Shell Scripting for Font Builds

The basics of making font build systems that are approachable, scalable, and repeatable

@ArrowType

TypeLab 2021

github.com/arrowtype/typelab-2021

DISCLAIMERS

1. This talk is Mac-specific
2. This is just one approach, mostly for .glyphs / .ufo
3. This is also about Python
4. I’m still learning!

DISCLAIMERS

fonts used today

Recursive

https://recursive.design

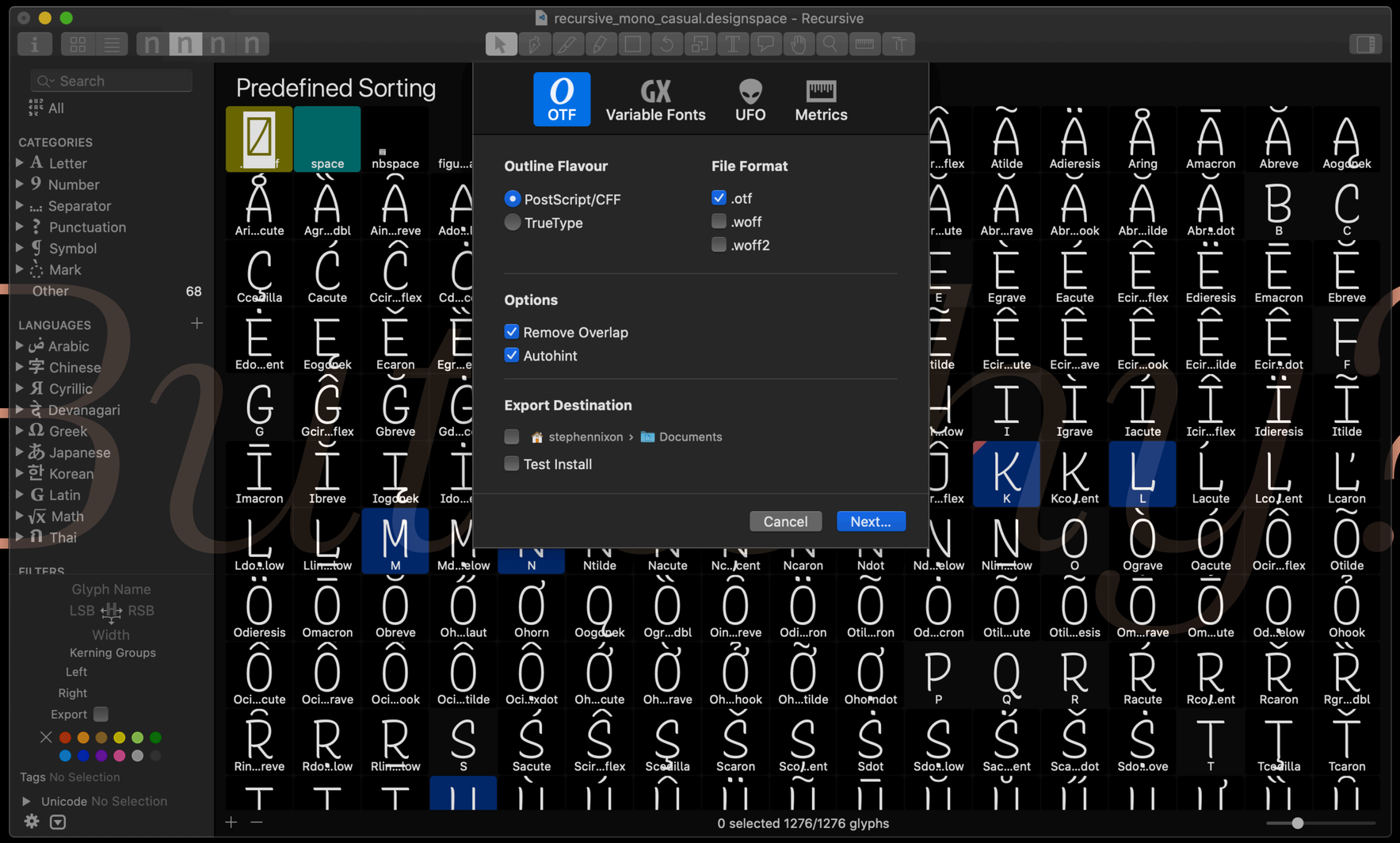
Name Sans

v0.6 on Future Fonts!

