Buildtools and Packaging

Lets never manually compile things, eh?

Jo Hallett

October 29, 2025

University of BRISTOL

Format shifting

So much of what we do with a computer is about format shifting

- Convert these JPEGs to PNGs
- Summarise this spreadsheets worth of data
- Build an AI model from these observations
- **Convert this code into a binary**

Convert this code into a binary

For programmers this is a really common one
This lecture we're going to be talking about tools to do this:

- in general (`make`)
- for Java code specifically (`maven`)

The key point I want you to take away:

- You shouldn't have to remember a bunch of random shell commands to compile your code
- You should just. type. *`make`*.
- (and it should be somewhat smart about it)

Confessions of a rabbid fan…

I am scarily obsessed with `make`

- I will actively abuse it to do more than it was ever designed to do
- I have written non-trivial programs in it to do horrific things
- Take the bits that are useful to you `;-)`

Lets work from an example...

I have some C code.

- `greeter.c` which is a program for greeting people (`main()` is here)
- `library.c` which is a set of library code used by `greeter.c`
- `library.h` which is the library's header file

I want to build these into a single program

- How do I do that?

First attempt

```
$ gcc -c library.c
$ gcc greeter.c library.o -o greeter
```

Could stick into a shellscript...

This will work...

- But it sucks

Why does this suck?

```
$ gcc -c library.c
$ gcc greeter.c library.o -o greeter
```

What if `library.c` fails to compile?

The compilation will continue anyway…

```
$ gcc -c library.c || exit $?
$ gcc greeter.c library.o -o greeter
```

This also sucks (unless you like Go…)

What if we need to use Clang or a set of optimizations

Could type them all out but that adds duplication...

```
CC=clang
CFLAGS=-O2
$CC $CFLAGS -c library.c
$CC $CFLAGS greeter.c library.o -o greeter
```

What if we update `greeter.c` only?

It will still recompile `library.o`

- Seems wasteful

```
if [ ! library.o -nt library.c ]; then
  gcc -c library.c
fi

if [ ! greeter -nt greeter.c -a ! library.o -nt greeter.c ]; then
  gcc greeter.c library.o -o greeter
fi
```

This sucks.

And if you put them all together...

```
CC=clang
CFLAGS=-O2
if [ ! greeter -nt greeter.c -a ! library.o -nt library.c ]; then
  $CC $CFLAGS -c library.c || exit $?
fi

if [ ! library.o -nt greeter.c]; then
  $CC $CFLAGS greeter.c library.o -o greeter
fi
```

And then you add two more libraries (one of whom `library.c` depends on)

```
CC=clang
CFLAGS=-O2

if [ !library1.o -nt library1.c ]; then
  $CC $CFLAGS -c library1.c || exit $?
fi

if [ ! library2.o -nt library2.c ]; then
  $CC $CFLAGS -c library2.c || exit $?
if

if [ ! library.o -nt library.c -a ! library2.o -nt library.c ]; then
  $CC $CFLAGS -c library.c library2.o || exit $?
fi

if [ ! greeter -nt greeter.c -a ! library.o -nt greeter.c -a ! library1.o -nt greeter.c ]; then
  $CC $CFLAGS greeter.c library.o library1.o -o greeter
fi
```

And now I want it to run in parallel...

```
CC=clang
CFLAGS=-O2

if [ !library1.o -nt library1.c ]; then
  ($CC $CFLAGS -c library1.c || exit $?) &
fi

if [ ! library2.o -nt library2.c ]; then
  ($CC $CFLAGS -c library2.c || exit $?) &
if

wait

if [ ! library.o -nt library.c -a ! library2.o -nt library.c ]; then
  ($CC $CFLAGS -c library.c library2.o || exit $?) &
fi

wait

if [ ! greeter -nt greeter.c -a ! library.o -nt greeter.c -a ! library1.o -nt greeter.c ]; then
  $CC $CFLAGS greeter.c library.o library1.o -o greeter
fi
```

Oh and I'm guessing that is broken...

I suspect that the `|| exit $?` won't do what I want and I need to send a semafore and handle that now.

