Python Command Line Tools

Design powerful apps with Click

Noah Gift & Alfredo Deza

Python Command Line Tools

Design powerful apps with Click

Noah Gift and Alfredo Deza

This book is for sale at http://leanpub.com/pythoncli

This version was published on 2020-06-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction **1**

Chapter 1: Getting Started with Click **2**
What is wrong with the alternatives 2
A helpful Hello World 4
Map a function to a command 8

Chapter 2: Test with Click **14**
Test a small click app 14

Chapter 3: Understand IPython **18**
Using IPython, Jupyter, Colab and Python executable 18

Chapter 4: Integrating Linux Commands with Click **29**
Understand subprocess 29
Parsing Results 31
Shell Safety 37

Chapter 5: Writing pure Bash or ZSH command-line tools **39**
Understanding environmental variables 40
Understand shell profiles 43
Write Shell functions 45

Chapter 6: Use Advanced Click Features **48**
Subcommands 49
Utilities 56

Chapter 7: Turbocharging Click **60**
Using The Numba JIT (Just in time Compiler) 60
Running True Multi-Core Multithreaded Python using Numba 64

Chapter 8: Integrate Click with the Cloud **68**
Cloud Developer Workflow 68
Using Cloud-based development environments 70
Build a Computer Vision Tool with AWS Boto3 70

Chapter 9: Case Studies **78**
Distribute a containerized click application to DockerHub 78
Converting a Command-line tool to a Web Service 88
Documenting your Project with Sphinx 89

Chapter 10: Command-line Rosetta Stone **93**
R Hello World 93
Bash Hello World 94
Go Hello World 95
Node Hello World 97
Multi-paradigm Node.js 100
Python Hello World 104

Introduction

Why build complicated a complicated UI, when you can accomplish more with less code? This is the spirit of developing tools for the command-line. In this book you learn to focus on the basics of command-line tools. Ultimately the cli is the best UI. Let's get started.

Chapter 1: Getting Started with Click

Alfredo Deza

Building command-line tools is fun. For me, though, and for the longest time, it meant struggling with the different types of libraries Python had to offer. It all began with `optparse`, which is deprecated since Python 2.7, and it is unfortunately still in the standard library after all these years. It was replaced with `argparse` which, as you will see soon, it is still complicated to deal with. It is crucial to understand what Python has to offer to grasp why the alternative (the Click framework) is better.

Click is a project created by Armin Ronacher[1]. If you haven't heard of him before, you probably have heard about projects he created or co-authored:

- Pygments[2]: Syntax highlighter
- Jinja[3]: Templating engine
- Sphinx[4]: Documentation framework
- Werkzeug[5]: Server and web utility
- Babel[6]: Internationalization library
- MarkupSafe[7]: Safely cleans up HTML strings
- Flask[8]: Web framework
- itsdangerous[9]: Cryptographic library

I've used most of these before, and have had nothing but great experiences with them. What brings us to this chapter is Click[10] and how to get started creating powerful tooling with it.

What is wrong with the alternatives

The standard library offerings (via `optparse` and `argparse`) work fine. From this point forward, I'll briefly touch on `argparse` only since `optparse` is deprecated (I strongly suggest you avoid it). My problem with `argparse` is that it is tough to reason about, and it wasn't created to support some of the use cases the Click supports. One of these examples is sub-commands. Let's build a command-line

[1] https://lucumr.pocoo.org/about/
[2] http://pygments.pocoo.org/
[3] http://jinja.pocoo.org/
[4] http://sphinx.pocoo.org/
[5] http://werkzeug.pocoo.org/
[6] http://babel.pocoo.org/
[7] https://pypi.python.org/pypi/MarkupSafe
[8] http://flask.pocoo.org/
[9] http://pythonhosted.org/itsdangerous
[10] https://click.palletsprojects.com/en/7.x/

```
import argparse

def main():
    parser = argparse.ArgumentParser(
        description="Tool with sub-commands"
    )

    subparsers = parser.add_subparsers(help='Sub-commands')

    sub_parser_1 = subparsers.add_parser('sub1')
    sub_parser_2 = subparsers.add_parser('sub2')

    parser.parse_args()

if __name__ == '__main__':
    main()
```

Crafting the tool requires dealing with a class called `ArgumentParser` and then adding to it. I don't have anything against classes, but this reads convoluted, to say the least. Save it as `sub-commands.py` and run it to see how it behaves:

```
$ python sub-commands.py -h
usage: sub-commands.py [-h] {sub1,sub2} ...

Tool with sub-commands

positional arguments:
  {sub1,sub2}  Sub-commands

optional arguments:
  -h, --help   show this help message and exit
```

What are *"positional arguments"* ? I explicitly followed the documentation to add sub-commands (called `sub_parsers` in the code), why are these referred to as positional arguments? They aren't! These are sub-commands. If you use them as a sub-command in the terminal you can verify this:

```
$ python sub-commands.py sub1 -h
usage: sub-commands.py sub1 [-h]

optional arguments:
  -h, --help  show this help message and exit
```

This is not good. Aside from how complex it is to deal with an instantiated class that grows with sub-commands, the implementation doesn't match what it advertises. In essence, the sub-command is calling `add_subparsers()`, and they display in the help output as *positional arguments* which are sub-commands.

It is entirely possible to change the help output to force it to say something else, but this misses the point. I found the sub-command handling of `argparse` so rough that I ended up creating a small library[11] that attempts to make it a bit easier. Even though that little library is in a few production tools, I would recommend you take a close look to Click, because it does everything that `argparse` does, and makes it simpler - including sub-commands.

Even if you aren't going to use sub-commands at all, the interface to the class and objects that configure the command-line application creates much friction. It has taken me a while to find a suitable alternative, and although Click has been around for quite some time, it hasn't been until recently that I've acknowledged how good it is.

A helpful Hello World

The simplest use case for Click has a caveat: it doesn't handle the help menu in a way that *I prefer* when building command-line tools. Have you ever tried a tool that doesn't allow `-h`? How about it does allow `-h` but doesn't like `--help`? What about one that doesn't like `-h`, `--help`, or `help`?

```
$ find -h
find: illegal option -- h
usage: find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
       find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]
$ find --help
find: illegal option -- -
usage: find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
       find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]
$ find help
find: help: No such file or directory
```

Perhaps I'm raising the bar too high and `find` is too old of a tool? Let's try with some modern ones:

```
$ docker -h
Flag shorthand -h has been deprecated, please use --help
$ python3 help
/usr/local/bin/python3: can't open file 'help': [Errno 2] No such file or directory
```

[11] https://github.com/alfredodeza/tambo

Do you think this level of helpfulness is over the top? Some tools do work with all three of them, like Skopeo[12] and Coverage.py[13]. Not only is it possible to be *that* helpful, but it should also be a priority if you are creating a new tool. As I mentioned, Click doesn't do all three by default, but it doesn't take much effort to get them all working. There are lots of basic examples for Click, and in this section, I'll concentrate on creating one that allows all these help menus to show.

Starting with the most basic example so that we can check what is missing and then go fix it up, save the example as `cli.py`:

```
import click


@click.command()
def main():
    return

if __name__ == '__main__':
    main()
```

This is the first example of a command-line application using Click, and it looks very straightforward. It uses a decorator to go over the `main()` function, and has the piece of magic at the bottom to tell Python it should call the `main()` function if it executes directly. The first time I interacted with a command-line application built with Click, I was surprised to find it so easy to read, compared to `argparse` and other libraries I've interacted with in the past. Nothing should happen when you run this directly, there is no action or information to be displayed, but the help menus are all there ready to be displayed. Run it in the terminal with the three variations of help flags we are looking (`-h`, `--help`, and `help`):

```
$ python cli.py -h
Usage: cli.py [OPTIONS]
Try 'cli.py --help' for help.

Error: no such option: -h
```

No good! It doesn't like `-h`. Try again with `--help`:

[12] https://github.com/containers/skopeo
[13] https://coverage.readthedocs.io/en/coverage-5.1/

```
$ python cli.py --help
Usage: cli.py [OPTIONS]

Options:
  --help  Show this message and exit.
```

Much better. As you can see, the help menu is produced with no effort at all, just by having decorated the `main()` function. Finally, try with `help` and see what happens:

```
$ python cli.py help
Usage: cli.py [OPTIONS]
Try 'cli.py --help' for help.

Error: Got unexpected extra argument (help)
```

It doesn't work, but there is an important distinction: it reported `-h` as an option that is not available and `help` as an unexpected extra argument. Argument vs. options doesn't sound like much, but internally, Click already has an understanding of both. There is no need to configure anything. Now let's make this work the way we envisioned in the beginning.

First, address the problem with `-h` not functioning. To do this, we need to (at last) get into some configuration of the framework. The changes mean that some unique interactions need to happen. In this case, that is working with *the context*. This context is a particular object that keeps options, flags, values, and internal configuration available for any command and sub-command. The framework pokes inside this context to check for any special directives. In this case, we want to change how it deals with the help option names, so define a dictionary that has these expanded:

```
import click

CONTEXT_SETTINGS = dict(help_option_names=['-h', '--help'])

@click.command(context_settings=CONTEXT_SETTINGS)
def main():
    return

if __name__ == '__main__':
    main()
```

The updated example doesn't look that different, except for declaring `CONTEXT_SETTINGS` at the top with a dictionary. That dictionary sets a list onto `help_option_names`. Rerun it with `-h`:

```
$ python cli.py -h
Usage: cli.py [OPTIONS]

Options:
  -h, --help  Show this message and exit.
```

The help menu works, and it displays both flags as available options. This is great and solves the problem with `-h` not being recognized. But what about `help`? Things get tricky here because, without the dashes, it may very well be a sub-command. Declaring `help` as a sub-command is, in fact, part of the solution:

```
import click

CONTEXT_SETTINGS = dict(help_option_names=['-h', '--help'])

@click.group(context_settings=CONTEXT_SETTINGS)
def main():
    return

@main.command()
@click.pass_context
def help(ctx):
    print(ctx.parent.get_help())

if __name__ == '__main__':
    main()
```

It may be tricky to realize at first that there are other differences aside from creating the `help()` function acting as a sub-command. The `@click.command()` decorator gets removed from the `main()` function in favor of `@click.group`. That is how the framework can understand other sub-commands belong to the `main()` function. Another side-effect of this, is that a new decorator is available. The new decorator starts with the name of the function of the group (`main()` in this case is `@main.command()`). Finally, the context is requested with `@click.pass_context` and declared as an argument (`ctx`). This context is what allows the function to print the help menu from the parent command (in `main()`), instead of generating some other help menu. Run this once again to see how it behaves with the new sub-command:

```
$ python cli.py help
Usage: cli.py [OPTIONS] COMMAND [ARGS]...

Options:
  -h, --help  Show this message and exit.

Commands:
  help
```

Good! It now displays the help with any of the three combinations. It is a bit unfortunate that `help` shows in the `Commands:` section. If you are bothered by this, you can hide it from the help by adding an argument to the `help()` function:

```
@main.argument(hidden=True)
@click.pass_context
def help(ctx): #, topic, **kw):
    print(ctx.parent.get_help())
```

Rerun the `cli.py` script to check it out:

```
$ python cli.py help
Usage: cli.py [OPTIONS] COMMAND [ARGS]...

Options:
  -h, --help  Show this message and exit.
```

The downside? `help` is no longer part of the help menu itself, which is a good compromise for being helpful and allowing multiple flags to show the help menu.

Map a function to a command

The previous examples have hinted at how you can start expanding a command-line tool with the Click framework. Let's explore that further by filling all the gaps I skipped over to demonstrate some flags along with the help menu. This section creates a small tool to fix a problem I often encounter when working with SSH[14]: when creating the configuration files and keys, I can't ever get the permissions correctly. The SSH tool requires permissions to be of a certain type for different files; otherwise, it complains that *permissions are too open*. The problem with the error is that it doesn't tell you how to fix it:

[14] https://www.openssh.com/

```
Permissions 0777 for '/Users/alfredo/.ssh/id_rsa' are too open.
It is recommended that your private key files are NOT accessible by others.
This private key will be ignored.
```

There can be multiple different files in the `.ssh/` directory, these are just a few of the conditions that the program needs to ensure:

- The `.ssh/` directory needs to have `700` permission.
- `authorized_keys`, `known_hosts`, `config`, and any public key (ending in `.pub`) needs to have `644` permissions.
- All private keys need to have `600` permission bits.