Lang Syne

v0.1 coming soon!

fonts used today



Glyphs, RoboFont, FontLab, and other editors are awesome. Why not just use their export tools?

INDEPENDENCE

* Of course, this font building workflow still depends on an open-source ecosystem.

‡ In a greater sense, we all actually are extremely dependent on one another in a society for safety, food, roads, education, healthcare, and on and on. No man is an island, and it is obtuse to claim otherwise. Please pay your taxes and vote for improved social safety nets. Thank you.

Building fonts with code is…

→ **Customizable:** you have fuller control over what you make

→ **Repeatable:** there is less to remember, plus less clicking & dragging

→ **Durable:** open-source build tools are future-proof (mostly)

→ **Debuggable:** you can dig into underlying code to solve problems

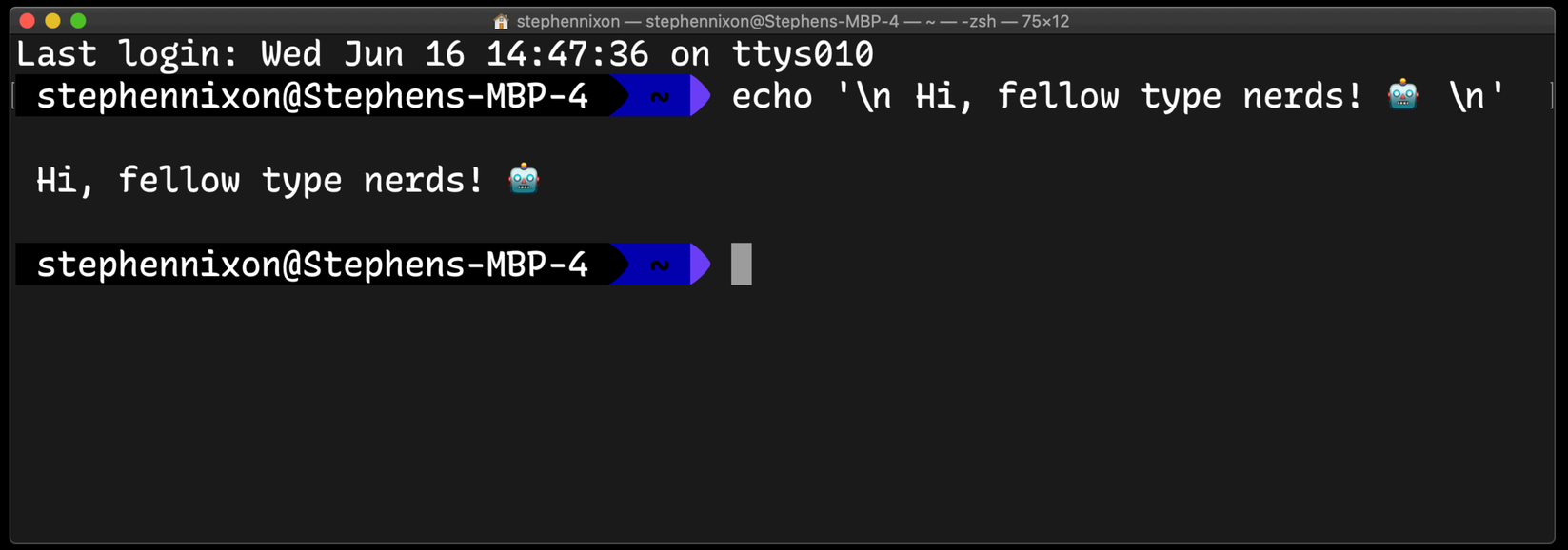
Some useful *definitions*

Font Building

→ The process of taking type sources (.ufo, .glyphs, .vfb, etc) and making them into working font files (.ttf, .otf, .woff2, etc).