- I'm not going to test this
- It's a hypothetical example
- Either way this is gross and it sucks and its repetitious and I HATE IT

Luckilly we have better tools

`make` is a tool for shifting files between formats

- It fixes all the bugbears with the shell version automatically
- It dates back to the dawn of computers
- If you see a `Makefile` you compile it by typing `make`

Unfortunately, since it is so old…

- There are competing implementations
  - POSIX Make is the standard one (BSD/Macs use versions of this)
  - GNU Make is the one everyone uses (Linux uses this… BSD/Mac users install gnumake and call it as `gmake`)
- The syntax is a bit weird

What does it look like?

Create a file called `Makefile` (or `GNUmakefile`)

```
CC=gcc
CFLAGS=-O2

.default: greeter

greeter: greeter.c library.o library1.o
    $CC $CFLAGS greeter.c library.o library1.o

library.o: library.c library2.o
    $CC $CFLAGS -c library.c library2.o -o library.o

library1.o: library1.c
    $CC $CFLAGS -c library1.c -o library1.o

library.2: library2.c
    $CC $CFLAGS -c library2.c -o library2.o
```

And what does that mean?

```
greeter: greeter.c library.o library1.o
	$CC $CFLAGS greeter.c library.o library1.o
```

To build `greeter` you'll need the files:

- `greeter.c`, `library.o` and `library1.o`
- Rebuild any of them if their source is newer
- And rebuild `greeter` if any of these are newer

To do the build you need to run

- `$CC $CFLAGS greeter.c library.o library1.o`
- This line is indented with a **tab!**

And the CC bit?

```
CC=gcc
CFLAGS=-O2
```

Set variables to these values

- No spaces around the =

.default?

```
.default greeter
```

If you run `make` with no arguments…

- It'll run the first rule by default
- or it'll run the one you mark as `.default`
  - Explicit beats implicit!

And if you run `make`

It will build the default rule

- or the first one if not specified

It will see if theres a way to make all the dependencies

- And check if it needs to build them at all

It will do it in parallel

- …if you use `-j4` (or however many processes you want)

Can we do better?

If you look at our rules for building object files there's a pattern

- To build something `.o` you compile something `.c` (and all its other dependencies) with the `-c` flag

Can we abstract this?

```
%.o: %.c
   $CC $CFLAGS -c $^ -o $@
```

`%.o: %.c` To get /something/=.o= you need a /something/=.c=

`$^` The entire dependency list

`$@` The target output

Patternrule GNUmakefile

```
CC=gcc
CFLAGS=-O2

.default: greeter

%.o: %.c
    $CC $CFLAGS -c $^ -o $@

%: %.c
    $CC $CFLAGS -$^ -o $@

greeter: greeter.c library.o library1.o
library.o: library.c library2.o
```

But make is old

It has builtin rules for C (C++ and Pascal and a few others)...

```
CC=gcc
CFLAGS=-O2
.default: greeter
greeter: greeter.c library.o library1.o
library.o: library.c library2.o
```

Thats make, folks!

That is 90% of everything you'll ever need with `make`.

- But I said this is my favourite tool
- I should show you some more advanced tricks

Some more general good practices

- You should add a rule called `all` that builds everything
- You should add a rule called `install` that installs your into `$PREFIX/bin`
- You should add a rule called `clean` that removes all build artefacts
- You should declare targets that build things that aren't output files as `.phony`

```
CC=gcc
CFLAGS=-O2
.default: all
.phony: all clean install
all: greeter

clean:
   $RM -rf $(git ls-files --others --exclude-standard)

install: greeter
   install -m 0755 -o root -g root -s greeter "$PREFIX/greeter"

greeter: greeter.c library.o library1.o
library.o: library.c library2.o
```

Bonus tricks

We set `CC` to be `gcc`... but what if the user wants to override it?