Create a new file called `ssh-medic.py`, and start with the entry point to define what this tool is, including a good description in the docstring of the function:

```
import click

CONTEXT_SETTINGS = dict(help_option_names=['-h', '--help', 'help'])

@click.group(context_settings=CONTEXT_SETTINGS)
def main():
    """
    Ensure SSH directories and files have correct permissions:

    \b
    $HOME/.ssh      -> 700
    authorized_keys -> 644
    known_hosts     -> 644
    config          -> 644
    *.pub keys      -> 644
    All private key -> 600
    """
    return


if __name__ == '__main__':
    main()
```

The `main()` function's docstring explains what the permission bits[15] should be. It includes a special directive (`\b`), which tells Click not to wrap the lines around my formatting. The wrapping removes the lines breaks and makes my formatting look the same as in the script. Run the script with the help flag to verify the information:

[15] https://en.wikipedia.org/wiki/File_system_permissions

```
$ python ssh-medic.py -h
Usage: ssh-medic.py [OPTIONS] COMMAND [ARGS]...

  Ensure SSH directories and files have correct permissions:

  $HOME/.ssh       -> 700
  authorized_keys  -> 644
  known_hosts      -> 644
  config           -> 644
  *.pub keys       -> 644
  All private keys -> 600

Options:
  -h, --help  Show this message and exit.
```

Next, I want to introduce a separate sub-command that checks the permissions and reports them back in the terminal. This feature is nice because it is a *read-only* command, nothing is going to happen except for some reporting. The `main()` function already is good to go because it is using the `@click.group` decorator, so all is needed is a new function that gets decorated with `@main.command()` to start expanding:

```
@main.command()
def check():
    click.echo('This sub-command innspects files in ~/.ssh and reports back')
```

I'm introducing a new helper from the framework: `click.echo()`, which is replacing `print()`. This helper utility is useful because it doesn't mind Unicode or binary, and understands how to handle different use-cases transparently (unlike the `print()` function). Rerun the tool, using the new sub-command:

```
$ python ssh-medic.py check
This sub-command innspects files in ~/.ssh and reports back
```

The output verifies that the newly added function works as intended, so it is time to do something with it. The `os` and `stat` modules can help us with listing files and checking permissions respectively, so that gets imported and used to expand the `check()` function:

```
import os
import stat

@main.command()
def check():
    ssh_dir = os.path.expanduser('~/.ssh')
    files = [ssh_dir] + os.listdir(ssh_dir)
    for _file in files:
        file_stat = os.stat(os.path.join(ssh_dir, _file))
        permissions = oct(file_stat.st_mode)
        click.echo(f'{permissions[-3:]} -> {_file}')
```

There is quite a bit of newly added code in the function. First, the home directory (as indicated with the tilde) is expanded to a full path, stored to verify permissions later. The script can run from anywhere now. And it reports correctly on the full path rather than just the names of the directories. Then, it creates a `files` list with all the interesting files that are needed, and a loop goes over each one, calling `stat` to get all file metadata. The `st_mode` is the method that provides the information we need, and the function passes the `ssh_dir` joined with the file to produce an absolute path. Finally, the result is converted to an octal form, and reported with `click.echo`. Run it to see how it behaves in your system:

```
$ python ssh-medic.py check
700 -> /Users/alfredo/.ssh
644 -> config
600 -> id_rsa
644 -> authorized_keys
644 -> id_rsa.pub
644 -> known_hosts
```

The permissions on all my files are correct, but I have to correlate what I see with what the permissions should be. Not that useful yet. What is needed here is to perform that check for me instead. Create a mapping of the files and what they should be so that it can report back, and take into account that private keys can be named anything, and public keys may end up with the `.pub` suffix:

```
@main.command()
def check():
    ssh_dir = os.path.expanduser('~/.ssh')
    absolute_path = os.path.abspath(ssh_dir)
    files = [ssh_dir] + os.listdir(ssh_dir)

    # Expected permissions
    public_permissions = '644'
    private_permissions = '600'
    expected = {
        ssh_dir: '700',
        'authorized_keys': '644',
        'known_hosts': '644',
        'config': '644',
        '.ssh': '700',
    }

    for _file in files:
        # Public keys can use the .pub suffix
        if _file.endswith('.pub'):
            expected[_file] = public_permissions

        # Stat the file and get the octal permissions
        file_stat = os.stat(os.path.join(ssh_dir, _file))
        permissions = oct(file_stat.st_mode)[-3:]

        try:
            expected_permissions = expected[_file]
        except KeyError:
            # If the file doesn't exist, consider it as a private key
            expected_permissions = private_permissions
        # Only report if there are unexpected permissions
        if expected_permissions != permissions:
            click.echo(
                f'{_file} has {permissions}, should be {expected_permissions}'
            )
```

The function is now much longer and starting to get into the need for some refactoring. It's OK for now to demonstrate how useful it is. The mapping is now in place and has some fallback values for public and private keys, which can be named almost anything. The loop is somewhat similar to before and checks if the permissions are adhering to the expected values. I've added a few keys with incorrect permissions in my system to test the new functionality:

```
$ python ssh-medic.py check
jenkins_rsa has 664, should be 600
jenkins_rsa.pub has 664, should be 644
```

Nothing reports back if all the files are OK, which is not very helpful. The script has lots of room to improve, like adding a `fix` sub-command that would change the permissions on the fly, and perhaps a dry-run flag that would not change anything but could show what would end up happening. Command-line tools are exciting, and they can grow in features, just make sure that you are keeping functions readable and small enough to test them. Anything that can be extracted into a separate utility for easier testing and maintainability is a positive change.

Chapter 2: Test with Click

Noah Gift

Test a small click app

Writing command-line tools is one of my favorite ways to write code. A command-line tool is one of the simplest ways to take programming logic and create a flexible tool. Let's walk through a simple `click`[16] tool.

The following is an intentionally simple command-line tool that takes a flag `--path` (path to a file) and flag `--ftype` (file type). If this runs without specifying, it will prompt a user for both of these flags.

Simple `click` based cli that searches for files: `gcli.py`

```
#!/usr/bin/env python
import click
import glob

# this is bad code intentionally
# varbad=


@click.command()
@click.option(
    "--path",
    prompt="Path to search for files",
    help="This is the path to search for files: /tmp",
)
@click.option(
    "--ftype", prompt="Pass in the type of file", help="Pass in the file type:  i.e \
csv"
)
def search(path, ftype):
    results = glob.glob(f"{path}/*.{ftype}")
    click.echo(click.style("Found Matches:", fg="red"))
    for result in results:
```

[16] https://click.palletsprojects.com/en/7.x/

```
        click.echo(click.style(f"{result}", bg="blue", fg="white"))


if __name__ == "__main__":
    # pylint: disable=no-value-for-parameter
    search()
```

Another useful feature of `click` is the `--help` flag comes for free and autogenerates from the options. This following is the output from `./glci --help`.

Output of `./gcli --help`

```
(.tip) $  click-testing git:(master) $ ./gcli.py --help
Usage: gcli.py [OPTIONS]

Options:
  --path TEXT   This is the path to search for files: /tmp
  --ftype TEXT  Pass in the file type:  i.e csv
  --help        Show this message and exit.
```

Next up is to run `./gcli.py` and let it prompt us for both the `path` and the `file type`.

Output of search with prompts

```
(.tip) $  click-testing git:(master) $ ./gcli.py
Path to search for files: .
Pass in the type of file: py
Found Matches:
./gcli.py
./test_gcli.py
./hello-click.py
./hello-click2.py
```

Another run of `./gcli.py` shows how it can be run by passing in the flags ahead of time.

Output of search with flags

```
(.tip) $  click-testing git:(master) $ ./gcli.py --path . --ftype py
Found Matches:
./gcli.py
./test_gcli.py
./hello-click.py
./hello-click2.py
```

So can we test this command-line tool? Fortunately, the authors of `click` have an easy solution for this as well[17]. In a nutshell, by using the line `from click.testing import CliRunner` it invokes a test runner as shown.

Test file: `test_gcli.py`

```
from click.testing import CliRunner
from gcli import search


# search(path, ftype):


def test_search():
    runner = CliRunner()
    result = runner.invoke(search, ["--path", ".", "--ftype", "py"])
    assert result.exit_code == 0
    assert ".py" in result.output
```

Like `pytest`, a simple `assert` statement is the only thing needed to create a test. Two different types of `assert` comments show. The first `assert` checks the result of the call to the command line tool and ensures that it returns a shell exit status of `0`. The second assert parses the output returned and ensures that `.py` is in the production since the file type passed in is `py`.

When the command `make test` is run, it generates the following output. Again take note of how using the convention `make test` simplifies the complexity of remembering what exact flags to run. This step can set once and then forget about it.

[17] https://click.palletsprojects.com/en/7.x/testing/?highlight=test

Output of `make test`

```
(.tip) $  click-testing git:(master) make test
python -m pytest -vv --cov=gcli test_gcli.py
======================================== test session starts ================
platform darwin -- Python 3.7.6, pytest-5.3.2, py-1.8.1, pluggy-0.13.1 --
cachedir: .pytest_cache
rootdir: /Users/noahgift/testing-in-python/chapter11/click-testing
plugins: cov-2.8.1
collected 1 item

test_gcli.py::test_search PASSED

---------- coverage: platform darwin, Python 3.7.6-final-0 -----------
Name      Stmts   Miss  Cover
-----------------------------
gcli.py      11      1    91%


========================================= 1 passed in 0.03s ==================
```

Chapter 3: Understand IPython

Noah Gift

Back in 2007 I wrote an article for IBM Developerworks[18] on IPython[19] and SNMP (Simple Network Management Protocol) and this chapter steals some of those ideas. In the physical data center era, the SNMP protocol is a useful way to collect metrics about CPU, memory, and the performance of a particular machine.

What makes IPython, the software engine behind Jupyter Notebook[20] particularly useful is the interactive nature. An object or variable is "declared," then it is "played with." This style is quite helpful.

Here is an example of how that works. A user can import the library `snmp`, declare a variable `oid`, then pass it into the library and get a result. This step dramatically speeds up the ability to build solutions.

```
In [1]: import netsnmp

        In [2]: oid = netsnmp.Varbind('sysDescr')

        In [3]: result = netsnmp.snmpwalk(oid,
        ...:                              Version = 2,
        ...:                              DestHost="localhost",
        ...:                              Community="public")

        In [4]: result
        Out[4]: ('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27
        18:58:54 EDT 2007 i686',)
```

This book is a book about command-line tools, but it is essential to highlight the power of IPython as a useful tool in this endeavor. The `click` library makes it simple to map a function to an instrument. One workflow is to develop the service interactively via IPython, then put it into an editor like Visual Studio Code.

Using IPython, Jupyter, Colab and Python executable

Let's dive into key features of IPython and how they can empower the command-line tool developer.

[18] https://developer.ibm.com/tutorials/au-netsnmpnipython/
[19] https://ipython.org/news.html
[20] https://jupyter.org/

IPython

IPython is very similar to Jupyter, but run from terminal:

* IPython predates Jupyter
* Both Jupyter and IPython accept *!ls -l* format to execute shell commands

To run a shell command, put a `!` in front.

```
!ls -l
```

```
total 4
drwxr-xr-x 1 root root 4096 Nov 21 16:30 sample_data
```

Note that you can assign the results to a variable.

```
var = !ls -l
type(var)
```

The type generated is called an `SList`.

```
IPython.utils.text.SList
```

Another python `method` use you can use from a `SList` is `fields`.

```
#var.fields?
```

Here is an example of a `grep` method. The `SList` object return from using a `!` in front has both a `grep` and a `fields`.

```
var.grep("data")
```

```
['drwxr-xr-x 1 root root 4096 Nov 21 16:30 sample_data']
```

Jupyter Notebook

Going beyond just IPython, there are many flavors of Jupyter Notebook. A few popular ones:

- 

Jupyter

- **JupyterHug**

- **Colab**

- **Kaggle**

- **Sagemaker**

Hosted Commercial Flavors

- Google Colaboratory[21]: Free
- Kaggle[22]: Free

Pure Open Source

- Jupyter[23] standalone, original
- JupyterHub[24] multi-user, docker friendly

[21] https://colab.research.google.com/notebook
[22] https://www.kaggle.com/
[23] http://jupyter.org/
[24] https://github.com/jupyterhub/jupyterhub

Hybrid Solutions

- Running Jupyter on AWS Spot Instances[25]
- Google Data Lab[26]
- Azure Data Science Virtual Machines[27]
- AWS Sagemaker[28]
- Azure Machine Learning Studio[29]

Colab Notebook Key Features

The "flavor" of Jupyter I like the most is colab. Generally, I like it because it "just works."

- 12.1 Perform Colaboratory basics [30]
- 12.2 Use Advanced Colab Features[31]

* Can enable both TPU and GPU runtimes
* Can upload regular Jupyter Notebooks into colab
* Can have a Google Drive Full of Colab Notebooks
* Can sync colab notebooks to Github. Here is an example of a gist of this notebook[32]
* Can connect to a local runtime[33]

- Can create forms in Colab[34]

Magic Commands

Both Jupyter and IPython have "magic" commands. Here are a few examples.

%timeit

```
too_many_decimals = 1.912345897

print("built in Python Round")
%timeit round(too_many_decimals, 2)
```

[25] https://aws.amazon.com/ec2/spot/
[26] https://cloud.google.com/datalab/
[27] https://azure.microsoft.com/en-us/services/virtual-machines/data-science-virtual-machines/
[28] https://aws.amazon.com/sagemaker/
[29] https://studio.azureml.net/
[30] https://www.safaribooksonline.com/videos/essential-machine-learning/9780135261118/9780135261118-EMLA_01_12_01
[31] https://www.safaribooksonline.com/videos/essential-machine-learning/9780135261118/9780135261118-EMLA_01_12_02
[32] https://gist.github.com/noahgift/c69200e05c057cf239fc7ea0be62e043
[33] https://research.google.com/colaboratory/local-runtimes.html
[34] https://colab.research.google.com/notebooks/forms.ipynb

```
    built in Python Round
    The slowest run took 28.69 times longer than the fastest. This could mean that a\
n intermediate result is being cached.
    1000000 loops, best of 3: 468 ns per loop
```

%alias

The `%alias` command is a way to create shortcuts.

```
alias lscsv ls -l sample_data/*.csv
```

```
lscsv
```

```
    -rw-r--r-- 1 root root   301141 Oct 25 16:58 sample_data/california_housing_test\
.csv
    -rw-r--r-- 1 root root  1706430 Oct 25 16:58 sample_data/california_housing_trai\
n.csv
    -rw-r--r-- 1 root root 18289443 Oct 25 16:58 sample_data/mnist_test.csv
    -rw-r--r-- 1 root root 36523880 Oct 25 16:58 sample_data/mnist_train_small.csv
```

To learn more, you can see the Magic Command references[35].