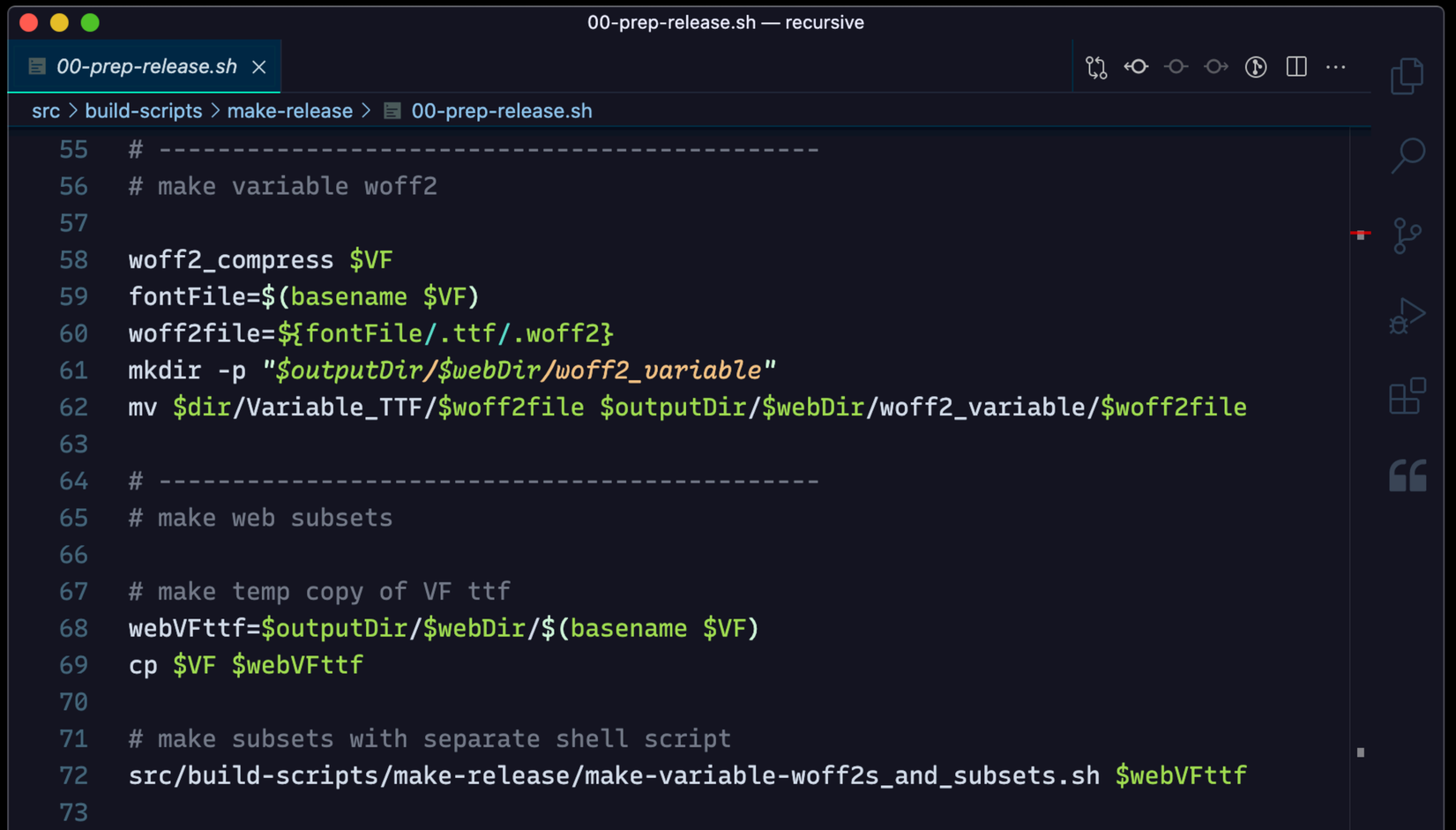
Terminal / Shell / Command Line

→ A tool that lets you control a computer with code



Shell Scripts

→ Scripts that allow you to program a series of shell commands



Why use shell scripting?

