceding and following the equals sign.

The capitalization of the macro name is not a requirement but it improves readibility by sending clear notice to the human Sakefile reader that a macro is being used.

To actually use a macro in a Sakefile, the macro name must be prefixed by a dollar sign (`"$"`) as with this example:

```
formula: >
    cp ./$A_FILE $THE_PATH
```

One thing to watch out for when using macros in Sakefiles is that it can only be used in a string. This is because the dollar sign is a special character in YAML and is only ignored when it is in what is, unambiguously, a string. For this reason a macro cannot (and shouldn't be) used as a target name unless it is quoted. It may be used anywhere in a formula, though, because the formula is one large string.

If the macro is meant to be included in a larger word, using curly braces can improve readibility (and potentially reduce ambiguity). This example depicts the use of curly braces when using macros:

```
formula: >
    cp ./$A_FILE ${THE_PATH}Frameworks
```

which copies the file `areallyreallyreallylongfilename.jpg` to the directory `/Applications/Xcode.app/Contents/Developer/Library/Frameworks`.

If you want a literal dollar-sign (`"$"`) in the formula, you have to use two dollar-signs (`"$$"`).

3.6 The special "all" target

The last component that a Sakefile may contain is a special target, named `"all"`, that is a list of targets to build when `sake` is run with no other targets named.

Normally, when the `sake` command is run and a target is not passed in as a parameter, sake will build *all* of the targets in the Sakefile. If there is an "all" target specified, though, `sake` will only build the targets that "all" names. This can be helpful to prevent certain targets from running all the time, or if `sake` is run while a target is actively being written.

3.7 Building a project

To properly cover the topic of building a sake project, an example of a full example Sakefile would be helpful for reference, and to help put everything in context:

```
---
# Macros
```

```
#! TEEN_STATS_URL = http://mathforum.org/workshops/sum96/data...

fetch teen stats:
    help: fetches various teen statstics from the web
    # no dependencies
    formula: >
        curl -o teenstats.xls $TEEN_STATS_URL;
    output:
        - teenstats.xls

formatting:
    help: formatting and conversion steps
    convert teen stats to csv:
        help: >
            uses gnumerics ssconvert to convert ugly xls to csv
            and cleans it
        dependencies:
            - teenstats.xls
            - convert.sh
        formula: >
            ./convert.sh;
        output:
            - teenstats.csv
    format dui stats:
        help: format raw (copy and pasted) dui/state data using perl
        dependencies:
            - rawdata.txt
        formula: >
            perl -pe 's/^(\D+)\s+([\d,]+)\s+([\d,]+)\s*/\1\t\2\t\3\n/'
            rawdata.txt | sed 's/,//g' > duistats.tsv;
        output:
            - duistats.tsv

find correlates:
    help: >
        calls R script that finds correlates of DUI arrest in
        various teen statistics
    dependencies:
        - duistats.tsv
        - teenstats.csv
        - dui-correlates.R
    formula: >
        ./dui-correlates.R
    output:
        - Rplots.pdf
        - lmcoeffs.txt
...
```

A short description of each of the steps appears in the "help" field on each entry. Basically, there are two source data files: one exists as raw text (copy and pasted from a website), and the other is fetched from the web using `curl`. The former is cleaned and formatted using `perl` and `sed`; the latter has to go through a process that converts downloaded excel file into a CSV and strips useless lines. Both of these source data files then get read by an R script which, ultimately, outputs a corrogram graphic and a summarization table.

The problem occurs when the file `"rawdata.txt"` is changed or updated, perhaps by another team member. One of two things can happen:

- The maintainer of the project can remain wholly unaware of the change to `"rawdata.txt"` and carry around `"Rplots.pdf"` and `"lmcoeffs.txt"` thinking that they are the most up-to-date reflection of the data available.
- The maintainer knows that the file has changed and has to either re-run all of the steps in the workflow, or dig into zer memory and remember that `"fetch teen stats"` needn't be re-run, but that `"format dui stats"` *and* `"find correlates"` have to be re-run, and in that order.

Admittedly, this is not a very complex workflow and is fairly easy to grasp all at once. But imagine if the workflow involved hundreds of files, with intricate inter-connectivity; it is unreasonable to expect that someone can keep track of which components have to be updated. The only recourse available is to re-run *all* the components. In modern science and data analysis, this can easily take hours or days.