%who

To print variables, an excellent command to understand is `%who`. This step shows what variables are declared.

```
var1=1
```

```
who
```

```
    drive     os     too_many_decimals     var     var1
```

```
too_many_decimals
```

[35] https://ipython.readthedocs.io/en/stable/interactive/magics.html

```
    1.912345897
```

%writefile

One useful way to write scripts (perhaps a whole command-line tool) is to use the `%%writefile` magic command.

```
%%writefile magic_stuff.py
import pandas as pd
df = pd.read_csv(
    "https://raw.githubusercontent.com/noahgift/food/master/data/features.en.openfoo\
dfacts.org.products.csv")
df.drop(["Unnamed: 0", "exceeded", "g_sum", "energy_100g"], axis=1, inplace=True) #d\
rop two rows we don't need
df = df.drop(df.index[[1,11877]]) #drop outlier
df.rename(index=str, columns={"reconstructed_energy": "energy_100g"}, inplace=True)
print(df.head())
```

```
    Writing magic_stuff.py
```

```
cat magic_stuff.py
```

```
    import pandas as pd
    df = pd.read_csv(
        "https://raw.githubusercontent.com/noahgift/food/master/data/features.en.ope\
nfoodfacts.org.products.csv")
    df.drop(["Unnamed: 0", "exceeded", "g_sum", "energy_100g"], axis=1, inplace=True\
) #drop two rows we don't need
    df = df.drop(df.index[[1,11877]]) #drop outlier
    df.rename(index=str, columns={"reconstructed_energy": "energy_100g"}, inplace=Tr\
ue)
    print(df.head())
```

```
!python magic_stuff.py
```

```
   fat_100g  ...                         product
0     28.57  ...  Banana Chips Sweetened (Whole)
2     57.14  ...           Organic Salted Nut Mix
3     18.75  ...                  Organic Muesli
4     36.67  ...                   Zen Party Mix
5     18.18  ...             Cinnamon Nut Granola

[5 rows x 7 columns]
```

```
!pip install -q pylint
```

....
[?25h

```
!pylint magic_stuff.py
```

```
************* Module magic_stuff
magic_stuff.py:3:0: C0301: Line too long (109/100) (line-too-long)
magic_stuff.py:4:0: C0301: Line too long (110/100) (line-too-long)
magic_stuff.py:5:24: C0326: Exactly one space required after comma
df = df.drop(df.index[[1,11877]]) #drop outlier
                        ^ (bad-whitespace)
magic_stuff.py:7:0: C0304: Final newline missing (missing-final-newline)
magic_stuff.py:1:0: C0114: Missing module docstring (missing-module-docstring)
magic_stuff.py:2:0: C0103: Constant name "df" doesn't conform to UPPER_CASE nami\
ng style (invalid-name)
magic_stuff.py:5:0: C0103: Constant name "df" doesn't conform to UPPER_CASE nami\
ng style (invalid-name)

-----------------------------------
Your code has been rated at -1.67/10
```

Bash

Another useful tool in automation is the use of `%%bash`. You can use it to execute bash commands.

```
%%bash
uname -a
ls
ps
```

```
    Linux bee9b4559c13 4.14.137+ #1 SMP Thu Aug 8 02:47:02 PDT 2019 x86_64 x86_64 x8\
6_64 GNU/Linux
    gdrive
    magic_stuff.py
    sample_data
        PID TTY          TIME CMD
          1 ?        00:00:00 run.sh
          9 ?        00:00:01 node
         24 ?        00:00:03 jupyter-noteboo
        119 ?        00:00:00 tail
        127 ?        00:00:06 python3
        298 ?        00:00:00 bash
        299 ?        00:00:00 drive
        300 ?        00:00:00 grep
        383 ?        00:00:01 drive
        394 ?        00:00:00 fusermount <defunct>
        450 ?        00:00:00 bash
        451 ?        00:00:00 tail
        452 ?        00:00:00 grep
        551 ?        00:00:00 bash
        554 ?        00:00:00 ps
```

```
!uname -a
```

Linux bee9b4559c13 4.14.137+ #1 SMP Thu Aug 8 02:47:02 PDT 2019 x86_64 x86_64 x86_64 GNU/Linux

Python2

Yet another useful trick is the ability to run Python2.

```
%%python2
print "old school"
```

```
    old school
```

Python executable

You can run scripts, REPL and even run python statements with -c flag and semicolon to string together multiple statements

```
!python -c "import os;print(os.listdir())"
```

```
    ['.config', 'magic_stuff.py', 'gdrive', 'sample_data']
```

```
!ls -l
```

total 712
drwx—— 4 root root 4096 Sep 9 16:16 gdrive
-rw-r–r– 1 root root 407 Sep 9 16:20 magic_stuff.py
-rw-r–r– 1 root root 712814 Sep 9 16:25 pytorch.pptx
drwxr-xr-x 1 root root 4096 Aug 27 16:17 sample_data

```
!pip install -q yellowbrick
```

```
#this is how you capture input to a program
import sys;sys.argv
```

```
    ['/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py',
     '-f',
     '/root/.local/share/jupyter/runtime/kernel-559953d0-ac45-4a1a-a716-8951070eaab5\
.json']
```

In summary, IPython and Jupyter are beneficial tools to develop any Python code. They come in especially handy in testing out automation. Put a little IPython into your command-line tool kit, and you won't regret it.

Chapter 4: Integrating Linux Commands with Click

Alfredo Deza

Python comes with lots of different utilities to interact with a system. Listing directories, getting file information, and even lower-level operations like socket communications. There are situations where these are not sufficient or just not solving the right problem for us. I remember this time I worked with Ceph (a distributed file storage system)[36] and had to interact with different disk utilities. There are quite a few tools to retrieve device information like `blkid`, `lsblk`, and `parted`. They all have some overlap and some distinct features. I had to retrieve some specific information from a device that one tool wouldn't have and then go to a different tool to retrieve the rest.

To make matters worse, and because Ceph supports different various Linux distributions, some tools didn't have the features I needed on older versions of a particular Linux distro. What a problem. I ended up creating utilities that would try one tool first and then fall back to the others if they failed, with an order of preference. The result, however, ended up being very robust and resilient to all of these differences. Along the way, there are a few essential pieces that need to be in place, though, and in this chapter, I go through these *pieces* that make the interaction seamless, practical, and extraordinary resiliency.

Understand subprocess

If you search on the internet for how to run a system command from Python, you shouldn't be surprised to find hundreds (thousands?) of examples that may show something like this one:

```
>>> import subprocess
>>> subprocess.call(['ls', '-l'])
total 13512
drwxrwxr-x    9 root  admin      288 Feb 11 13:13 itcl4.1.1
-rwxrwxr-x    1 root  admin  2752568 Dec 18 14:06 libcrypto.1.1.dylib
-rwxrwxr-x    1 root  admin    88244 Dec 18 14:07 libformw.5.dylib
-rwxrwxr-x    1 root  admin    43080 Dec 18 14:07 libmenuw.5.dylib
-rwxrwxr-x    1 root  admin   408344 Dec 18 14:07 libncursesw.5.dylib
-rwxrwxr-x    1 root  admin    25924 Dec 18 14:07 libpanelw.5.dylib
-rwxrwxr-x    1 root  admin   529676 Dec 18 14:06 libssl.1.1.dylib
-r-xrwxr-x    1 root  admin  1441716 Dec 18 14:06 libtcl8.6.dylib
```

[36] https://ceph.io/

```
drwxrwxr-x    5 root  admin      160 Dec 18 14:06 tcl8
drwxrwxr-x   17 root  admin      544 Feb 11 13:13 tcl8.6
-rw-rw-r--    1 root  admin     8275 Dec 18 14:06 tclConfig.sh
drwxrwxr-x    5 root  admin      160 Feb 11 13:13 thread2.8.2
-rw-rw-r--    1 root  admin     4351 Dec 18 14:07 tkConfig.sh
0
```

If the example is trying to capture the results and assign it to a variable, it may use something like this though:

```
>>> from subprocess import Popen, PIPE
>>> process = subprocess.Popen(['ls', '-l'], stdout=PIPE)
>>> output = process.stdout.read()
>>> for line in output.decode('utf-8').split('\n'):
        print(line)
total 13512
drwxrwxr-x    9 root  admin      288 Feb 11 13:13 itcl4.1.1
-rwxrwxr-x    1 root  admin  2752568 Dec 18 14:06 libcrypto.1.1.dylib
-rwxrwxr-x    1 root  admin    88244 Dec 18 14:07 libformw.5.dylib
-rwxrwxr-x    1 root  admin    43080 Dec 18 14:07 libmenuw.5.dylib
-rwxrwxr-x    1 root  admin   408344 Dec 18 14:07 libncursesw.5.dylib
-rwxrwxr-x    1 root  admin    25924 Dec 18 14:07 libpanelw.5.dylib
-rwxrwxr-x    1 root  admin   529676 Dec 18 14:06 libssl.1.1.dylib
-r-xrwxr-x    1 root  admin  1441716 Dec 18 14:06 libtcl8.6.dylib
drwxrwxr-x    5 root  admin      160 Dec 18 14:06 tcl8
drwxrwxr-x   17 root  admin      544 Feb 11 13:13 tcl8.6
-rw-rw-r--    1 root  admin     8275 Dec 18 14:06 tclConfig.sh
drwxrwxr-x    5 root  admin      160 Feb 11 13:13 thread2.8.2
-rw-rw-r--    1 root  admin     4351 Dec 18 14:07 tkConfig.sh
0
```

The `output` variable `saves the return string from` process.stdout.read(), `then it gets decoded` (read()‘ returns bytes, not a string), and finally, it prints the result. This is not very useful, except for demonstrating how to keep the output around for processing.

These examples are everywhere. Some go further into checking exit status codes and waiting for the command to finish, but they are lacking into crucial factors of correctly (and safely) interacting with system commands. These are a few questions that come to mind that need answering when crafting these types of interactions:

- What happens if the tool does not exist or is not in the path?
- What to do when the tool has an error?
- If the tool takes too long to run, how do I know if it is hanging or doing actual work?

Running system commands looks easy, but it is vital to understand resilient interfaces so that addressing failures is easier.

There are, primarily, two types of system calls you interact with: one doesn't care about the output, like starting a web server, and the other one that produces useful output that needs to be processed. There are strategies that you need to apply to each one depending on the use case to ensure a transparent interface. As long as there is consistency, these system interactions are easy to work with.

Parsing Results

When the output of a system command gets saved for post-processing, then parsing code *must* be implemented. The parsing can be as easy as checking if a specific word is in the output as a whole. For more involved output, a line-by-line parsing has to be crafted. The more difficult the output, the more chance there is for brittleness in the handling. The parsing then needs to apply a strategy from simple (more robust) to complex (prone to breakage). One common thought is to apply a regular expression to anything coming out from a system command to parse, but this is usually my last resort, as it is incredibly hard to debug compared to non-regular-expression approaches.

If the tool you are calling from Python has a way to produce a machine-readable format, then use it. It is almost always the best chance you have for an easy path to parsing results. The machine-readable format can be many things, most commonly it is JSON[37] or CSV (Comma Separated Values). Python has native support for loading these and interacting with them easily.

I find that working directly with `subprocess.Popen` takes too much boilerplate code, so I usually create a utility that runs a command and always returns the `stdout`, `stderr`, and exit code. It looks like this:

```
import subprocess


def run(command):
    process = subprocess.Popen(
        command,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
    )

    stdout_stream = process.stdout.read()
    stderr_stream = process.stderr.read()
    returncode = process.wait()
    if not isinstance(stdout_stream, str):
```

[37] https://www.json.org/json-en.html

```
        stdout_stream = stdout_stream.decode('utf-8')
    if not isinstance(stderr_stream, str):
        stderr_stream = stderr_stream.decode('utf-8')
    stdout = stdout_stream.splitlines()
    stderr = stderr_stream.splitlines()

    return stdout, stderr, returncode
```

It accepts the command (as a list), then reads all the output produced by both `stdout` and `stderr` and decodes them if necessary. The utility splits the lines for processing them later, making it easier to consume the output.

Simple Parsing

The simplest parsing, using the `run()` example utility, is going to be checking if there is a specific line or string in the output. For example, imagine you need to check if a given device is using the XFS file system. A utility needs to check if the reporting is mentioning that it is XFS, and nothing more. In my system, I can do this with the following:

```
$ sudo blkid /dev/sda1
/dev/sda1: UUID="8ac075e3-1124-4bb6-bef7-a6811bf8b870" TYPE="xfs"
```

So all I need to do is check if `TYPE="xfs"` is in the output. The utility becomes straightforward:

```
def is_xfs(device):
    stdout, stderr, out = run(['sudo', 'blkid', device])
    if 'TYPE="xfs"' in stdout:
        return True
    return False
```

Interacting with this utility is easy enough, including when there are errors that the `blkid` tool reports but don't matter to determine if a device is using the XFS filesystem:

```
>>> is_xfs('/dev/sda1')
 True
>>> is_xfs('/dev/sda2')
False
>>> is_xfs('/dev/sdfooobar')
False
```

In some cases, no parsing at all needs to happen because the exit code gives you everything you need to know. That system call and its subsequent exit code answers the question: "Was the command successful?". A quick example of this is using the Docker[38] command-line tool. Have you ever tried stopping a container? It is pretty simple, first check if the container is running:

[38] https://www.docker.com/

```
$ docker ps | grep pytest
542818cd6d7f        anchore/inline-scan:latest   "docker-entrypoint.sh"   14 minutes\
 ago      Up 14 minutes (healthy)   5000/tcp, 5432/tcp, 0.0.0.0:8228->8228/tcp   pyt\
est_inline_scan
```

I've confirmed it is running so now I stop the `pytest_inline_scan` container, and then check its exit code:

```
$ docker stop pytest_inline_scan
pytest_inline_scan
$ echo $?
0
```

The container is stopped, and its exit status code is `0`. Although I highly recommend using the Python Docker SDK[39], you can use this minimal example to guide you when you only need to check for an exit status:

```
def stop_container(container):
    stdout, stderr, code = run(['docker', 'stop', container])
    if code != 0:
        raise RuntimeError(f'Unable to stop {container}')
```

Advanced Parsing

There are multiple levels of painful parsing for command-line output. Like I've mentioned earlier in this chapter, you should always try to tackle the problem with the simple approaches first, then move on to machine-readable output if possible or just checking the exit code. This section dives into some of the advanced parsing I've implemented in production where most of the time, I avoid regular expressions until I've exhausted every other way, making it necessary.

