Lecture 08: Metaprogramming

"Metaprogramming" here is more about *process* than they are about writing code or working more efficiently. We should note that “metaprogramming” can also mean “programs that operate on programs”, whereas that is not quite the definition we are using for the purposes of this lecture.

Build systems

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.

`make` is one of the most common build systems out there. When you run make, it consults a file called **Makefile** in the current directory.

```
paper.pdf: paper.tex plot-data.png
  pdflatex paper.tex

plot-%.png: %.dat plot.py
  ./plot.py -i $*.dat -o $@
```

Each directive in this file is a **rule** for how to produce the left-hand side using the right-hand side. The things named on the right-hand side are *dependencies*, and the left-hand side is the *target*. The first directive also defines the *default* goal.

If you run make with no arguments, this is the target it will build. Alternatively, you can run something like `make plot-data.png`, and it will build that target instead.

The `%` in a rule is a “pattern”, and will match the same string on the left and on the right. `$@` is a automatic variable that contains the target name.

Makefile Tutorial by Example

This tutorial is based on the topics covered in the GNU Make book. This tutorial teaches mainly through examples in order to help quickly explain the concepts in the book. Makefile Syntax A Makefile consists of a set of rules. A rule generally looks like this: The

https://makefiletutorial.com/

Dependency management

Most projects that other projects depend on issue a *version number* with every release. Usually something like 8.1.3 or 64.1.20192004.

Version numbers serve many purposes, and one of the most important of them is to ensure that software keeps working.

The exact meaning of each one varies between projects, but one relatively common standard is *semantic versioning*. With semantic versioning, every version number is of the form: major.minor.patch. The rules are:

- If a new release does not change the API, increase the patch version.
- If you *add* to your API in a backwards-compatible way, increase the minor version.
- If you change the API in a non-backwards-compatible way, increase the major version.

Now, if my project depends on your project, it should be safe to use the latest release with the same major version as the one I built against when I developed it, as long as its minor version is at least what it was back then. In other words, if I depend on your library at version `1.3.7`, then it should be fine to build it with `1.3.8`, `1.6.1`, or even `1.3.0`. Version `2.2.4` would probably not be okay.

A **lock file** is simply a file that lists the exact version you are currently depending on of each dependency. An extreme version of this kind of dependency locking is **vendoring**, which is where you copy all the code of your dependencies into your own project.

Continuous integration systems

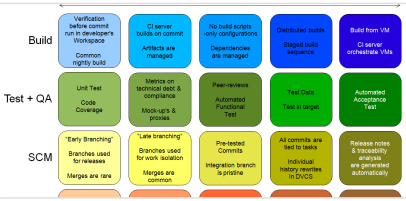
Continuous integration, or **CI**, is an umbrella term for "*stuff that runs whenever your code changes*", and there are many companies out there that provide various types of CI, often for free for open-source projects.

By far the most common one is a rule like "when someone pushes code, run the test suite". When the event triggers, the CI provider spins up a virtual machines (or more), runs the commands in your "recipe", and then usually notes down the results somewhere. You might set it up so that you are notified if the test suite stops passing, or so that a little badge appears on your repository as long as the tests pass.

Continuous Integration Essentials | Codeship

Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day. Each integration can then be verified by an automated build and automated tests. While automated testing is

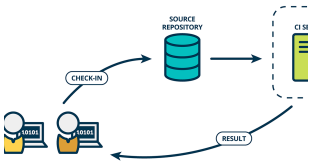
https://codeship.com/continuous-integration-essentials



The Product Managers' Guide to Continuous Delivery and DevOps - Mind the Product

In this guide, I aim to demystify Continuous Delivery and DevOps. I'll explain these practices, tell you just how important they are to you "on the business side" and help you get involved. It's not that complicated, we have pictures and everything. This guide is for you if ...

https://www.mindtheproduct.com/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/



A brief aside on testing

There are some approaches to testing and testing terminology that you may encounter in the wild:

- Test suite: a collective term for all the tests
- Unit test: a “micro-test” that tests a specific feature in isolation
- Integration test: a “macro-test” that runs a larger part of the system to check that different feature or components work *together*.
- Regression test: a test that implements a particular pattern that *previously* caused a bug to ensure that the bug does not resurface.
- Mocking: the replace a function, module, or type with a fake implementation to avoid testing unrelated functionality. For example, you might “mock the network” or “mock the disk”.

Others

Phony Targets (GNU make)

A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request. There are two reasons to use a

https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html

What is the purpose of .PHONY in a makefile?

By default, Makefile targets are "file targets" - they are used to build files from other files. Make assumes its target is a file, and this makes writing Makefiles

https://stackoverflow.com/questions/2145590/what-is-the-purpose-of-phony-in-a-makefile

Standard Targets (GNU make)

All GNU programs should have the following targets in their Makefiles: Compile the entire program. This should be the default target. This target need not rebuild any documentation files; Info files should normally be included in the distribution, and DVI (and other documentation format) files should be made only when explicitly asked for.

https://www.gnu.org/software/make/manual/html_node/Standard-Targets.html#Standard-Targets

Standard targets for make files

The Cargo Book

Your crates can depend on other libraries from crates.io or other registries, git repositories, or subdirectories on your local file system. You can also temporarily override the location of a dependency - for example, to be able to test out a bug fix in the dependency that you are working on locally.

https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html

Specifying Dependencies

Learn how to improve your Git skills

An introductory guide and resource for Git hooks. Learn how to use pre-commit hooks, post-commit hooks, post-receive hooks, and more. Created by Matthew Hudson, a programmer experimenting with combing Git + WebHooks + Webpipes.

https://githooks.com/

Git hooks introduction

Using Python with GitHub Actions

You can create a continuous integration (CI) workflow to build and test your Python project. This guide shows you how to build, test, and publish a Python package. GitHub-hosted runners have a tools cache with pre-installed software, which includes Python and PyPy. You don't have to install anything!

https://help.github.com/en/actions/language-and-framework-guides/using-python-with-github-actions