Shell scripting is…

→ **Well-Supported:** lots of font dev tools have Command-Line Interfaces

→ **Helpful:** you *could* remember CLI commands, but you don't have to

→ **Powerful:** you can sequence many tools & steps in a font build, easily

→ **Concise:** a shell script can coordinate CLIs, Python, and other code

What’s in a build?

A typical build workflow might include...

→ **Prep:** take working source UFOs and set info, remove draft glyphs, etc

→ **Build:** build sources into static/variable fonts, fix font data in post

→ **Organize:** sort outputs into a custom folder structure, copy in docs

→ **Test:** run FontBakery to check for errors in font data

→ **Proof:** make PDF specimens, web tests with Python, and more

Here's my typical approach

Python	**Prep:** take working source UFOs and set info, remove draft glyphs, etc
Shell Script(s)	**Build:** build sources into static/variable fonts, fix font data in post **Organize:** sort outputs into a custom folder structure, copy in docs
Terminal	**Test:** run FontBakery to check for errors in font data
Desktop, Web, DrawBot, etc	**Proof:** make PDF specimens, web tests with Python, and more

A few *details*

The anatomy of a terminal command

```
cd <dest>
```

Program

e.g. “Change Directory”

Argument(s)

Angle brackets mean “your argument goes here”

Basic Terminal commands

`cd <dest>` – change directory (move location)

`mv <path> <dest>` – move a file to another path

`cp <path> <dest>` – copy a file to another path

`echo <text>` – print text to output

`say <text>` – speak text aloud in a robotic computer voice

Checking a program's manual

`man`

Program

to show the "Manual"

`mv`

Program

MV(1) BSD General Commands Manual MV(1)

NAME

mv -- move files

SYNOPSIS

mv [**-f** | **-i** | **-n**] [**-v**] *source* *target*
mv [**-f** | **-i** | **-n**] [**-v**] *source* ... *directory*

DESCRIPTION

In its first form, the **mv** utility renames the file named by the *source* operand to the destination path named by the *target* operand. This form is assumed when the last operand does not name an already existing directory.

In its second form, **mv** moves each file named by a *source* operand to a destination file in the existing directory named by the *directory* operand. The destination path for each operand is the pathname produced by the concatenation of the last operand, a slash, and the final pathname component of the named file.

The following options are available:

-f Do not prompt for confirmation before overwriting the destination path. (The **-f** option overrides any previous **-i** or **-n** options.)

-i Cause **mv** to write a prompt to standard error before moving a file that would overwrite an existing file. If the response from the standard input begins with the character 'y' or 'Y', the move is attempted. (The **-i** option overrides any previous **-f** or **-n** options.)

-n Do not overwrite an existing file. (The **-n** option overrides any previous **-f** or **-i** options.)

-v Cause **mv** to be verbose, showing files after they are moved.

It is an error for either the *source* operand or the destination path to specify a directory unless both do.

If the destination path does not have a mode which permits writing, **mv** prompts the user for confirmation as specified for the **-i** option.