Even though I highly recommend configuring tools to produce machine-readable formats like CSV or JSON, sometimes this is not possible. One time, I saw that the `lsblk` tool (another tool to inspect devices like `blkid`) had the `--json` flag to produce an easily consumable output in Python. After creating the implementation, I realized that this wouldn't work in older Linux distributions because that special flag didn't exist, as it is a somewhat new feature. To have one implementation that would work for older Linux versions as well as new ones, I had to do the hard thing: parsing.

The first thing to do is to separate the implementation in two parts: one that runs the command, and the other one that parses the output. This is crucial to do because testing, maintaining, and fixing any problems is easier when the pieces are isolated with lots of tests to ensure expected behavior. I first start by running the command to produce the output I'm going to work within the parser. I know what the command is, and how the flags should be issued, so I want to verify the output before writing the parsing:

[39] https://docker-py.readthedocs.io/en/stable/

```
$ lsblk -P -p -o NAME,PARTLABEL,TYPE /dev/sda
NAME="/dev/sda" PARTLABEL="" TYPE="disk"
NAME="/dev/sda1" PARTLABEL="" TYPE="part"
```

The tool allows a little bit of machine-readable friendliness with the `-P` flag which produces pairs of values, as if this output was going to be read by a BASH script. The `-p` flag uses absolute paths for the output, and finally `-o` specifies what *device labels* I'm interested in. With that output, the parsing part can get started. I want the parsing to return a dictionary, so I need to extract the label (`NAME` for example) as a key, and then the value that is within the quotes. A good way to tinker with the parsing is to do this in the Python shell:

```
>>> line = 'NAME="/dev/sda" PARTLABEL="" TYPE="disk"'
>>> line
'NAME="/dev/sda" PARTLABEL="" TYPE="disk"'
>>> line.split(' ')
['NAME="/dev/sda"', 'PARTLABEL=""', 'TYPE="disk"']
>>> line.split('" ')
['NAME="/dev/sda', 'PARTLABEL="', 'TYPE="disk"']
```

I try two ways of splitting the line, and I decide to use the last one that splits on the double quote because it partially cleans up the value. These are minor implementation details, and you can try other ways of splitting this that can be more efficient. The important part is to separate the parsing from the command execution function, add tests, and narrow down the path by playing with the output in a Python shell.

Now that I'm happy with the splitting, I do another pass of splitting each item to produce the pairs:

```
>>> for item in line.split('" '):
...     item.split('="')
...
['NAME', '/dev/sda']
['PARTLABEL', '']
['TYPE', 'disk"']
>>>
```

Very close. There is a trailing quote that needs cleaning in the last item, but the good thing is that the items are now paired nicely, so the parsing doesn't need to guess what value needs to go with what key. For example, if one value was empty or if there are spaces and the splitting goes bad, it is easier because of grouping. Let's try again by adding items into a dictionary and cleaning up further:

```
>>> parsed = {}
>>> for item in line.split('" '):
...     key, value = item.split('="')
...     parsed[key] = value.strip('"')
...
>>> parsed
{'NAME': '/dev/sda', 'PARTLABEL': '', 'TYPE': 'disk'}
```

Excellent. The parsing side is complete, as I'm happy with the result in the shell. Before moving forward, I write a dozen unit-tests to make sure I got this right. However, I know you are thinking about doing this with regular expressions, because that would be super easy to do, and you think I'm plain silly in not writing a simple regular expression that would split on a group of upper case letters. Regular expressions are not straightforward, and they are hard to test (no `if` or `else` conditions, impossible to get a sense of coverage). Let's try a couple of rounds of regular expressions to achieve the same result.

First, and somewhat on cheating mode here, I split on whitespace:

```
>>> import re
>>> line = 'NAME="/dev/sda" PARTLABEL="" TYPE="disk"'
>>> line
'NAME="/dev/sda" PARTLABEL="" TYPE="disk"'
>>> re.split('\s+', line)
['NAME="/dev/sda"', 'PARTLABEL=""', 'TYPE="disk"']
```

I now have each part with its pairs. Many regular expressions can be thrown at these items in the list to produce what we want. I'm the person at work that tries the simplest approach first and ensure that it works great. Simple always wins for me. Instead of splitting further, I use a regular expression to get rid of the characters I don't need:

```
>>> for item in re.split('\s+', line):
...     re.sub('("|=)', ' ', item)
...
'NAME  /dev/sda '
'PARTLABEL   '
'TYPE  disk '
```

It looks broken without any delimiters or quotes as it was presented before, but this is fine because in the next step, I split on whitespace which is the fantastic default for the `.split()` string method:

```
>>> for item in re.split('\s+', line):
...       result = re.sub('("|=)', ' ', item)
...       result.split()
...
['NAME', '/dev/sda']
['PARTLABEL']
['TYPE', 'disk']
```

In this case, `result` is each string separated by whitespace that was produced by replacing the characters I don't want. Then `.split()` removes that whitespace creating the pairs. The final piece of code that produces the dictionary mapping is now easy:

```
>>> parsed = {}
>>> for item in re.split('\s+', line):
...     result = re.sub('("|=)', ' ', item)
...     key, value = result.split()
...     parsed[key] = value
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: need more than 1 value to unpack
```

Oh no. What happened here? I was distracted and didn't realize that by replacing the double quotes along with the equal character it caused some values to disappear. That is why `PARTLABEL=""` produces just one item, not two. The fix is to just remove the = character and clean up the quoted value later:

```
>>> for item in re.split('\s+', line):
...     result = re.sub('=', ' ', item)
...     key, value = result.split()
...     parsed[key] = value.strip('"')
...
>>> parsed
{'TYPE': 'disk', 'NAME': '/dev/sda', 'PARTLABEL': ''}
```

I mentioned I cheated because, in the end, I'm still trying to follow the pattern of splitting and then doing further cleaning. Separating the parsing process in this way is easier to understand and improve when output isn't conforming to the expectations. This chapter doesn't cover the testing part of this, but it is imperative to add as many tests as possible to ensure the parsing is correct.

The end result of these approaches gives us a nice API to work with:

```
def _lsblk_parser(lines):
    parsed = {}
    for line in lines:
        for item in lines.split('" '):
            key, value = item.split('="')
            parsed[key] = value.strip('"')

    return parsed


def lsblk(device):
    command = [
        'lsblk',
        '-P',   # Produce pairs of key/value
        '-p',   # Return absolute paths
        '-o',   # Define the labels we are interested in
        'NAME,PARTLABEL,TYPE',
        device
    ]

    stdout, stderr, code = run(command)
    return _lsblk_parser(stdout)
```

And the resulting API interaction with the functions:

```
>>> lsblk('/dev/sda')
{'NAME': '/dev/sda1', 'PARTLABEL': '', 'TYPE': 'part'}
>>> lsblk('/dev/sda1')
{'NAME': '/dev/sda1', 'PARTLABEL': '', 'TYPE': 'part'}
>>> lsblk('/dev/sdb')
{}
```

Shell Safety

One of the things that prevented me from loving Python in the first place (I was coming from BASH), was that `subprocess.Popen` defaults to accepting a list for the command to run. This can get very tiring and repetitive quite fast, but it is generally *safer* to use a list. It doesn't mean that using a plain string (accepted if `shell=True` gets used in `Popen`) is inadequate or very unsafe. It depends where this string is coming from. In all the examples in this chapter, the input was curated and carefully constructed by the functions. Still, if the interfaces in a command-line tool were to accept input from a user, then that is a security concern. Python spawns a sub-shell to evaluate the input first,

expanding variables, for example, which can have undesired effects. In security, this is called *shell injection*, where input can result in other arbitrary execution.

Even in the case where you aren't accepting input from external sources, you are at the mercy of the system and how it interprets (or expands) variables and other behavior with a sub-shell.

In short: don't use `shell=True` to pass a whole string and always sanitize input coming from the user.

Chapter 5: Writing pure Bash or ZSH command-line tools

Alfredo Deza

I've been horrified before trying to figure out a piece of production code that was mixing shell scripting and Python. Why would one try to do something like this? A step further was when a large (and custom) Python test framework was doing a system call to a shell that then itself executed Python on a remote system. Can you imagine fixing bugs in that codebase? Where do you start? In the remote server running some odd version of Python, or using a previous version of BASH that has a built-in that behaves differently? Or perhaps in Python that can change some subtle things (like dictionary ordering) from one version to the other?

```
    writes = run(
        args=[
            'sudo', 'mkdir', '-p', '/etc/app', escaped('&&'),
            'sudo', 'chmod', '0755', '/etc/app', escaped('&&'),
            'sudo', 'python',
            '-c',
            'import shutil, sys; shutil.copyfileobj(sys.stdin, file(sys.argv[1], "wb\
"))',
            conf_path,
            escaped('&&'),
            'sudo', 'chmod', '0644', conf_path,
            ],
        stdin=run.PIPE,
        wait=False,
        )
    feed_many_stdins_and_close(conf_fp, writes)
    run.wait(writes)
```

I've changed some paths and names, the idea here is not to point fingers looking to blame anyone - I'm guilty of writing horrendous code before too! This piece of code is like using real lunar dust to create a representation of the Moon for your 4th grade Science Class. There is absolutely no need whatsoever to do this, but you certainly can. The example has a few red flags; it compounds multiple shell statements into one using an escaped double ampersand (`&&`), which is problematic if one of these pieces fail. That removes the nicety of fine error control and introduces all the roughness in shell scripting. Next, it calls out to Python (remember this is originating from Python) as a shell

command that executes something that could very well be another shell command: copies a file from `stdin`. Finally, it changes the permissions in the path, returning to Python.

This chapter *does not* intend to encourage this type of programming, and I start with a bad example because it demonstrates how easy one can abuse the flexibility of Python and the shell. On the contrary, this chapter showcases some good uses, where mixing some shell scripts with Python, and Python within shell scripts is perfectly valid and useful.

Understanding environmental variables

Environment variables can be magical, and to some extent they can be useful. One of the problems with environment variables is that it isn't always possible to tell where these are coming from because they can be overridden. Environment variables are variables that are defined in the system and are most commonly used in shell scripts. These variables are also available through Python, so it is possible to inspect them from there. Try a quick test in your computer by opening a terminal and running the `env` command:

```
$ env
TERM_SESSION_ID=w1t10p0:54756B1E-79CF-46D6-B6C9-98DBE779ABA2
LC_TERMINAL_VERSION=3.3.9
COLORFGBG=15;0
ITERM_PROFILE=Alfredo
XPC_FLAGS=0x0
LANG=en_US.UTF-8
PWD=/Users/alfredo/python/python-command-line-tools-book
SHELL=/bin/zsh
TERM_PROGRAM_VERSION=3.3.9
TERM_PROGRAM=iTerm.app
PATH=/Users/alfredo/go/bin:/usr/local/go/bin:/usr/local/bin:\
/Library/Frameworks/Python.framework/Versions/3.5/bin:\
/Library/Frameworks/Python.framework/Versions/3.7/bin:\
/Library/Frameworks/Python.framework/Versions/3.8/bin:\
/Library/Frameworks/Python.framework/Versions/3.6/bin:\
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/sbin:\
/usr/local/mysql/bin:/Users/alfredo/bin:/usr/texbin:/usr/local/go/bin
LC_TERMINAL=iTerm2
COLORTERM=truecolor
TERM=xterm-256color
HOME=/Users/alfredo
TMPDIR=/var/folders/pz/vqrg684d10n8jmv6fz60kxjw0000gn/T/
USER=alfredo
LOGNAME=alfredo
```

```
PIP_DOWNLOAD_CACHE=/Users/alfredo/.pip_cache
GOROOT=/usr/local/go
GOPATH=/Users/alfredo/go
KEYTIMEOUT=1
ARCHFLAGS=-arch i386 -arch x86_64
MAKEOPTS=-j17
LESS=FRSXQ
PYTHONSTARTUP=/Users/alfredo/dotfiles/pythonstartup.py
EDITOR=/usr/local/bin/vim
_=/usr/bin/env
```

A lot shows in my system. It gives you some insight on what I'm using and how. The text editor I prefer (Vim) is set there, as well as the shell (ZSH). Other tools like Git can access these values and use the preferred text editor when crafting a commit message, for example. Environment variables allow programs (or environments) to set some named values to use them throughout the program or interchangeably with other applications. Imagine if you had to create a program in Python and then retrieve some values from a different program or service like the Nginx web server, it would be challenging if it wasn't for environment variables.

As I've mentioned, the environment variables already set for my user are accessible via Python with the `os` module:

```
>>> import os
>>> os.environ['LOGNAME']
alfredo
```

As you can see, `os.environ` is a mapping that behaves almost exactly like a plain Python dictionary. Accessing keys and values is the same, and the module allows a few extra methods for manipulating the environment like `putenv()` and `unsetenv()`. These two helpers are dependent on support by the system where Python is running, so you may not see them always when running Python. It is safe to treat the `os.environ` mapping as a plain dictionary, where you set keys and values and use them elsewhere.