- What if they want to do a build with `-O3` or `-g` in their `CFLAGS`?

```
CC?=gcc
CFLAGS?=-O2
```

Now the user can override them with an environment variable

```
$ CC=clang make all
```

What if you don't want to list all your files

Say I'm writing a paper with figures

- If any of my figures change I will need to recompile my paper

```
paper.pdf: paper.tex figures/figure1.png figures/figure2.png figures/figure3.png
   pdflatex paper
```

Seems tedious to keep updating the dependencies as I add figures?

```
paper.pdf: paper.tex $(wildcard figures/*.png)
   pdflatex paper
```

Say I need to convert a bunch of files…

As part of my paper I have a bunch of flowcharts written in GraphViz

- I convert these to PNGs with the `dot` command

```
%.png: %.dot
   dot -Tpng $< -o $@

flowcharts=$(patsubst .dot,.png,$(wildcard figures/*.dot))
paper.pdf: paper.tex $(wildcard figures/*.png) ${flowcharts}
   pdflatex paper
```

Yer what now?

```
%.png: %.dot
   dot -Tpng $< -o $@

flowcharts=$(patsubst .dot,.png,$(wildcard figures/*.dot))
paper.pdf: paper.tex $(wildcard figures/*.png) ${flowcharts}
   pdflatex paper
```

Lets say I have 3 files:

- `figures/diagram.dot`
- `figures/flowchart.dot`
- `figures/chart.dot`
- `$(wildcard figures/*.dot)`
  - Expands to `figures/diagram.dot figures/flowchart.dot figures/chart.dot`
- `$(patsubst .dot,.png,$(wildcard figures/*.dot))`
  - Expands to `figures/diagram.png figures/flowchart.png figures/chart.png`

If `diagram.dot` changes…

- Then `make` will rebuild `diagram.png` and `paper.pdf`

There is more

A whole bunch more!

- Read the manual… its not that bad for a technical document
- Forcing ordering in dependencies
- Forcing randomization of dependencies
- More variables and functions than you can shake a stick at
- If you prefix a command with `@` it doesn't echo it (useful for printing debug messages!)

But there is one thing that `make` doesn't do particularly well…

(Library) Dependencies

Make is really good about knowing how to shift one file to another

- But it doesn't know anything about the code its compiling
- It's just pattern matching on extensions and access times

Modern languages have libraries

- We don't normally compile everything from scratch anymore
- …usually.
- We'd like our build tools to fetch them automatically

Library-aware buildtools

Every language has their own tooling!

Commonlisp	ASDF and Quicklisp
Go	Gobuild
Haskell	Cabal
Java	Ant, Maven, Gradle…
JavaScript	NPM
Perl	CPAN
Python	Distutils and `requirements.txt`
R	CRAN
Ruby	Gem
Rust	Cargo
LaTeX	CTAN and TeXlive

…and many more.

And they're all different

Very little similarity between any of them.

- You need to learn the ones you use.
- We'll play in the labs with Maven for Java a little bit

Maven

Build tool for Java (mostly)

- Others exist (`gradle` and `ant`)
- Configured in XML
- Fairly standard and available everywhere
- Needlessly verbose

(I dislike it and generally use Make and manage things myself but YMMV…)

Lets create a new project

```
$ mkdir /tmp/src
$ cd /tmp/src
$ mvn archetype:generate \
     -DgroupId=uk.ac.bristol.cs \
     -DartifactId=hello \
     -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false
```

(Plus a lot of downloads I've omitted...)

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] >>> archetype:3.2.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< archetype:3.2.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO]
[INFO] --- archetype:3.2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: basedir, Value: /tmp/src
[INFO] Parameter: package, Value: uk.ac.bristol.cs
[INFO] Parameter: groupId, Value: uk.ac.bristol.cs
[INFO] Parameter: artifactId, Value: hello
[INFO] Parameter: packageName, Value: uk.ac.bristol.cs
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /tmp/src/hello
[INFO] ----------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ----------------------------------------------------------------------------
```

So whats that done?

```
find . -type f
```

```
./hello/pom.xml
./hello/src/main/java/uk/ac/bristol/cs/App.java
./hello/src/test/java/uk/ac/bristol/cs/AppTest.java
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0␣http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>uk.ac.bristol.cs</groupId>
  <artifactId>hello</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

This is `xml`!

