Application CLIs

Evan Carlin*, Robert Nagler, Paul Moeller

*evan@radiasoft.net

November 8, 2024

HEPiX, Norman, OK, USA

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Science, Office of Basic Energy Sciences

Introduction

- Evan Carlin
  - Senior software engineer
- RadiaSoft (https://www.radiasoft.net/)
  - Software company in Boulder, CO that specializes in R&D and consulting for high energy physics
- Context for the talk
  - We manage many tools/applications in a variety of bespoke deployments (our systems and clients)
  - Application CLIs are part of what make this possible

Outline

- Application CLIs
  - Define
  - Give motivation for why they are nice to use
- Use case
  - Starting a development server, loading a datasets into a db, etc
- Advantages of Application CLIs
  - One language, availability/distribution/namespacing, dependencies, and abstractions (argument handling, errors, documentation)
- pkcli
  - Overview of a Python framework for writing application CLIs
- Shell scripts still have their place
  - Running many commands, configuring environment
- Prior art
  - Kubernetes, DNF, Django, Ruby on Rails, Laravel, WordPress

Application CLIs

- What is an application CLI?
  - A command line interface to perform some action related to your application (e.g., start your application)
  - Written in the language of your application
  - They live with your application
  - Installed on the system when your application is installed
- Why would I want to use one?
  - Applications need a way to be interacted with on the command line. At the very least to start them.
  - They give you "hooks" into the application which expand the ways your application can be used
  - For libraries it increases the possible uses

Use Cases

- Starting a server (source code)
  - `sirepo service http`
- Loading data into a database (source code)
  - `slactwin db insert_runs`
- Checking for newlines at end of files in CI (source code)
  - `pykern ci check_eof`
- Backing up a GitHub repository (source code)
  - `pykern github backup`
- Checking for expiring TLS certificates (source code)
  - `rsconf letsencrypt check_expiring`
- Helping new users start using a tool (source code)
  - `rsopt quickstart start`
- And many more… (GitHub search)

Many CLIs mentioned at this conference

- HTCondor
- Dask
- Kubernetes
- Puppet
- Ansible
- nfdump
- kubetail
- tmux
- zeek
- elastiflow
- dnf
- brew
- Jupyter
- ALU-Schumacher/auditor
- go-dnscollector
- dnstap
- pdnssoc-cli
- clickhouse

Where do These CLIs Live Without Application CLIS?

- Shell scripts
  - Maybe in a bin/ directory in the repo
  - Maybe in your home directory
  - Maybe in your shell history
  - Maybe on a teammates computer
  - Maybe an administrator installed them for you
- Maybe nowhere at all
  - Without the ability to easily write and share command line scripts sometimes they just are not written

Advantages of Application CLIs

- One language
  - Less cognitive load
  - You get all of your other tooling (formatter, linter, testing, etc) for "free"
- Availability, distribution, and namespacing
  - Without additional infrastructure shell scripts are hard to distribute and can cause naming conflicts when sourced
- Dependencies
  - Shell programming languages do not have a nice way to manage dependencies (ex installing "jq" varies from OS to OS)

Advantages of Application CLIs

- Application CLIs are in the language of your application
  - You can use all of the APIs that already exist in your application
  - You can use the CLI from your application as library code
  - sirepo admin moderate -> GUI
- Abstractions
  - No more parsing arguments by hand in each cli. Arguments in function definitions become arguments/flags for CLIs (A "classic" stack overflow answer to argument parsing in bash)
  - No more log error and exit 1. pkcli.command_error handles it (raises error that is caught instead of killing script)
  - No more duplicated documentation across code and cli. Documentation from code is turned into CLIs

Make it a Framework: pkcli

- pykern (source code)
  - Python application support utilities (logging, common routines, etc)
- projex (source code)
  - A scaffolding tool that sets up the structure of a repo (basic directory structure, pyproject.toml, docs, test)
  - It is a pkcli itself!
- pkcli (source code)
  - The pykern name for application CLIs
  - Provides structure mentioned previously: Namespacing, argument handling, dependencies, etc.
  - We use argh which uses python argparse undernath
  - It uses dispatching to have a centralized hook and call out to CLIs people create