One thing to be careful, and has created issues for me before, is that you are not allowed to set any value that isn't a string. This constraint comes from environment variables themselves, which are always strings assigned to keys. If you are unaware of this detail you get an error similar to this:

```
>>> import os
>>> os.environ['my_value'] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/os.py", line\
 678, in __setitem__
    value = self.encodevalue(value)
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/os.py", line\
 748, in encode
    raise TypeError("str expected, not %s"
```

Another important thing to understand is that environment variables manipulated in Python are not long-lived. They persist for however long the Python program is running, and this is true for both setting new environment variables as well for removing or altering existing ones:

```
>>> os.environ['my_value'] = '1'
>>> os.environ['my_value']
'1'
>>> ^D
```

Then on the terminal, the `my_value` variable is just not there:

```
$ env | grep my_value || echo "variable not found"
variable not found
```

Manipulating an existing variable yields similar behavior:

```
>>> os.environ['SHELL']
'/bin/zsh'
>>> os.environ['SHELL'] = ''
>>> os.environ['SHELL']
''
```

After setting `SHELL` to an empty string and exiting the program, the variable hasn't changed at all:

```
$ env | grep SHELL
SHELL=/bin/zsh
```

Understand shell profiles

Profiles are different files that can override the system-wide configuration of your shell environment. I prefer using ZSH[40], but these notions apply for a vast majority of shells out there, including the omnipresent BASH shell. Depending on the system, these files have different locations. For OSX, there is a system-wide BASH configuration file for example:

```
$ cat /etc/bashrc
# System-wide .bashrc file for interactive bash(1) shells.
if [ -z "$PS1" ]; then
   return
fi

PS1='\h:\W \u\$ '
# Make bash check its window size after a process completes
shopt -s checkwinsize

[ -r "/etc/bashrc_$TERM_PROGRAM" ] && . "/etc/bashrc_$TERM_PROGRAM"
```

For the most part, you add configuration for your environment in the specific configuration file for your user. A system can have many different users, but (ideally) a user is only for one person, which can have modifications done in one file that gets loaded every time a new session starts. A new session could be a new terminal window (or tab) opened, or login into a machine. There are still subtle differences between login into a system and opening a new terminal (creating a new session), and just for informational purposes, these are a list of how these files are loaded (in order of execution) in BASH:

- `/etc/profile` : Read first, for system-wide configurations.
- `~/.bash_profile` : The file that determines configuration for login shells, that is, login into a new shell (unlike opening a new terminal window)
- `~/.bash_login` : Same (analogous) to `~/.bash_profile`. It is confusing but safe to ignore; you should use `~/.bash_profile` instead.
- `~/.profile` : The fallback legacy file that it reads in case it exists, and primarily for backwards compatibility with administrators that still use this file.
- `~/.bashrc` : Finally, the user configuration file that gets read both for login into a new shell and creating a new session, and it is primarily the place where users customize their shell environment, like adding environment variable or aliases.

[40] https://www.zsh.org/

Customizing your shell

Having some helpers and neat aliases is helpful. Almost always, these customizations go into the shell configuration file (`~/.bashrc` for BASH and `~/.zshrc` for ZSH). Although I like ZSH and it is what I've used for the past few years, the examples below should work fine with BASH as well. I advise you to keep your customizations in version control to ensure these aren't lost, maintaining a track of changes throughout time. Once in version control, I create symbolic links from the repository where my configurations are, back to what the SHELL needs. For example, most of my configuration files are in my dotfiles[41] repository, so I clone it in my home directory and then link them. For the `.zshrc` file that would be something like this:

```
$ git clone https://github.com/alfredodeza/dotfiles.git
[...]
$ ln -s dotfiles/.zshrc ~/.zshrc
```

If the `.zshrc` file didn't exist before, the linking succeeds, and my full customization is ready to go. There are several things I use for my shell, and I demonstrate them in the next examples.

Having a long history (or almost unlimited history) is great because you can forget a useful command, or set of commands, from three months ago. History is there to save you, for BASH it looks like this:

```
export HISTFILESIZE=
export HISTSIZE=
export HISTTIMEFORMAT="[%F %T] "
export HISTFILE=~/.bash_eternal_history
```

And for ZSH, I do this:

```
HISTFILE=$HOME/.zsh_history
HISTSIZE=10000000
SAVEHIST=10000000
setopt SHARE_HISTORY
setopt APPEND_HISTORY
```

Instead of using the `history` built-in to search what you need, you can use `Ctrl-r` to do a reverse search of history that works looking into the full history content as you type a command or even partial commands. Try it out by pressing `Ctrl-r` and then typing the command you are looking for. In ZSH it looks something like this:

[41] https://github.com/alfredodeza/dotfiles

```
$ sudo rm -rf sha256
bck-i-search: sudo _
```

The `bck-i-searc` is where I am typing `sudo`, and the prompt above me keeps updating with what matches. BASH looks a bit similar:

```
(reverse-i-search)`ls': ls
```

The typed command appears right after the `reverse-i-search`, and the result is after the colon.

Many of my customizations are for ZSH only, and since most everyone uses BASH primarily, I'll concentrate on the examples that can work interchangeably. Aliases is one of them, here is one that allows me to quickly move up directories with dots instead of writing `cd ../../`:

```
# cd aliases
alias ..="cd .."
alias ...="cd ../.."
alias ....="cd ../../.."
alias .....="cd ../../../.."
alias ......="cd ../../../../.."
```

As a Vim user, I like to exit a shell without being required to do `Ctrl-D`, so an alias for `:q` is great, bonus points for upper case variant because sometimes I press the `Shift` key too long:

```
alias \:q='exit'
alias \:Q='exit'
```

If your shell doesn't display colors to differentiate between directories, files, and executables, this handy alias works well, regardless of Linux or OSX:

```
ls --color -d . &>/dev/null 2>&1 && alias ls='ls --color=if-tty' || alias ls='ls -G'
```

Write Shell functions

Aside from small customizations and aliases that go in your `.profile` or shell config file, a neat trick to explore is writing shell functions. Any shell function that gets defined in those files as the shell starts up is available in the terminal as an executable:

```
my_function() {
    echo "This is actually a function!"
}
```

After saving that function in your shell config file, start a new session, so the file is re-read. The `my_function` appears as an available command that can get called:

```
$ my_function
This is actually a function!
```

Arguments and options can work with these functions and expand their usage. The main problem with these is that you have to make sure they don't get big. Anything larger than ten lines of shell is aking to be an actual command-line tool. As long as you are aware of this, feel free to keep adding them when you need to solve a particular problem with just a couple of lines in a shell function. One particular problem I encounter every now and then is that I'm not sure where a Python module is coming from. Since Python keeps having issues figuring out proper packaging solutions, users need to rely on virtual environments and solving problems with libraries installed in different places. One way to quickly check where a module exists is by importing it and printing it in an interactive Python session:

```
>>> import click
>>> print(click)
<module 'click' from '/Users/alfredo/.virtualenvs/cli/lib/python3.8/site-packages/cl\
ick/__init__.py'>
```

I don't want to start an interactive Python shell to check on module locations every time, so a quick function makes sense here. First, I need to make sure a module can be imported, handling exceptions if that is the case, and finally return the path. I call this helper `try()`:

```
try() {
    python3 -c "
exec('''
try:
    import ${1} as _
    print(_.__file__)
except Exception as e:
    print(e)
''')"
}
```

Its usage in the command-line is straightforward; it only accepts an argument:

```
$ try os
/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/os.py
$ try click
/Users/alfredo/cli/lib/python3.8/site-packages/click/__init__.py
$ try foo
No module named 'foo'
```

To help with other interesting package metadata, I use the `pkg_resources` module which knows how to retrieve that information as well as the location of a module, I call this helper function `welp()`, and it depends on the `try()` helper from before:

```
welp() {
    P_VERSION=`python3 -c "
exec('''
try:
    import pkg_resources
    print(pkg_resources.get_distribution(\'${1}\').version)
except Exception:
    print(\'Not found\')
''')"`
    echo "Path: $(try ${1})"
    echo "Version: ${P_VERSION}"
}
```

Very useful in the command-line:

```
$ welp foo
Path: No module named 'foo'
Version: Not found
$ welp click
Path: /Users/alfredo/cli/lib/python3.8/site-packages/click/__init__.py
Version: 7.1.1
```

Chapter 6: Use Advanced Click Features

Alfredo Deza

Once you are past simple command-line tools that have a particular objective, it is time to move on and explore solutions that perhaps you weren't aware you needed. Recently at work, I was in charge of rethinking the reporting output of a command-line tool that was providing a table with columns and headers as the default format. The problem with a header-and-column format is that users will end up wanting more columns and get more information. This is what ended happening to this tool at work. It was reporting so many columns that the output would not fit in my large monitor. When the information reported has any chance of including more fields, it will. Users aren't all the same; consumers are all different. Someone will want an extra field that doesn't make sense for others. My proposal to get rid of the columns was to report vertically and nesting anything that requires grouping:

```
Item 1
  total: 30
  size: 23MB
  path: /var/lib
  severity: High
  issues:
    - issue-17
    - issue-29
```

The output now looks a bit like structured YAML. It doesn't matter, because after taking a second look, it seemed I could do better. What if I could give visual cues that informed without using any more space? This is what coloring can do for you: red can indicate an error, while blue can be informational or for low-priority items. Using a red color for `Item 1` would implicitly describe this item as high severity, allowing the removal of `severity: High` as it would be redundant.

Next, command-line applications tend to like configuration files to implement human-readable defaults, or to avoid too many flags and values in the execution, simplifying working with a tool. Although there is no specific handler for configuration files, in this chapter, we also cover how to integrate config files into Click command-line tools. Finally, sub-commands and other friendly terminal-based interfaces complete the circle of powerful extras to enhance a tool that needs to grow in complexity. In chapter 1, I briefly covered sub-commands to explain how terrible sub-commands are implemented in Python's standard library (with `argparse` primarily), which is an excellent introduction. Let's go into more detail in this chapter.

Subcommands

We already covered a brief introduction of sub-commands in Chapter 1 while explaining the differences between what the Python standard library offers and how Click makes things way more interesting. Click, as with most everything it offers, is comprehensive yet still very simple most of the time.

In this section, I go into using Click to build a nested sub-command tool that talks to a real HTTP API. This is a typical case, where if an HTTP API exists, a *companion* command-line tool exists. A few years ago, I built a package repository that would store system packages (mostly RPM, and DEB) and then created repositories for different Linux Distributions automatically. The service also offered an API, and I had many users asking for tasks that I only had access to like:

- Forcefully remove a repository
- Recreate the repository (triggering a rebuild)
- Upload a package or set of packages
- Remove a package

Designing the command-line tool was fun, and it allowed me to think about how to compose these sub-commands to make sense. I encourage you to think deeply about how the commands read when executing them. One example of oddness when crafting sub-commands is in init systems (tools in charge of starting system services in a server). Which one of these would be nicer to work with?

- `service start nginx`
- `service nginx start`

There is no straightforward answer to which one is better. As with most things that are somewhat complex, the answer is that it depends. Using `start nginx` is great if you don't plan to add anything else to the command after it because `nginx` is the argument. It is also easier to implement since `nginx` is an argument (think of it as a value) and not something that is pre-configured (Nginx is only available if installed). The counterpart to this is valid as well. If you need to nest other sub-commands, flags, or options, then `nginx start` can be easier since `nginx` would be an actual sub-command. The implementation would be trickier because it gets dynamically generated.

For this section, I demonstrate how to create a tool with sub-commands that communicates with an external HTTP API. I've chosen the Jenkins CI/CD Platform[42], which is easy to install and startup, and many workplaces tend to have at least one running somewhere. I'm not covering how to install and configure Jenkins here, but you need to know the full HTTP address and ensure you can reach it from your current workstation. A fully qualified domain name (FQDN) of the running Jenkins instance, along with the port and the protocol (HTTPS for SSL, for example) is required. An easy way to run Jenkins is with a container; you can find setup instructions[43] in their Github repository. Using

[42] https://www.jenkins.io/
[43] https://github.com/jenkinsci/docker/blob/master/README.md

the container is what I chose for these examples, and after installing, I create a user (named `admin`) and generate a token. The token is crucial for talking to the HTTP API; without it, the examples don't work.

Since I have Jenkins running locally with a user named `admin`, that means my URL is: http://localhost:8080/[44]. And I've generated a new token using http://localhost:8080/user/admin/configure[45]. That token is not to be shared, and it should only be available once to copy it from the web interface. For me, that token looks similar to this: `1102a43986753ffb17ee76db2164bba0c8`. Now create a new virtual environment for this new tool, and install the Jenkins library, which is needed to talk to the HTTP API:

```
$ python3 -m venv venv
$ source venv/bin/activate
$ pip install python-jenkins
Collecting python-jenkins
  ...
Installing collected packages:
  idna, certifi, urllib3, chardet, requests, multi-key-dict,
  six, pbr, python-jenkins
  Running setup.py install for multi-key-dict ... done
Successfully installed
  certifi-2020.4.5.1 chardet-3.0.4 idna-2.9 multi-key-dict-2.0.3
  pbr-5.4.5 python-jenkins-1.7.0 requests-2.23.0 six-1.14.0 urllib3-1.25.9
```

Make sure you are installing `python-jenkins`, not `jenkins` which is a different library that doesn't work to talk to the Jenkins API. After installing, and ensuring you have Jenkins running and a token available, start a Python shell to interact with the API:

```
>>> import jenkins
>>> conn = jenkins.Jenkins(
  "http://localhost:8080/",
  username="admin",
  password="secret-token")
>>> conn.get_jobs()
[]
>>> conn.get_whoami()
{'_class': 'hudson.model.User',
 'absoluteUrl': 'http://localhost:8080/user/admin',
 'description': None,
 'fullName': 'admin',
 'id': 'admin',
```

[44] http://localhost:8080
[45] http://localhost:8080/user/admin/configure

```
 'property': [{'_class': 'jenkins.security.ApiTokenProperty'},
  {'_class': 'hudson.model.MyViewsProperty'},
  {'_class': 'hudson.model.PaneStatusProperties'},
  {'_class': 'jenkins.security.seed.UserSeedProperty'},
  {'_class': 'hudson.search.UserSearchProperty', 'insensitiveSearch': True},
  {'_class': 'hudson.model.TimeZoneProperty'},
  {'_class': 'hudson.security.HudsonPrivateSecurityRealm$Details'},
  {'_class': 'hudson.tasks.Mailer$UserProperty',
   'address': 'admin@example.com'}]}
```

After importing the library, a connection object returns from `jenkins.Jenkins`. One thing to note here is that the library wants a `password` argument, which in reality, is the secret token that got generated in the Jenkins web interface. Then, using the connection object, you can interact with Jenkins with almost anything, there is almost a complete parity between the web interface and what the library offers. In the example, I ask to get all the jobs (there aren't any, so an empty list returns), and then I use the `get_whoami()` method to get some metadata about my account.