XML primer

Format for writing trees that can be parsed by a computer and a human

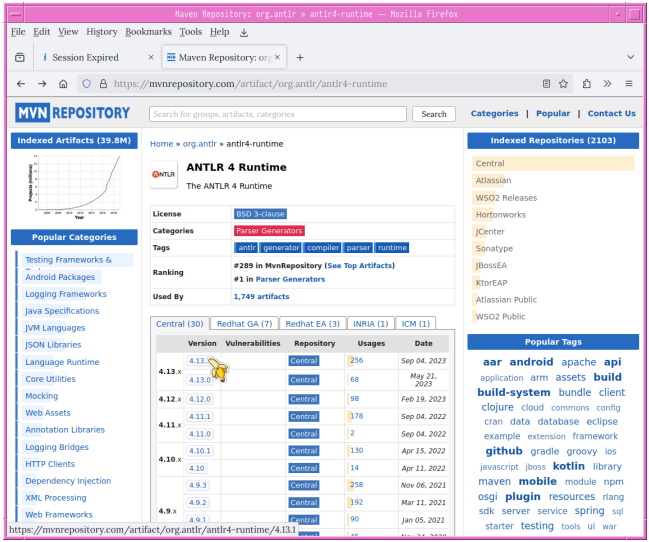
- Basically a generalized form of HTML
- `schema` defines what all the tags mean

```
<!-- This is a comment -->
<tag attribute=value>
  <innerTag>Hello</innerTag>
  <innerTag>World!</innerTag>
</tag>
```

You can't stick stuff wherever… the tags define relationships between what they are and what they contain.

To add a library

If I want to add a library… Go find it on a Maven repository and pick the version you want:



Get the <dependency>...

Maven Repository: org.antlr » antlr4-runtime » 4.13.1 — Mozilla Firefox

File Edit View History Bookmarks Tools Help

Session Expired | Maven Repository: org

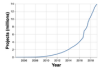
https://mvnrepository.com/artifact/org.antlr/antlr4-runtime/4.13.1

MVN REPOSITORY

Search for groups, artifacts, categories — Search

Categories | Popular | Contact Us

Indexed Artifacts (39.8M)



Popular Categories

- Testing Frameworks & ...
- Android Packages
- Logging Frameworks
- Java Specifications
- JVM Languages
- JSON Libraries
- Language Runtime
- Core Utilities
- Mocking
- Web Assets
- Annotation Libraries
- Logging Bridges
- HTTP Clients
- Dependency Injection
- XML Processing
- Web Frameworks
- I/O Utilities

Home » org.antlr » antlr4-runtime » 4.13.1

ANTLR 4 Runtime » 4.13.1

The ANTLR 4 Runtime

License	BSD 3-clause
Categories	Parser Generators
Tags	antlr generator compiler parser runtime
Date	Sep 04, 2023
Files	pom (3 KB) jar (318 KB) View All
Repositories	Central Mulesoft
Ranking	#289 in MvnRepository (See Top Artifacts) #1 in Parser Generators
Used By	1,749 artifacts

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/org.antlr/antlr4-runtime -->
<dependency>
    <groupId>org.antlr</groupId>
    <artifactId>antlr4-runtime</artifactId>
    <version>4.13.1</version>
</dependency>
```

☑ Include comment with link to declaration

Copied to clipboard!

Indexed Repositories (2103)

- Central
- Atlassian
- WSO2 Releases
- Hortonworks
- JCenter
- Sonatype
- JBossEA
- KtorEAP
- Atlassian Public
- WSO2 Public

Popular Tags

aar **android** apache **api** application arm assets **build** **build-system** bundle client clojure cloud commons config cran data database eclipse example extension framework **github** gradle groovy ios javascript jboss **kotlin** library maven **mobile** module npm osgi **plugin** resources rlang sdk server service spring sql starter testing tools ui war

And add it to the pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0␣http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>uk.ac.bristol.cs</groupId>
  <artifactId>hello</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.antlr/antlr4-runtime -->
    <dependency>
      <groupId>org.antlr</groupId>
      <artifactId>antlr4-runtime</artifactId>
      <version>4.13.1</version>
    </dependency>
  </dependencies>
</project>
```