Like stated in the introduction, sake seeks to solve this problem by keeping track of the dependency structure for you. After the first build, any subsequent call to `sake` will only result in the building of the components reliant on changed files.

The first time the Sakefile above is run, the output looks like this:

```
$ sake -q
Running target format dui stats
Running target fetch teen stats
Running target convert teen stats to csv
Running target find correlates
Done
```

(we used the `"-q"` flag to cut down on the volume of the output for this example)

If `sake` is called again, the output looks like this:

```
$ sake -q
Running target fetch teen stats
Done
```

Sake knows that no dependencies have changed, and that the only target that needs to run again is `"fetch teen stats"` (because it has no dependencies). Had the excel file that `fetch teen stats` downloads from the net been modified in any way, sake would detect this and re-run the appropriate targets, and *only* those targets. If the change to `"teenstats.xls"` was trivial and had no bearing on the output `"teenstats.csv"`, sake would be smart enough to know not to re-run `find correlates` because it would not change.

3.7.1 Building specific parts of a project

There are two ways to tell `sake` to build only certain parts of a project:

1. Add an `"all"` target that lists only the targets that you wish to build.
2. Supply the name of the target you'd like to build to `sake` as a parameter. You can see of full list of the targets you can build with the `sake help` command. If you have spaces in your target name, be sure to put the target name in quotes so the shell doesn't interpret the words as separate arguments, like

```
sake "find correlates"
```

It is important to note that even if they haven't been named, sake will *not* ignore targets if they are dependencies of the target that was asked to be built. Only the targets that don't have anything to do with the target that was asked to be built will be ignored.

In some project configurations, if a specific target is named, sake will require that another target/targets run as well. This occurs when two targets share dependencies and building one without the other will break project integrity.

3.7.2 Debugging a Sakefile

When writing and testing any Sakefile for a project of sufficient complexity, you may come across problems. If it isn't a bug in sake, it may be helpful to diagnose the problem by running sake in verbose mode. This will list all the steps that sake took to execute and explain reasoning for running certain targets and not running others.

```
sake -v
```

Verbose mode's logical inverse is "quiet mode" which lists only the targets that are being built as they are executed. With this option, the commands in the formulas, and the output of those commands are suppressed.

It is important to note that errors are never suppressed, even in quiet mode.

3.7.3 Choosing from different Sakefiles

In certain circumstances, it may be helpful for a project to have more than one Sakefile, perhaps with instructions for building on different platforms. As stated above, `sake` will look for a file called `"Sakefile"`, `"Sakefile.yaml"`, `"Sakefile.yml"` (in that order) and uses the first one it finds. You can force `sake` to use a specific Sakfile thusly:

```
sake -s yoursakefile
```

The extention need not be ".yaml" or ".yml" in order for `sake` to accept it.

3.8 Parallel building

If two different targets are not dependencies of each other (one does not depend on the other, and vice-versa) that means the two targets can be built simultaneously. On modern computers with multi-core processors, this usually means that the two targets will build on two separate cores. This means that building the two targets will only take as long as the building of the target that takes the longest to complete.

Consider a `sake` project consisting of two targets with no interdependencies and the targets take the same amount of time to build. If the project is built in `parallel` mode, you can expect the time to build the whole project to be cut roughly in half. Similarly, on a machine with 8-cores, and in a project with 8 independent (non-interdependent) targets whose build times are identical, you can expect a parallel project build time of 1/8th of the time it would take to build the project serially (non-parallel).

In real-world projects, the targets are rarely all independent, but running a project in parallel mode may drastically decrease total build-time anyway.

Using `sake`'s parallel mode is very easy; all you have to do is supply the `-p` flag to `sake` on the command-line. `Sake` will automatically determine which targets can be run in parallel and do so.