Start by creating a new file with a single sub-command called `jobs`. This sub-command helps when interacting with Jenkins jobs as they come from the API. Remember to define the `token` that generated earlier. I save the example as `staffing.py` in reference to something that handles jobs:

```
import jenkins
import click

token = "secret"


@click.group()
def api():
    pass


@api.command()
def jobs(_list):
    """Interact with Jobs in a remote Jenkins server"""
    pass


if __name__ == '__main__':
    api()
```

Many things are missing from this first iteration, but now, the tool can be executed in the terminal and show some help menus, verifying that it is all wired up correctly:

```
$ python staffing.py jobs --help
Usage: staffing.py jobs [OPTIONS]

  Interact with Jobs in a remote Jenkins server

Options:
  --help  Show this message and exit.
```

Now add a flag to the `jobs()` function, that shows a list of jobs available in the Jenkins server:

```
url = 'http://localhost:8080'


@api.command()
@click.option('--list', '_list', is_flag=True, default=False)
def jobs(_list):
    """Interact with Jobs in a remote Jenkins server"""
    if _list:
        conn = jenkins.Jenkins(
            username='admin', password=token, url=url
        )
        jobs = conn.get_jobs()
        if jobs:
            for job in jobs:
                click.echo(job)
            return
        return click.echo(
            f'No jobs available in remote Jenkins server at: {url}'
        )
```

First, I include a new option that is aliased to `_list` so that it doesn't override the built-in `list` function in Python. I define it as a flag and that it defaults to `False`. Next, if `_list` is passed in, it creates a connection which calls `get_jobs()` and loops over each one. If a job is available, it shows in the output; otherwise, it returns a helpful message saying nothing is available yet for the given URL.

```
$ python staffing.py jobs --list
No jobs available in remote Jenkins server at: http://localhost:8080
```

After creating a quick example job in Jenkins the output is very different:

```
$ python staffing.py jobs --list
{
    '_class': 'hudson.model.FreeStyleProject',
    'name': 'example',
    'url': 'http://localhost:8080/job/example/',
    'color': 'notbuilt',
    'fullname':
    'example'
}
```

The JSON output looks odd in the terminal, make it a little bit better to extract a subset of useful fields:

```
@api.command()
@click.option('--list', '_list', is_flag=True, default=False)
def jobs(_list):
    """Interact with Jobs in a remote Jenkins server"""
    if _list:
        conn = jenkins.Jenkins(
            username='admin', password=token, url=url
        )
        jobs = conn.get_jobs()
        if jobs:
            for job in jobs:
                click.echo(job['fullname'])
                click.echo(' ' + job['url'])
                click.echo(' Status: {0}'.format(job['color']))
                click.echo('')
            return
        return click.echo(
            f'No jobs available in remote Jenkins server at: {url}'
        )
```

Terminal output is now much better:

```
python staffing.py jobs --list
example
 http://localhost:8080/job/example/
 Status: notbuilt
```

The remote connection for Jenkins is needed for the next commands, so extract it into a separate function to make it easier to reuse:

```
def connection():
    return jenkins.Jenkins(
        username='admin', password=token, url=url
    )
```

To dynamically load jobs that are present in the remote Jenkins server as sub-commands, a custom class needs to get in place and handle the loading. Click allows this with the `click.MultiCommand` class and this is how the implementation looks for this case:

```
class DynamicJobs(click.MultiCommand):

    def list_commands(self, ctx):
        conn = connection()
        jobs = conn.get_jobs()
        return [i['name'] for i in jobs]

    def get_command(self, ctx, name):
        conn = connection()
        jobs = conn.get_jobs()

        if name in [i['name'] for i in jobs]:
            return Job(
                name=name,
                params=[
                    click.Option(
                        ['-d', '--delete'], is_flag=True)
                ]
            )
```

This class can list jobs as sub-commands and map a remote job to an actual command (called `Job()`), which needs to be defined as well:

```
class Job(click.Command):

    def invoke(self, ctx):
        conn = connection()
        if ctx.params['delete']:
            click.echo(f'Deleting job: {self.name}')
            conn.delete_job(self.name)
        click.echo('Completed deletion of job')
```

Now create an empty sub-command called `job()`, which is the placeholder for all the dynamically loaded jobs. Whatever jobs exist in the remote Jenkins server, these jobs get listed here:

```
@api.group(cls=DynamicJobs)
def job():
    pass
```

With the two classes and the new `job()` sub-command, the command-line tool can verify that it is reading correctly from the remote server and mapping to sub-commands:

```
python staffing.py job --help
Usage: staffing.py job [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  example
```

In this output, `example` is the remote Jenkins job. If I create another job named `deploy` appears here as well. This is how it looks now:

```
python staffing.py job --help

Usage: staffing.py job [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  deploy
  example
```

The two classes added, `DynamicJobs()` which loads and matches, and `Job()` which does the actual work, are crucial to add behavior to these lazily loaded commands. An option exists for deleting the job, which you verify by using the name of the job as the command:

```
$ python staffing.py job deploy --help

Usage: staffing.py job deploy [OPTIONS]

Options:
  -d, --delete
  --help        Show this message and exit.
```

Go ahead and delete that job now and verify the output once again:

```
$ python staffing.py job deploy --delete

Deleting job: deploy
Completed deletion of job
```

Running the `staffing.py` tool once more shows that the job no longer exists, and thus, it no longer appears in the help menu:

```
$ python staffing.py job --help

Usage: staffing.py job [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  example
```

There are many ways to extend tools and sub-commands, and being able to interact with a remote API (Jenkins, in this case) is an excellent way to make it easier to handle complex interactions.

Utilities

There are a few utilities that I like to mention about Click because they make it so much easier to work with. Handling colors, launching a pager or an editor, or even handling files seamlessly between `stdin` and regular file paths is straightforward. You will probably not need all of these, but knowing they exist is useful to address the different needs of command-line tools. I'm guilty of implementing a lot of these utilities before, with implementations that aren't as nice as those included with Click.

Colored Output

I'm guilty of implementing colored output (from scratch) many times. Proper colored output is difficult to get right. If you haven't tried this before, it involves sending ASCII escape codes to the terminal, which, in turn, is interpreted as a specific color. Not only one needs to send these escape codes to the terminal (as part of the string), but a reset code is needed when the coloring needs to end. This is because, otherwise, the coloring wouldn't know where to stop. Neither newlines or whitespace is a delimiter for colors to stop.

If you are curious, this is a handy mapping I've written many times for some basic colors:

```
colors = dict(
    blue='\033[34m',
    green='\033[92m',
    yellow='\033[33m',
    red='\033[91m',
    bold='\033[1m',
    ends='\033[0m'
)
```

The `ends` key, indicates that the coloring has ended. The implementation on strings looks something like this:

```
>>> yellow = colors['yellow']
>>> ends = colors['ends']
>>> message = '{0}This is a warning{0}'.format(yellow, ends)
>>> message
'\x1b[33mThis is a warning\x1b[33m'
```

If you print the string or write it directly to the terminal with some other method, it makes the string yellow while keeping any subsequent string or lines the default color. All this bookkeeping is tedious and requires more granular controls to determine if a terminal is capable of sending colors or disable them altogether. This level of control is useful for programs that run on continuous integration systems that run commands, capture the output, and implement it correctly. Luckily, Click addresses this problem head-on, and allows controlling colored output with a separate library.

There are two ways (unfortunately) to accomplish coloring of output through Click:

```
import click

# Output combining click.echo and click.style
click.echo(click.style('Command Line Tools Book!', fg='green'))

# Output with click.secho
click.secho('Command Line Tools Book!', fg='green', blink=True)
```

The API for combining both `click.echo` and `click.style` is what I prefer as it reads much better.

There are many combinations to use for coloring text output, and my suggestion is to try the simplest first while making a conscious effort to avoid adding more color combinations. Click allows changing the background to a different color from the foreground, use bold characters, and even blink. Don't abuse these, and try to make it as minimal as possible.

File handling

File handling is not easy. It *is easy* in the sense that opening a file and reading it in Python is straightforward, but inside command-line applications it isn't. Many different constraints need to get preemptively addressed to have a solid command-line tool. Here are just a few things that can happen that one needs to guard against:

- A file path can be misspelled by the user
- The path might be correct but permissions aren't sufficient to open it
- After opening the file, the encoding is not handled properly, causing encoding errors
- Relative paths or shortened paths (like `~/`) need to get expanded
- `stdin` is handled differently, even though it behaves like a file descriptor

You still need to handle all of the above, but the interface that Click offers is seamless and makes it less painful. Here are a few examples on what happens in error conditions:

```
>>> click.open_file('/etc/aliases.db').read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lib/python/site-packages/click/utils.py", line 336, in open_file
    f, should_close = open_stream(filename, mode, encoding, errors, atomic)
  File "lib/python/site-packages/click/_compat.py", line 533, in open_stream
    return _wrap_io_open(filename, mode, encoding, errors), True
  File "lib/python/site-packages/click/_compat.py", line 511, in _wrap_io_open
    return io.open(file, mode, **kwargs)
PermissionError: [Errno 13] Permission denied: '/etc/aliases.db'
```

When the file can't be found:

```
>>> click.open_file('/etc/bash')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lib/python/site-packages/click/utils.py", line 336, in open_file
    f, should_close = open_stream(filename, mode, encoding, errors, atomic)
  File "lib/python/site-packages/click/_compat.py", line 533, in open_stream
    return _wrap_io_open(filename, mode, encoding, errors), True
  File "lib/python/site-packages/click/_compat.py", line 511, in _wrap_io_open
    return io.open(file, mode, **kwargs)
FileNotFoundError: [Errno 2] No such file or directory: '/etc/bash'
```

Finally, when there are valid files that can be read, the encoding (defaulting to utf-8) is automatically handled:

```
>>> >>> click.open_file('/etc/bashrc')
<_io.TextIOWrapper name='/etc/bashrc' mode='r' encoding='UTF-8'>
>>> click.open_file('/etc/passwd')
```

Chapter 7: Turbocharging Click

It's as good a time to be writing code as ever, these days, a little bit of code goes a long way. Just a single function is capable of performing incredible things. Thanks to GPUs, Machine Learning, the Cloud, and Python, it's easy to create "turbocharged" command-line tools. Think of it as upgrading your code from using a basic internal combustion engine to a nuclear reactor. The basic recipe for the upgrade? One function, a sprinkle of compelling logic, and, finally, a decorator to route it to the command-line.

Writing and maintaining traditional GUI applications, web or desktop, is a Sisyphean task at best. It all starts with the best of intentions, but can quickly turn into a soul-crushing, time-consuming ordeal where you end up asking yourself why you thought becoming a programmer was a good idea in the first place. Why did you run that web framework setup utility that essentially automated a 1970's technology, the relational database, into a series of python files? The old Ford Pinto with the exploding rear gas tank has newer technology than your web framework. There has got to be a better way to make a living.

The answer is simple: stop writing web applications and start writing nuclear powered command-line tools instead. The turbocharged command-line tools that I share below focus on fast results via minimal lines of code. They can do things like learning from data (machine learning), make your code run 2,000 times faster, and, best of all, generate colored terminal output.

Here are the raw ingredients that will be used to make several solutions:

- Click Framework[46]
- Python CUDA Framework[47]
- Numba Framework[48]
- Scikit-learn Machine Learning Framework[49]

Using The Numba JIT (Just in time Compiler)

Python has a reputation for slow performance because it's fundamentally a scripting language. One way to get around this problem is to use the Numba JIT. Here's what that code looks like:

First, use a timing decorator to get a grasp on the runtime of your functions:

[46] https://click.palletsprojects.com/en/7.x/
[47] https://developer.nvidia.com/how-to-cuda-python
[48] http://numba.pydata.org/
[49] http://scikit-learn.org/dev/tutorial/machine_learning_map/index.html

```
def timing(f):
    @wraps(f)
    def wrap(*args, **kwargs):
        ts = time()
        result = f(*args, **kwargs)
        te = time()
        print(f'fun: {f.__name__}, args: [{args}, {kwargs}] took: {te-ts} sec')
        return result
    return wrap
```

Next, add a `numba.jit` decorator with the "nopython" keyword argument, and set to true. This step will ensure that the code will run by the JIT instead of the regular Python.

```
@timing
@numba.jit(nopython=True)
def expmean_jit(rea):
    """Perform multiple mean calculations"""

    val = rea.mean() ** 2
    return val
```

When you run it, you can see both a "jit" as well as a regular version run via the command-line tool:

```
$ Python nuclearcli.py jit-test
`"

```python
Running NO JIT
func:'expmean' args:[(array([[1.0000e+00, 4.2080e+05, 4.2350e+05, ..., 1.0543e+06, 1\
.0485e+06,
        1.0444e+06],
       [2.0000e+00, 5.4240e+05, 5.4670e+05, ..., 1.5158e+06, 1.5199e+06,
        1.5253e+06],
       [3.0000e+00, 7.0900e+04, 7.1200e+04, ..., 1.1380e+05, 1.1350e+05,
        1.1330e+05],
       ...,
       [1.5277e+04, 9.8900e+04, 9.8100e+04, ..., 2.1980e+05, 2.2000e+05,
        2.2040e+05],
       [1.5280e+04, 8.6700e+04, 8.7500e+04, ..., 1.9070e+05, 1.9230e+05,
        1.9360e+05],
       [1.5281e+04, 2.5350e+05, 2.5400e+05, ..., 7.8360e+05, 7.7950e+05,
        7.7420e+05]], dtype=float32),), {}] took: 0.0007 sec
```
```

```
$ python nuclearcli.py jit-test -jit
`"