Case Study - pkcli (rslearn) code

```
def train(
    model,
    input_file,
    epochs=1000,
    gpu=0,
    save_dir=None,
    target_file=None,
):
    """Run `model` on `input_file` and save

    Load `input_file`, create a model, compile, fit, and save.

    Args:
        model (str): Name of a function in `rslearn.model_tf`
        input_file ([str, py.path]): numpy loadable file used to train model
        epochs (int): how many epochs to run [1000]
        gpu (int): which gpu number to use [0]
        save_dir (str): directory to save trained model [timestamp-data_npy.purebasename]
        target_file ([str, py.path]): numpy loadable file used for target (fit.y) [input_file]

    Returns:
        py.path: directory where model is stored
    """

    def _compile(shape):
        f = getattr(rslearn.model_tf, model, None)
        # do minimal validation on the function, since this could match any attribute
        if not f or not model.startswith("autoencoder"):
            pykern.pkcli.command_error("model={} not found in rslearn.model_tf", model)
        res = f(shape)
        res.model.compile(loss="mse", optimizer="adam")
        return res
```

*source code

Case Study - pkcli (rslearn) usage

```
~$ rslearn model train --help
usage: rslearn model train [-h] [-e EPOCHS] [-g GPU] [-s SAVE_DIR] [-t TARGET_FILE] model input-file

Run `model` on `input_file` and save

Load `input_file`, create a model, compile, fit, and save.

Args:
    model (str): Name of a function in `rslearn.model_tf`
    input_file ([str, py.path]): numpy loadable file used to train model
    epochs (int): how many epochs to run [1000]
    gpu (int): which gpu number to use [0]
    save_dir (str): directory to save trained model [timestamp-data_npy.purebasename]
    target_file ([str, py.path]): numpy loadable file used for target (fit.y) [input_file]

Returns:
    py.path: directory where model is stored

positional arguments:
  model                 -
  input-file            -

optional arguments:
  -h, --help            show this help message and exit
  -e EPOCHS, --epochs EPOCHS
                        1000
  -g GPU, --gpu GPU     0
  -s SAVE_DIR, --save-dir SAVE_DIR
                        -
  -t TARGET_FILE, --target-file TARGET_FILE
                        -
```

Case study - jupyter to k8s code

```
if __name__ == "__main__":
    argparser = argparse.ArgumentParser()
    argparser.add_argument(
        "--debug",
        action="store_true",
        help="Run helm lint and helm template with the --debug flag",
    )
    argparser.add_argument(
        "--strict",
        action="store_true",
        help="Run helm lint with the --strict flag",
    )
    argparser.add_argument(
        "--values",
        default="lint-and-validate-values.yaml",
        help="Specify Helm values in a YAML file (can specify multiple)",
    )
```

```
def check_call(cmd, **kwargs):
    """Run a subcommand and exit if it fails"""
    try:
        subprocess.check_call(cmd, **kwargs)
    except subprocess.CalledProcessError as e:
        print(
            "`{}` exited with status {}".format(
                " ".join(map(pipes.quote, cmd)),
                e.returncode,
            ),
            file=sys.stderr,
        )
        sys.exit(e.returncode)
```

Case study - jupyter to k8s usage

```
templates (main)$ python lint-and-validate.py --help
usage: lint-and-validate.py [-h] [--debug] [--strict] [--values VALUES] [--output-dir OUTPUT_DIR] [--yamllint-config YAMLLINT_CONFIG]

optional arguments:
  -h, --help            show this help message and exit
  --debug               Run helm lint and helm template with the --debug flag
  --strict              Run helm lint with the --strict flag
  --values VALUES       Specify Helm values in a YAML file (can specify multiple)
  --output-dir OUTPUT_DIR
                        Output directory for the rendered templates. Warning: content in this will be wiped.
  --yamllint-config YAMLLINT_CONFIG
                        Specify a yamllint config
```

- Output is nice (looks like pkcli because we are both using argparse)
- Have to call python. How is the script installed?
- Have to parse the arguments manually (verbose and everyone repeats it)
- log error and sys.exit
- How do I add a new subcommand?

Shell Scripts Still Have Their Place

- Bootstrapping!
- Integrating commands from multiple places (multilingual)
- Configuring environment

Prior Art

- This concept is not new
  - bin scripts in package.json
  - console_scripts in setup.py
  - django-admin in Django
  - Kubernetes and dnf have frameworks written into their code
  - …many more
  - Maybe you have a similar concept you've developed? Please share!

Conclusion

- Every app (and many libraries) need a CLI
  - To be an app you need a way to start the app. To start the app you can use an application CLI
  - Many libraries become more useful with a CLI
- Creating a structure for CLIs provides many advantages
  - One language, availability/distribution/namespacing, dependencies, ability to create abstractions
- I encourage you to give them a try
  - Once developers have easy access to creating application CLIs they make many helpful utilities. No more "bin" directories in projects with a hodgepodge of scripts

Thanks! Questions?

Disclaimer