To make this more concrete, consider this sakefile that builds a four-line poem line-by-line. Each line takes two seconds to output to an intermediate file. After all of the lines are outputted to files, the one-line files are combined in order to product `poem.txt` which contains the whole poem...

```
line by line:
    help: print each line of the poem to a file
    first line:
        help: prints the first line
        dependencies:
        formula: >
            sleep 5;
            echo Twinkle twinkle little bat > first.txt;
        output:
            - first.txt

    second line:
        help: prints the second line
        formula: >
            sleep 5;
            echo How I wonder what youre at > second.txt
        output:
            - second.txt

    third line:
        help: prints the third line
        formula: >
            sleep 5;
            echo Up above the world you fly > third.txt
        output:
            - third.txt

    fourth line:
        help: prints the fourth line
        formula: >
            sleep 5;
            echo Like a tea tray in the sky > fourth.txt
        output:
            - fourth.txt

combine them:
    help: combine all the lines
```

```
dependencies:
    - first.txt
    - second.txt
    - third.txt
    - fourth.txt
formula: >
    cat first.txt second.txt third.txt fourth.txt > poem.txt;
output:
    - poem.txt
```

Building this project by just issuing the `sake` command takes 20.24 seconds on my machine. This is because each target (taking 5 seconds) is run one after another. If the project is build in parallel mode, however...

```
sake -p
```

it takes only 5.24 seconds to build.

The example project above is a bit of an unatural example that was only constructed for pedagogical purposes; it was purposely designed to benefit greatly from parallel building. Nevertheless, in many real-world projects, using parallel mode can save an enormous amount of time.

3.9 Other Options

3.9.1 "Recon" mode

When coming back to building a project after changing a few files, particularly if it is a large project, it can also be helpful to know which targets sake will build/rebuild (and in what order) *before* you actually build them. This can be helpful for planning purposes as well as for debugging.

Sake allows you to do just this with `recon` mode. To use it, just supply the `-r` argument to the sake command. Instead of executing the commands in each target's formula, it will just print the name of the targets it *would* build. For example, after running `sake clean` on the Sakefile introduced in the previous section, running sake in recon mode will look like this:

```
# sake -r
Would run target: third line
Would run target: fourth line
Would run target: first line
Would run target: second line
Would run target: combine them
```

You can also combine parallel mode and recon mode to learn how sake will approach the parallel building:

```
# sake -r -p
Would run targets 'second line, first line, fourth line, third line' in parallel
Would run target 'combine them'
```

In a project that has already been built before, after changing a couple of files, `recon` mode can be an invaluable tool to investigate the impact of your changes.

You should be aware that recon mode cannot *a priori* tell all the changes that will happen down the road, though. For example, if `recon` mode indicates that a certain target will be re-run, if that target will change any output files that are dependencies of other targets, sake cannot know that and recon mode will not detect it.

3.9.2 Forcing building

If `sake` doesn't build a target, it is because none of its dependencies have changed. However, if you want `sake` to rebuild the target anyway, the best way to do this is to use `force` mode by supplying the `-F` command-line flag. This can be applied to just one target (or meta-target) or the whole project. If `force` mode is used to rebuild the whole project, it is roughly equivalent to running `sake clean` and then `sake` again.

3.9.3 Colored output

You can get `sake` to output success messages in green, warnings in yellow, and errors in red by supplying `sake` with the `-c` flag:

```
# sake -c
```

3.9.4 Enhanced formula error handling

A potential Sake "gotcha" in previous versions: if you had several commands in your `formula` separated by semicolons or literal new-lines, it was possible for one of the earlier commands to fail and for Sake to still think the command succeeded. Specifically, if the last command in a formula was successful, it'd appear to Sake as if the whole formula succeeded. This (usually) unsavory behavior could be thwarted (in some shells) by separating the commands not by semicolons (or literal new-lines) but with `&&`s:

```
command_1 && command_2
```

Doing this, the shell will only execute the second command if the first one returned a non-zero exit status.

Because terminating a series of commands in a formula at the first sign of error is such a common need–and because having Sake recognize any failed command as an "unsuccessful" formula build is usually very important–recent versions (starting at version 0.9.7) by default use a mode called "enhanced errors" (which, unfortunately, only works for POSIX (non-windows) systems. What this does is add the `-e` option to the shell that `sake` executes it's formulas in.

This behavior can be turned off by using the `-E` option:

```
# sake -E
```

3.9.5 Custom shells

By default, Python chooses the shell that will run your formulas. On Windows, it will chose `cmd.exe` and on POSIX systems, it will run the POSIX shell (`/bin/sh`. If you have constructs in your formulas (named pipes, subprocess redirection, etc...) that requires another shell, you can specify the alternative shell to use in your Sakefile:

```
shell: bash
```

3.10 Visualization

The last basic topic to cover is sake's visualization capabilities.

Since sake already knows which targets are dependent on others (and why), it was a small jump to enlist the help of the powerful graph visualization software, Graphviz, to be able to visualize these dependency structures.