```python
Running with JIT
func:'expmean_jit' args:[(array([[1.0000e+00, 4.2080e+05, 4.2350e+05, ..., 1.0543e+0\
6, 1.0485e+06,
        1.0444e+06],
       [2.0000e+00, 5.4240e+05, 5.4670e+05, ..., 1.5158e+06, 1.5199e+06,
        1.5253e+06],
       [3.0000e+00, 7.0900e+04, 7.1200e+04, ..., 1.1380e+05, 1.1350e+05,
        1.1330e+05],
       ...,
       [1.5277e+04, 9.8900e+04, 9.8100e+04, ..., 2.1980e+05, 2.2000e+05,
        2.2040e+05],
       [1.5280e+04, 8.6700e+04, 8.7500e+04, ..., 1.9070e+05, 1.9230e+05,
        1.9360e+05],
       [1.5281e+04, 2.5350e+05, 2.5400e+05, ..., 7.8360e+05, 7.7950e+05,
@click.option('--jit/--no-jit', default=False)
        7.7420e+05]], dtype=float32),), {}] took: 0.2180 sec
```

How does that work? Just a few lines of code allow for this simple toggle:

```
@cli.command()
def jit_test(jit):
    rea = real_estate_array()
    if jit:
        click.echo(click.style('Running with JIT', fg='green'))
        expmean_jit(rea)
    else:
        click.echo(click.style('Running NO JIT', fg='red'))
        expmean(rea)
```

In some cases, a JIT version could make code run thousands of times faster, but benchmarking is key. Another item to point out is the line:
```

```
click.echo(click.style('Running with JIT', fg='green'))
`"

This script allows for colored terminal output, which can be very helpful it creatin\
g sophisticated tools.

## Using a CUDA GPU

Another way to nuclear power your code is to run it straight on a GPU. *Note, this a\
ssumes you have access to an NVidia GPU*. This example requires you to run it on a m\
achine with a CUDA enabled. Here's what that code looks like:

```python
@cli.command()
def cuda_operation():
    """Performs Vectorized Operations on GPU"""

    x = real_estate_array()
    y = real_estate_array()

    print('Moving calculations to GPU memory')
    x_device = cuda.to_device(x)
    y_device = cuda.to_device(y)
    out_device = cuda.device_array(
        shape=(x_device.shape[0],x_device.shape[1]), dtype=np.float32)
    print(x_device)
    print(x_device.shape)
    print(x_device.dtype)

    print('Calculating on GPU')
    add_ufunc(x_device,y_device, out=out_device)

    out_host = out_device.copy_to_host()
    print(f'Calculations from GPU {out_host}')
```

It's useful to point out is that if the numpy array moves to the GPU, then a vectorized function does the work on the GPU. After that work completes, then the data transfers from the GPU. Using a GPU could be a significant improvement to the code, depending on its running. The output from the command-line tool shows below:
```

```
$ Python nuclearcli.py cuda-operation
`"

```python
Moving calculations to GPU memory

(10015, 259)
float32
Calculating on GPU
Calculcations from GPU [[2.0000e+00 8.4160e+05 8.4700e+05 ... 2.1086e+06 2.0970e+06 \
2.0888e+06]
 [4.0000e+00 1.0848e+06 1.0934e+06 ... 3.0316e+06 3.0398e+06 3.0506e+06]
 [6.0000e+00 1.4180e+05 1.4240e+05 ... 2.2760e+05 2.2700e+05 2.2660e+05]
 ...
 [3.0554e+04 1.9780e+05 1.9620e+05 ... 4.3960e+05 4.4000e+05 4.4080e+05]
 [3.0560e+04 1.7340e+05 1.7500e+05 ... 3.8140e+05 3.8460e+05 3.8720e+05]
 [3.0562e+04 5.0700e+05 5.0800e+05 ... 1.5672e+06 1.5590e+06 1.5484e+06]]
```

Running True Multi-Core Multithreaded Python using Numba

One common performance problem with Python is the lack of true, multi-threaded performance. This step also can be fixed with Numba. Here's an example of some basic operations:

```
@timing
@numba.jit(parallel=True)
def add_sum_threaded(rea):
    """Use all the cores"""

    x,_ = rea.shape
    total = 0
    for _ in numba.prange(x):
        total += rea.sum()
        print(total)

@timing
def add_sum(rea):
    """traditional for loop"""

    x,_ = rea.shape
    total = 0
```
```

```
    for _ in numba.prange(x):
        total += rea.sum()
        print(total)

@cli.command()
@click.option('--threads/--no-jit', default=False)
def thread_test(threads):
    rea = real_estate_array()
    if threads:
        click.echo(click.style('Running with multicore threads', fg='green'))
        add_sum_threaded(rea)
    else:
        click.echo(click.style('Running NO THREADS', fg='red'))
        add_sum(rea)
```

Note that the parallel version's critical difference is that it uses `@numba.jit(parallel=True)` and `numba.prange` to spawn threads for iteration. Look at the picture below, all of the CPUs maxes out on the machine, but when almost the same code runs without the parallelization, it only uses a core.

```
$ Python nuclearcli.py thread-test
`"

```bash
$ python nuclearcli.py thread-test --threads
`"

## Integrate K-Means Cluster (Unsupervised Machine Learning)

One more powerful thing that can accomplish in a command-line tool is machine learni\
ng. In the example below, a KMeans clustering function creates with just a few lines\
 of code. This step clusters a pandas DataFrame into three clusters.

```python
def kmeans_cluster_housing(clusters=3):
    """Kmeans cluster a dataframe"""
    url = 'https://raw.githubusercontent.com/noahgift/socialpowernba/master/data/nba\
_2017_att_val_elo_win_housing.csv'
    val_housing_win_df =pd.read_csv(url)
    numerical_df =(
        val_housing_win_df.loc[:,['TOTAL_ATTENDANCE_MILLIONS', 'ELO',
        'VALUE_MILLIONS', 'MEDIAN_HOME_PRICE_COUNTY_MILLIONS']]
    )
    #scale data
```

```
    scaler = MinMaxScaler()
    scaler.fit(numerical_df)
    scaler.transform(numerical_df)
    #cluster data
    k_means = KMeans(n_clusters=clusters)
    kmeans = k_means.fit(scaler.transform(numerical_df))
    val_housing_win_df['cluster'] = kmeans.labels_
    return val_housing_win_df
```

The cluster number can be changed by passing in another number (as shown below) using click:

```
@cli.command()
@click.option('--num', default=3, help='number of clusters')
def cluster(num):
    df = kmeans_cluster_housing(clusters=num)
    click.echo('Clustered DataFrame')
    click.echo(df.head())
```

Finally, the output of the Pandas DataFrame with the cluster assignment shows below. Note, it has a cluster assignment as a column now.

```
$ Python -W nuclearcli.py cluster
`"

```python
Clustered DataFrame    0    1    2    3    4
TEAM    Chicago Bulls    Dallas Mavericks    Sacramento Kings    Miami Heat    Toron\
to Raptors
GMS    41    41    41    41    41
PCT_ATTENDANCE    104    103    101    100    100
WINNING_SEASON    1    0    0    1    1
......
COUNTY    Cook    Dallas    Sacremento    Miami-Dade    York-County
MEDIAN_HOME_PRICE_COUNTY_MILLIONS    269900.0    314990.0    343950.0    389000.0   \
 390000.0
COUNTY_POPULATION_MILLIONS    5.20    2.57    1.51    2.71    1.10
cluster    0    0    1    0    0
```
```

```
$ python -W nuclearcli.py cluster --num 2
`"

```python
Clustered DataFrame    0    1    2    3    4
TEAM    Chicago Bulls    Dallas Mavericks    Sacramento Kings    Miami Heat    Toron\
to Raptors
GMS    41    41    41    41    41
PCT_ATTENDANCE    104    103    101    100    100
WINNING_SEASON    1    0    0    1    1
......
COUNTY    Cook    Dallas    Sacremento    Miami-Dade    York-County
MEDIAN_HOME_PRICE_COUNTY_MILLIONS    269900.0    314990.0    343950.0    389000.0   \
 390000.0
COUNTY_POPULATION_MILLIONS    5.20    2.57    1.51    2.71    1.10
cluster    1    1    0    1    1
```

This chapter's goal is to show how simple command-line tools can be a great alternative to heavy web frameworks. In under 200 lines of code, you're now able to create a command-line tool that involves GPU parallelization, JIT, core saturation, as well as Machine Learning. The examples I shared above are just the beginning of upgrading your developer productivity to nuclear power, and I hope you'll use these programming tools to help build the future.

Many of the most powerful things happening in the software industry occur with functions: distributed computing, machine learning, cloud computing (functions as a service), and GPU based programming are all great examples. The natural way of controlling these functions is a decorator-based command-line tool, not clunky 20th Century clunky web frameworks. The Ford Pinto parked in a garage, and you're driving a shiny new "turbocharged" command-line interface that maps powerful yet simple functions to logic using the Click framework.
```

Chapter 8: Integrate Click with the Cloud

Cloud Computing is a strong use case for command-line tool development. The essence of the command-line is minimalism. Build a tool to solve the problem. Then build another tool to solve another problem. There is no "hotter" skill than cloud computing.

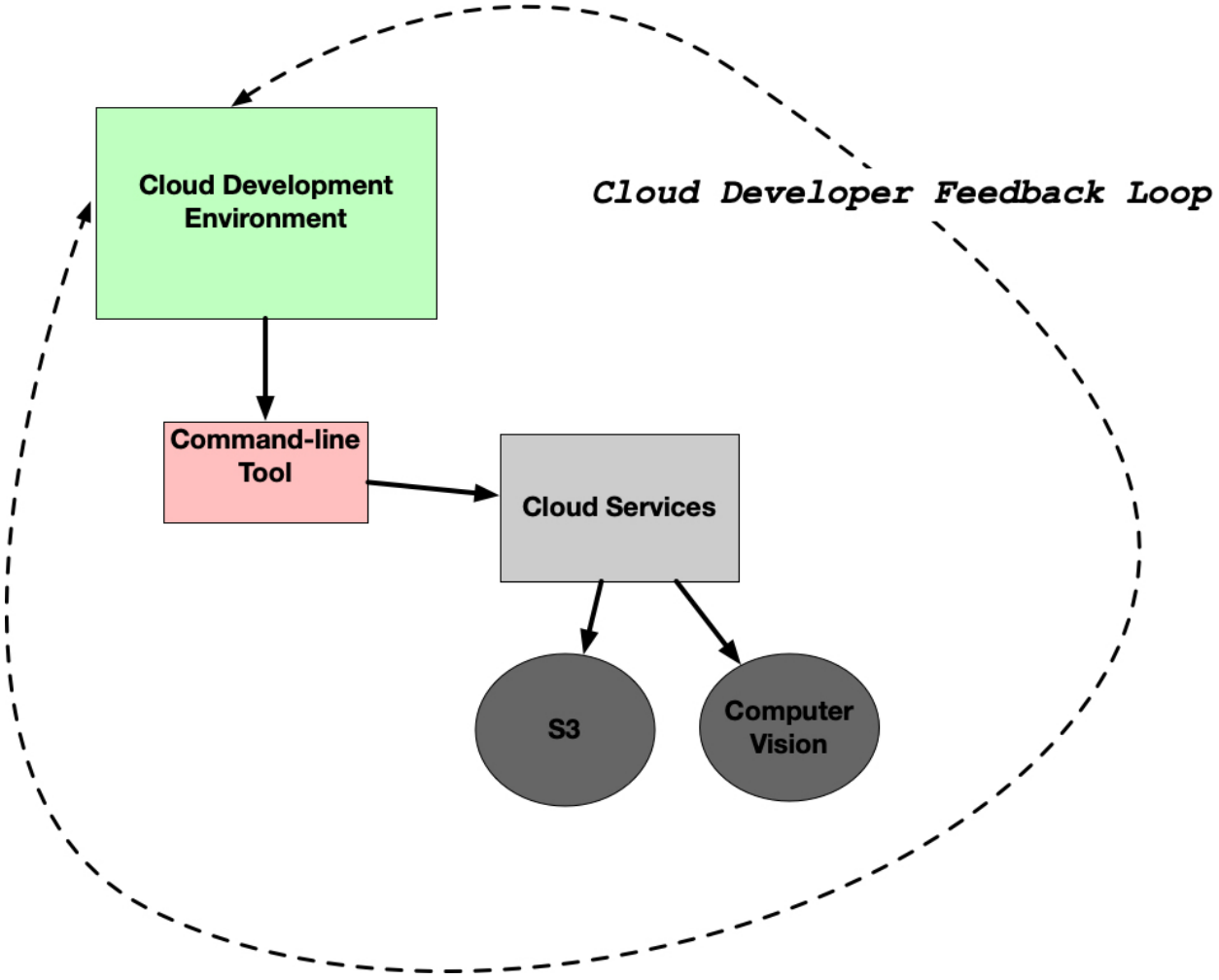
One way to think about cloud computing is a new type of operating system. With the Unix operating system, small tools like `awk`, `sed` and `cut` serve to allow a user to glue together solutions in `bash`. Similarly, a `python` command-line tool in the cloud can "glue" cloud services together. A well crafted command-line tool can be the simplest and most effective way to solve a problem in the cloud.

Cloud Developer Workflow

The ideal location to build a command-line tool for cloud computing isn't your laptop! All of the cloud providers have gravitated toward cloud development environments. This step allows the developer to build code in the background that it runs in.