As the rename(2) call does not work across file systems, **mv** uses cp(1) and rm(1) to accomplish the move. The effect is equivalent to:

rm -f destination_path && \

Checking a CLI's docs

`fontmake`

Program

e.g. FontMake, a program that builds fonts

`--help`

Flag to show docs from a CLI tool

- Most newer programs have a "`--help`" flag, rather than a "Manual"

```
stephennixon — stephennixon@Stephens-MBP-4 — ~ — -zsh — 92×57
Last login: Fri Jun 18 11:01:52 on ttys004

~
▶ fontmake --help
usage: fontmake [-h] [--version]
                (-g GLYPHS | -u UFO [UFO ...] | -m DESIGNSPACE)
                [-o FORMAT [FORMAT ...]]
                [--output-path OUTPUT_PATH | --output-dir OUTPUT_DIR]
                [-i [INSTANCE_NAME]] [--use-mutatormath] [-M]
                [--family-name FAMILY_NAME] [--round-instances]
                [--designspace-path DESIGNSPACE_PATH]
                [--master-dir MASTER_DIR] [--instance-dir INSTANCE_DIR]
                [--no-write-skipexportglyphs] [--validate-ufo]
                [--expand-features-to-instances] [--no-generate-GDEF]
                [--keep-overlaps] [--overlaps-backend BACKEND]
                [--keep-direction] [-e ERROR] [-f] [-a [AUTOHINT]]
                [--cff-round-tolerance FLOAT] [--optimize-cff OPTIMIZE_CFF]
                [--subroutinizer {compreffor,cffsubr}] [--no-optimize-gvar]
                [--filter CLASS] [--interpolate-binary-layout [MASTER_DIR]]
                [--feature-writer CLASS] [--debug-feature-file FILE]
                [--mti-source MTI_SOURCE]
                [--production-names | --no-production-names]
                [--subset | --no-subset] [-s | -S] [--timing]
                [--verbose LEVEL]

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --production-names    Rename glyphs with production names if available
                        otherwise use uninames.
  --no-production-names
  --subset              Subset font using export flags set by glyphsLib
  --no-subset
  -s, --subroutinize    Optimize CFF table using compreffor (default)
                        [DEPRECATED: use --optimize-cff option instead]
  -S, --no-subroutinize

Input arguments:
  The following arguments are mutually exclusive (pick only one):

  -g GLYPHS, --glyphs-path GLYPHS
                        Path to .glyphs source file
  -u UFO [UFO ...], --ufo-paths UFO [UFO ...]
                        One or more paths to UFO files
  -m DESIGNSPACE, --mm-designspace DESIGNSPACE
                        Path to .designspace file
```

Why the difference? `man` vs `--help`

¯_(ツ)_/¯

Flags

```
fontmake --help
```

Program

e.g. FontMake, a program that builds fonts

Flag(s)

- Most programs have a “`--help`” flag
- Flags specify optional arguments
- Many flags have abbreviations, like “`-h`”

A full command

```
fontmake -m "sources/Example.designspace" -o variable
```

Program

Flags with **Arguments**

A very basic shell script

example.sh — typelab-2021

example.sh

```
1  #!/bin/bash
2
3  dsPath="sources/Example.designspace"
4
5  fontmake -m "$dsPath" -o variable --output-path "fonts/Example.ttf"
6
```

A very basic shell script

I could talk
syntax all day,

but...