With Graphviz properly installed, creating a visualization of your workflow is as easy as running the command

```
sake visual
```

This will, by default, deposit a file called `"dependencies.svg"` into your current working directory. This file can then be opened (and printed) from an image viewer or a web browser. The SVG file format is a good choice for graph visualizations because it can be scaled arbitrarily large without pixelation.

Internally, the visualization feature follows these steps:

1. The dependency graph that sake stores internally is written to a temporary text file, in a format that Graphviz understands.
2. The Graphviz executable (called `"dot"`) is called to convert the text file that describes the graph into a image.
3. The temporary file is deleted.

If you'd like to name the resulting visualization something other than "dependencies.svg", you can call `sake visual` with the `-f` flag:

```
sake visual -f anothername
```

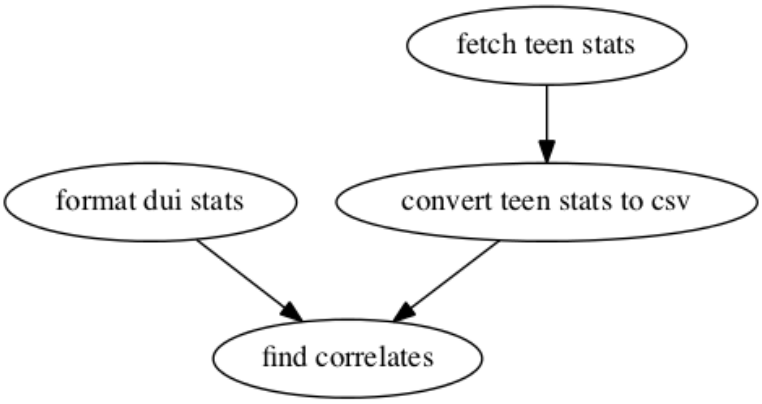
This will name the file `"anothername.svg"` (the file extention is automatically appended).

If another image format is desired, specify a filename with the `-f` flag and include a file extension. For example, to get a jpeg representation of the dependency network, use:

```
sake visual -f somename.jpg
```

Sake can handle the following file extentions and image formats: .svg, .jpg (or .jpeg), .png, .gif, .ps, .pdf

The Sakefile in the example above will yield a visualization like this:


fetch teen stats
format dui stats
convert teen stats to csv
find correlates

4 Advanced Sake Usage

4.1 Using Patterns

Sake may have a strong preference for elegance and simplicity, but that doesn't mean Sake is lacking advanced features. One such feature is 'formula patterns'.

Very often in project building, the same procedure has to be performed on many similar files. Formula patterns allow you to condense all of the targets down to one that gets automatically autoexpanded. If you find yourself with targets that have identical formulas except for a file that changes from target to target, this might be a perfect application for formula patterns.

Concretely, it is a common C programming/compilation idiom to compile all C files in a directory into object files (which then get linked together). If there are four such .c files (file1.c, file2.c file3.c, and file4.c) in a directory that all need to be object-compiled, you certainly *can* create four different targets to perform this, but you can also employ formula patterns here.

If the original Sakefile snippet looked like this:

```
compile C file1:
    help: compiles C file 1
    dependencies:
        - "file1.c"
    formula: >
        gcc -c -o file1.o file1.c
    output:
        - file1.o

compile C file2:
    help: compiles C file 2
    ....
```

the new target to replace the four targets above will look like:

```
compile %cfiles:
    help: compiles file %cfiles into object file
    dependencies:
        - "%cfiles.c"
    formula: >
        gcc -c -o %cfiles.o %cfiles.c
    output:
        - "%cfiles"
```

This may appear as but one target in the Sakefile, but as far as Sake is concerned, these are four different targets (or how ever many files the pattern matches–it was just four in our example).

To prove it, `sake help` will yield:

```
"compile file1":
  - compiles file file1 into object file
```

```
"compile file2":
  - compiles file file2 into object file
...
```

Additionally, since these are separate targets, `sake visual` will render them separately, and parallel sake will run them in parallel, if applicable.

Note that anywhere in the target the string `"%cfiles"` is used, the substitution is of the base name of the file, and not the file extentions.