And when you want to build...

```
mvn package
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------< uk.ac.bristol.cs:hello >-----------------------
[INFO] Building hello 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ hello ---
[WARNING] Using platform encoding (US-ASCII actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /tmp/src/hello/src/main/resources
[INFO]
[INFO] --- compiler:3.13.0:compile (default-compile) @ hello ---
[INFO] Recompiling the module because of changed source code.
[WARNING] File encoding has not been set, using platform encoding US-ASCII, i.e. build is platform dependent!
[INFO] Compiling 1 source file with javac [debug target 1.8] to target/classes
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ hello ---
[WARNING] Using platform encoding (US-ASCII actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /tmp/src/hello/src/test/resources
[INFO]
[INFO] --- compiler:3.13.0:testCompile (default-testCompile) @ hello ---
[INFO] Recompiling the module because of changed dependency.
[WARNING] File encoding has not been set, using platform encoding US-ASCII, i.e. build is platform dependent!
[INFO] Compiling 1 source file with javac [debug target 1.8] to target/test-classes
[INFO]
[INFO] --- surefire:3.2.5:test (default-test) @ hello ---
[INFO] Using auto detected provider org.apache.maven.surefire.junit.JUnit3Provider
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running uk.ac.bristol.cs.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s -- in uk.ac.bristol.cs.AppTest
[INFO]
[INFO] Results:
```

Other useful commands

`mvn test` run the test suite

`mvn install` install the JAR into your local JAR packages

`mvn clean` delete everything

But these are defined by whatever the archetype you chose at the beginning were.

You can define your own rules…

- But it isn't as elegant or as easy as Make
- Not unusual to find a `Makefile` that calls a whole other build system

```
all:
	mvn build

install:
	mvn install
```

Moving on

Okay thats Maven... bit verbose but it's basically fine.
What about other languages?

- What about python?

Python??!

Python isn't compiled? Why do you need to build it?

- They're just shellscripts fundamentally
- But large ones are spread over multiple files
- Scripts still need to be installed
- Native code still needs to be compiled

Well… it **usually** is compiled somewhat, its just the default implementation hides a lot of the details and runs it in a VM. If you compile Python with something like `pypy` it can be seriously fast (but sometimes isn't because of the silly way it implemented concurrency back in version 1.0 but still hadn't quite removed until the very latest version 3.14 released in October 2025!).

Pip

Pip is Python's tool for dealing with packages. It is usually installed with python.

- If it isn't `python -m ensurepip`
  - If that gives errors `python3 -m ensurepip`

```
$ python -m ensurepip
```

```
Looking in links: /tmp/tmpeemgjzm0
Requirement already satisfied: pip in /home/goblin/.local/share/python/lib/python3.13/site-packages (25.2)
```

To install a library

```
$ pip install bs4
```

```
Collecting bs4
Downloading bs4-0.0.2-py2.py3-none-any.whl.metadata (411 bytes)
Collecting beautifulsoup4 (from bs4)
Downloading beautifulsoup4-4.14.2-py3-none-any.whl.metadata (3.8 kB)
Collecting soupsieve>1.2 (from beautifulsoup4->bs4)
Downloading soupsieve-2.8-py3-none-any.whl.metadata (4.6 kB)
Requirement already satisfied: typing-extensions>=4.0.0 in /home/goblin/.local/share/python/lib/python3.13/site
Downloading bs4-0.0.2-py2.py3-none-any.whl (1.2 kB)
Downloading beautifulsoup4-4.14.2-py3-none-any.whl (106 kB)
Downloading soupsieve-2.8-py3-none-any.whl (36 kB)
Installing collected packages: soupsieve, beautifulsoup4, bs4

Successfully installed beautifulsoup4-4.14.2 bs4-0.0.2 soupsieve-2.8
```

If your code needs a lot of libraries

What people sometimes do is write a `README.txt` that tells you what to install…

- Hey go install this library and maybe this one too…

This sucks.

- I don't want to manually install things
- I don't want to work out which versions of things we need

requirements.txt

Normal way of listing dependencies for a Python project.

- Just a list of packages and optionally versions

```
pytest
pytest-cov
beautifulsoup4

# Comments and versions?! Wow!
docopt == 0.6.1
```

Then run:

```
$ pip install -r requirements.txt
```

To install everything

Two problems…

You can only have one version of a library installed

So if a `requirements.txt` depends on a library at a specific version you may get errors.

```
ERROR: pip's dependency resolver does not currently take into
account all the packages that are installed. This behaviour is the
source of the following dependency conflicts.

campdown 1.49 requires docopt>=0.6.2, but you have docopt 0.6.1 which is incompatible.
```

You still have to write it yourself

I'm lazy, remember!