DEMO TIME

https://github.com/arrowtype/typelab-2021

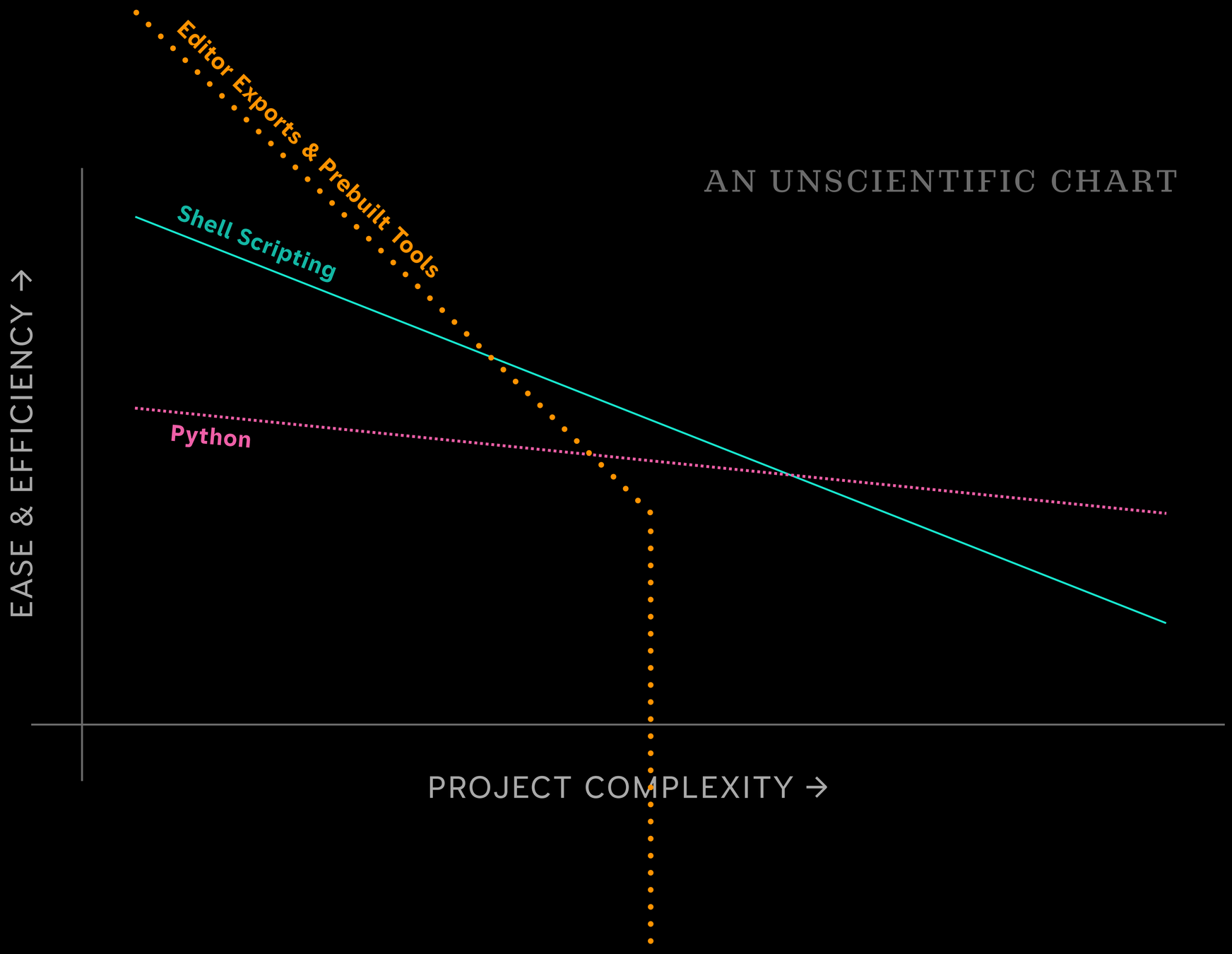
Drawbacks

Compared to Python, shell scripting can be…

→ **Annoying:** syntax can be picky, and some things require Googling

→ **Semi-repetitive:** Python packages are better for repeat-use code

→ **Inflexible:** shell scripts are best when kept concise & high-level


AN UNSCIENTIFIC CHART
Editor Exports & Prebuilt Tools
Shell Scripting
Python
EASE & EFFICIENCY →
PROJECT COMPLEXITY →

At the end of the day...

Shell scripting is
useful & approachable.
It's worth learning!

Where to learn more

How to Create and Use Bash Scripts – By Tania Rascia

A Guide to Python's Virtual Environments – Matthew Sarmiento

Open-source font projects like **Recursive**

Git repos for **FontMake**, **FontBakery**, **woff2**, **GF Tools**, and **FontTools**

github.com/arrowtype/typelab-2021

@ArrowType