For another example, say you have to convert all png images in a directory to jpeg files; you have a program called "picconvert" that performes the conversion. The Sakefile snippet will look a little like this:

```
convert %picfile:
    dependencies:
        - "%picfile.png"
    formula: >
        picconvert --from png --to jpg %picfile.png %picfile.jpg
    output:
        - "%picfile.jpg"
```

Important: The percent-sign ("%") is not a valid character in YAML unless it is in a string. Because Sakefiles are 100% valid YAML, you need to make sure that `sake` categorically recognizes attributes/values with a percent-sign in them as strings. You can make sure that this happens by wrapping them in quotation marks. Quoting the target name, help string, or formula is usually unecessary, but it is usually necessary to quote the 'dependencies' item and 'output' item.

When using patterns in your Sakefiles, there are four other things to keep in mind

1. The pattern must appear in the dependency field.
2. A target that has a pattern in the dependency must have the pattern in the target name. Because Sake treats every file match as a new target, and different targets cannot share a name, the pattern must be in the target name to get outexpanded and keep the target names different and specific.
3. A target using a pattern must use the pattern in its formula.
4. A target using a pattern must have an output, and must use the pattern in the output

Don't be worried about this requirements; It is hard to think of a legitimate usage of formula patterns that will violate any of the rules above.

4.2 Using Includes

Another one of Sake's advanced features is called "includes".

Includes are a lot like storing macros in a different file and then importing those macro definitions into your Sakefile.

The primary use case for this feature would be to use the same Sakefile across different architectures/platforms but be able to substitute values for platform dependent variables.

For example, let's say a C project decides to support the two most popular open source C compilers, clang and gcc. It would be silly to write two different Sakefiles–one that uses

gcc for compilation, and one that uses clang for compilation–and switch which Sakefile you use dependending on the machine being used.

Instead, a program can be run to interrogate a system, find the name of the extant/preferred C compiler, and drop that into a yaml file as a Sake compatible macro. That very macro can then be included in the Sakefile.

Concretely, let's say a configure script is run and it determines that gcc is the preferred C compiler. It can then create a file called `config.yaml` with the following content:

```
#! C_Compiler=gcc
```

In the Sakefile, we can include a line like this:

```
#< config.yaml
```

This will make the `C_Compiler` macro available for use in the Sakefile. (Note that `"config.yaml"` could have been named anything.)

If you include a non-existent config file, `sake` will throw a fatal error. There may be a use case, however, where you would like to include a macro definition file, but don't want to strictly *require* it. In this case, you can label the include file `optional`, as in this line:

```
#< filetoinclude.yaml optional
```

It is also possible to raise a warning when an optional include file is missing; the user will see this warning everytime `sake` is run until the include file becomes existent. This can be done thusly:

```
#< filetoinclude.yaml or warning: include file missing
```

Everything after the `"or"` will be considered the warning message.

Important: The yaml file that you include is always relative to the top-level project directory. That means that you can include a file in a subdirectory of a project's root by specifying the path relative to that root. It also means that in a project with two or more Sakefiles– where one is in a subdirectory of the project's root–if the one in the subdirectory includes a file "config.yaml", it will look for that file relative to the project's root, **not** in the subdirectory that houses the Sakefile that actually included the file.

4.3 Advanced Use of Macros

Similar to how having macro include files is useful in some settings, so too is it helpful to be able to supply macro definitions on the command-line. For example consider the following *complete* Sakefile:

```
compile hello:
    help: compiles "hello"
    dependencies:
        - hello.c
    formula:
        $CC -o hello $CFLAGS hello.c
    output:
        - hello
```

Note how the macros `CC` and `CFLAGS` are not specified by the Sakefile. If you use Sake to build this, the `$CC` and `CFLAGS` will not be replaced and will go into the formula, verbatim. If CC and CFLAGS are not environment variables in your shell, then the formula will fail.

To specify values for these macros (so that Sake can expand them appropriately) run `sake` using one or more `-D` key/values.

```
# sake -D CC=gcc -D CFLAGS="-Wall -Werror"
```

You can provide default values for these macros by putting this at the top of the Sakefile displayed above:

```
#! CC?=clang
#! CFLAGS?=-Wall -Werror
```

The `?=` construct basically says "if this macro is not defined, set it to this". If you don't include the question mark before the equals sign, the macro definitions on the command line (via `-D`) will *not* override it.

To require that a macro be pre-defined, either via the command-line `-D` option or in an include file, you can use the `or` option:

```
#! CC or Have to specify a C compiler
```

Everything after the `"or"` will be considered the error message to be printed.