- Oh and if one library depends on others you should really list all dependencies…

You're installing libraries in the system Python

So if your OS requires libraries at a specific version that'll break too

- Macs used to be especially bad for this, but all *NIX's suffer from dependency hell.

Virtual Environments

1. Create a new clean Python install just for your project
   - `$ python -m venv ./virtual-env`
   - `$ source ./virtual-env/bin/activate`
2. Install whatever libraries you need
   - `(virtual-env) $ pip install -r requirements.txt`
3. Generate `requirements.txt` with `pip freeze`

```
(virtual-env) $ pip freeze
beautifulsoup4==4.14.2
coverage==7.11.0
docopt==0.6.1
iniconfig==2.3.0
packaging==25.0
pluggy==1.6.0
Pygments==2.19.2
pytest==8.4.2
pytest-cov==7.0.0
soupsieve==2.8
typing_extensions==4.15.0
```

More complex projects

That's essentially fine if your Python project is a one-file script.

- More complex stuff means using a **build** tool
- Many exist… but are run through `pip`.

To build a package

Create directory structure...

```
./
+-- pyproject.toml # Build script
`-- src/
    `-- mypackage/
        |-- code.py # import mypackage.code
        `-- __init__.py
```

Add the following to `pyproject.toml`

```
# Other build systems exist...
[build-system]
requires = ["hatchling >= 1.26"]
build-backend = "hatchling.build"

[project]
name = "mypackage"
version = "0.0.1"
```

Abracadabra...

```
$ python -m build
```

```
* Creating isolated environment: venv+pip...
* Installing packages in isolated environment:
  - hatchling >= 1.26
* Getting build dependencies for sdist...
* Building sdist...
* Building wheel from sdist
* Creating isolated environment: venv+pip...
* Installing packages in isolated environment:
  - hatchling >= 1.26
* Getting build dependencies for wheel...
* Building wheel...
Successfully built mypackage-0.0.1.tar.gz and mypackage-0.0.1-py2.py3-none-any.whl
```

```
$ pip install ./dist/mypackage-0.0.1-py2.py3-none-any.whl
```

So we've done...

- Building generic software with `make`
- Building Java software with `maven`
- Building Python packages and dependency management with `pip`

What about when we `apt install` something in a Debian box?

Debian packaging format

Debian uses the `.deb` format for packages

- An awful lot of other Linux distros also use it too (Ubuntu, Mint, anything Debian-based)
- (The other format is `.rpm` which is completely different!)

`.deb` files let us build, install and distribute software at a **system** level

- So how do we make them?

Basic steps

1. Tell the system what you need to build your code
2. Tell the system how to build your code (`make`!)
3. Tell the system what you need to run your code
4. Build (with any special patches) and install the software into a **fakeroot**
5. Build the package from the fakeroot and the metadata

Debian specifics

For Debian the tool to build a package is `debuild`

Inside a `debian/` folder you place the build instructions

`debian/changelog` Whats changed with versions of the package (see `man dch`)

`debian/copyright` What license is your code

`debian/control` All the metadata about your package

`debian/rules` A `Makefile` saying how to build your package

`debian/source/format` What style of build are you doing (Debian has many)

Your aim is to write `debian/rules` to install your program, its libraries and docs into the fakeroot at `debian/package-name/`

And when you run `debuild`...

...it spews a **lot** of errors, warnings, and information.

- Building packages is finicky
- But you'll get a `.deb` file in the directory above with your package
- I've stuck an example in the lecture folder
  - But its probably a bit broken...

One last thing...

Wouldn't it be nice if you could have a package or the code built every time you commited your code?

- And for it to fail if your code won't build?

Git hooks

Stick a shellscript in `.git/hooks/pre-commit` to run `make` (or anything else)

- Neat huh?

Wrap up

I love `make`.
Maven is really verbose isn't it?
Everything else is overly complex shellscripts.