Let's take a look at how this plays out in practice in the following diagram. A developer spins up a cloud-based development like AWS Cloud9[50]. Next, they develop a command-line tool that interacts with a cloud service.

[50] https://aws.amazon.com/cloud9/



cloud-cli-workflow

Why is this workflow so powerful?

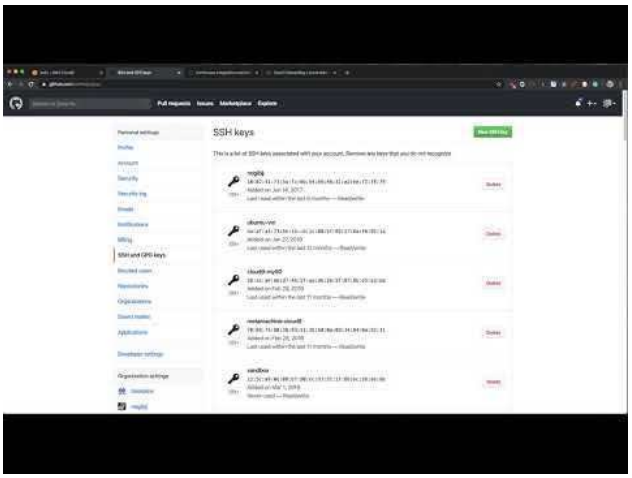
- Development takes place in the environment where the code runs (not your laptop).
- Deep integrations to the cloud development environment are included.
- A command-line tool is often the most efficient way to interact with a cloud service like computer vision or object storage
- Python itself is the ideal language to glue together solutions in the cloud. The cloud builds in a variety of high-performance styles that have better performance characteristics than Python. Still, Python can build on top of these solutions by orchestrating the API calls.

Using Cloud-based development environments

Just as many environments are Linux, it is also true that most deployment environments are in the cloud. Three of the largest cloud providers are: AWS[51], Azure[52] and GCP[53]. To write software that deploys on Cloud Computing environments, it often makes sense to write, test, and build code in cloud-specific development environments. Let's discuss two of these environments.

AWS Cloud9

The AWS Cloud9 Environment[54] is an IDE that allows a user to write, run, and debug code (including serverless code in Python) in the AWS cloud. This step simplifies many workflows, including security and network bandwidth. You can watch a walkthrough video here that creates a new AWS Cloud9 environment.



View this Video at https://www.youtube.com/watch?v=4SIFF1PAMbw[55].

Setup CI Pipeline with AWS Cloud9 and CircleCI

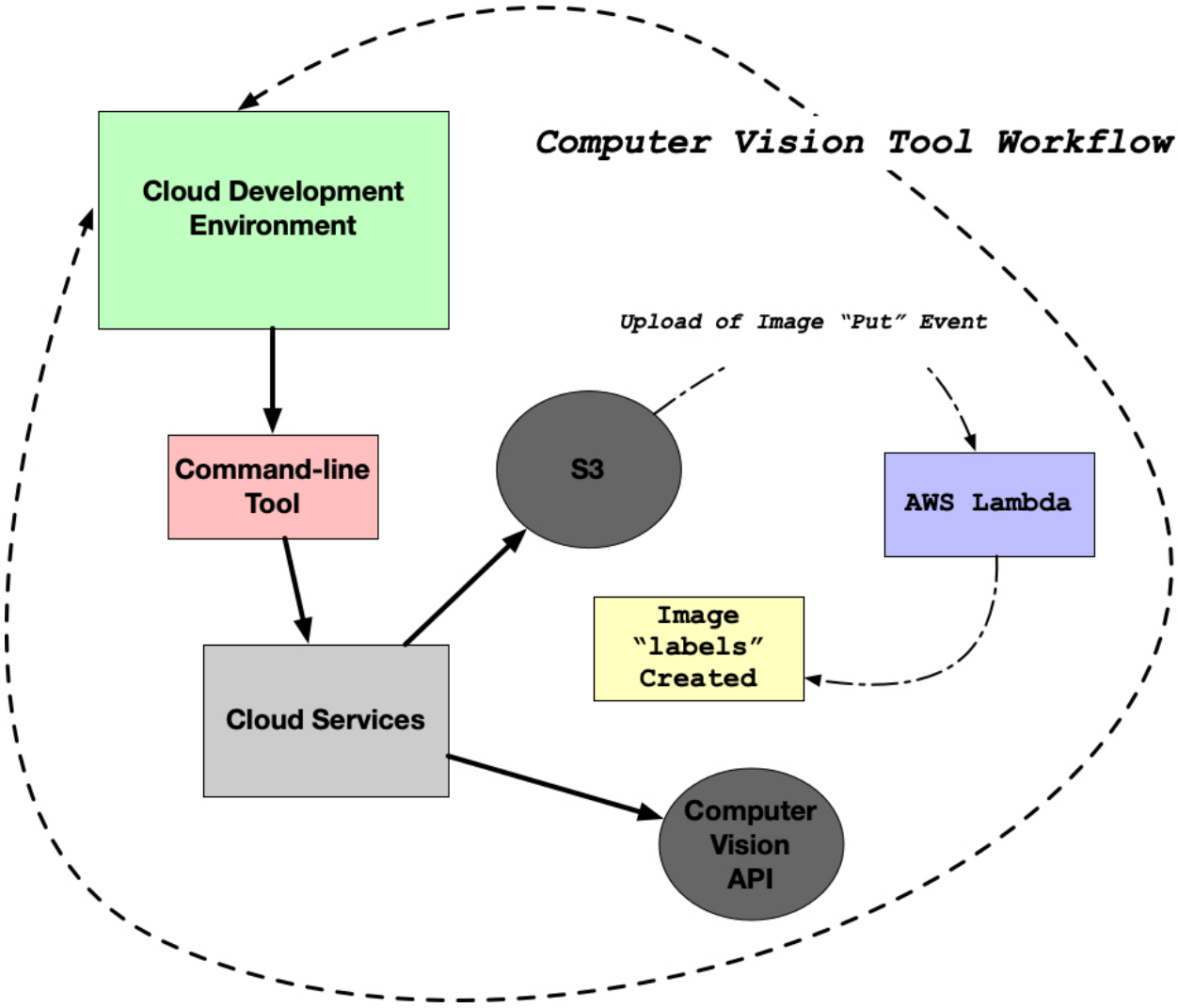
Build a Computer Vision Tool with AWS Boto3

How would a cloud developer go about using the power of command-line tools to develop a full-fledged computer vision application that triggers API calls that detect image labels? The following diagram shows the workflow.

- Create a cloud-based development environment
- Build a command-line tool that tests out the concept

[51] https://aws.amazon.com/
[52] https://azure.microsoft.com/en-us/
[53] https://cloud.google.com/
[54] https://aws.amazon.com/cloud9/

- Create a lambda function that triggers this same computer vision logic upon upload of an S3 image.



computer-vision-flow

This image of my dog will work throughout the examples.

dog2

First, a command-line tool using click accepts a bucket and a file name and passes it into the AWS Rekognition API[56].

```
#!/usr/bin/env python
import click
import boto3


@click.command()
@click.option("--bucket", prompt="S3 Bucket", help="This is the S3 Bucket")
@click.option(
    "--name",
    prompt="this is the name of the image",
    help="Pass in the name:  i.e. husky.png",
)
def labels(bucket, name):
    """This takes an S3 bucket and a image name"""

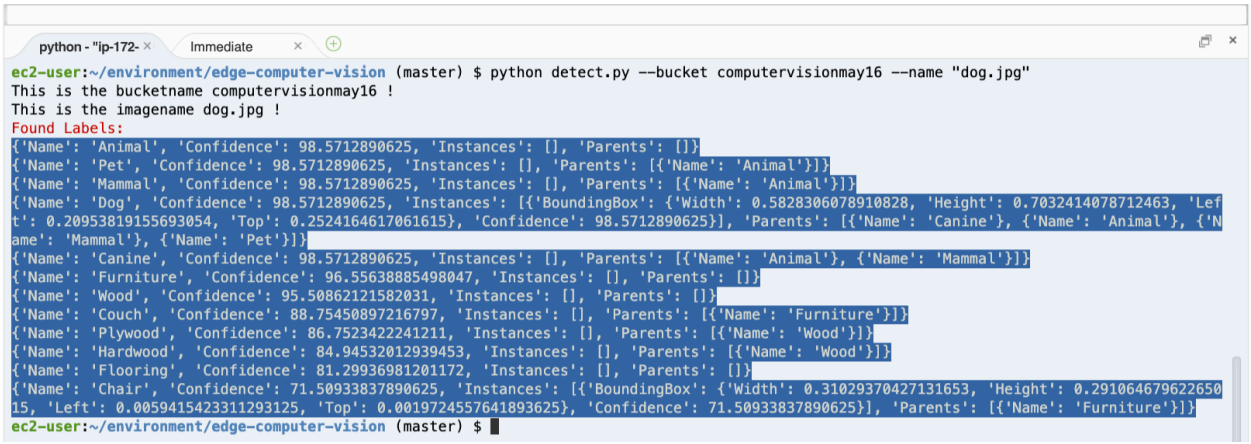
    print(f"This is the bucketname {bucket} !")
    print(f"This is the imagename {name} !")
    rekognition = boto3.client("rekognition")
    response = rekognition.detect_labels(
        Image={"S3Object": {"Bucket": bucket, "Name": name,}},
    )
    labels = response["Labels"]
    click.echo(click.style("Found Labels:", fg="red"))
    for label in labels:
        click.echo(click.style(f"{label}", bg="blue", fg="white"))


if __name__ == "__main__":
    # pylint: disable=no-value-for-parameter
    labels()
```

When this command-line tool runs, it generates the labels for the image of my dog. Notice the power of using the colored output to differentiate between different components of the command-line tool.

```
python detect.py --bucket computervisionmay16 --name "dog.jpg"
```

[56] https://aws.amazon.com/rekognition/



Screen Shot 2020-05-17 at 3 09 33 PM

After this proof of concept has proved out the workflow, a good intermediate step is to take the logic and put it into an AWS Lambda function. This Lambda function accepts a JSON payload.

The payload is `bucket` and a `name`.

```
{
  "bucket": "computervisionmay16",
  "name": "dog.jpg"
}
```

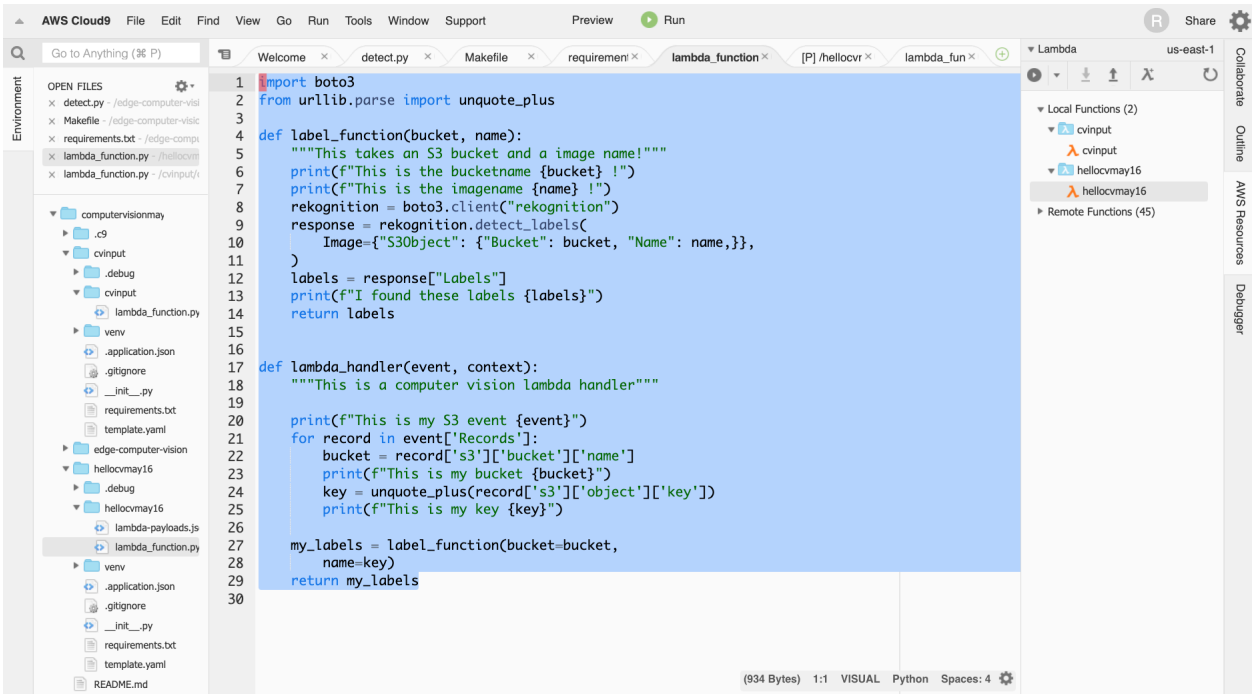
Next, the Lambda function takes it and returns a response with the labels for the object.

```
import boto3
import json


def lambda_handler(event, context):
    if "body" in event:
        event = json.loads(event["body"])
    image = event["image"]
    rekognition = boto3.client("rekognition")
    response = rekognition.detect_labels(
        Image={"S3Object": {"Bucket": "demoapril10", "Name": image,}},
    )

    print(response)
    return {"statusCode": 200, "body": json.dumps(response)}
```

A more sophisticated lambda function would not need a manual API call. Instead, it responds to an event. This step can be tested in the Cloud9 environment as well.



Screen Shot 2020-05-17 at 2 57 58 PM

The big takeaway is that the logic can again work. The `label_function` does the main work. The `lambda_handler` parses the `event['Records']` payload, which is the `PUT` event that results from an image stored in Amazon S3.

```
import boto3
from urllib.parse import unquote_plus

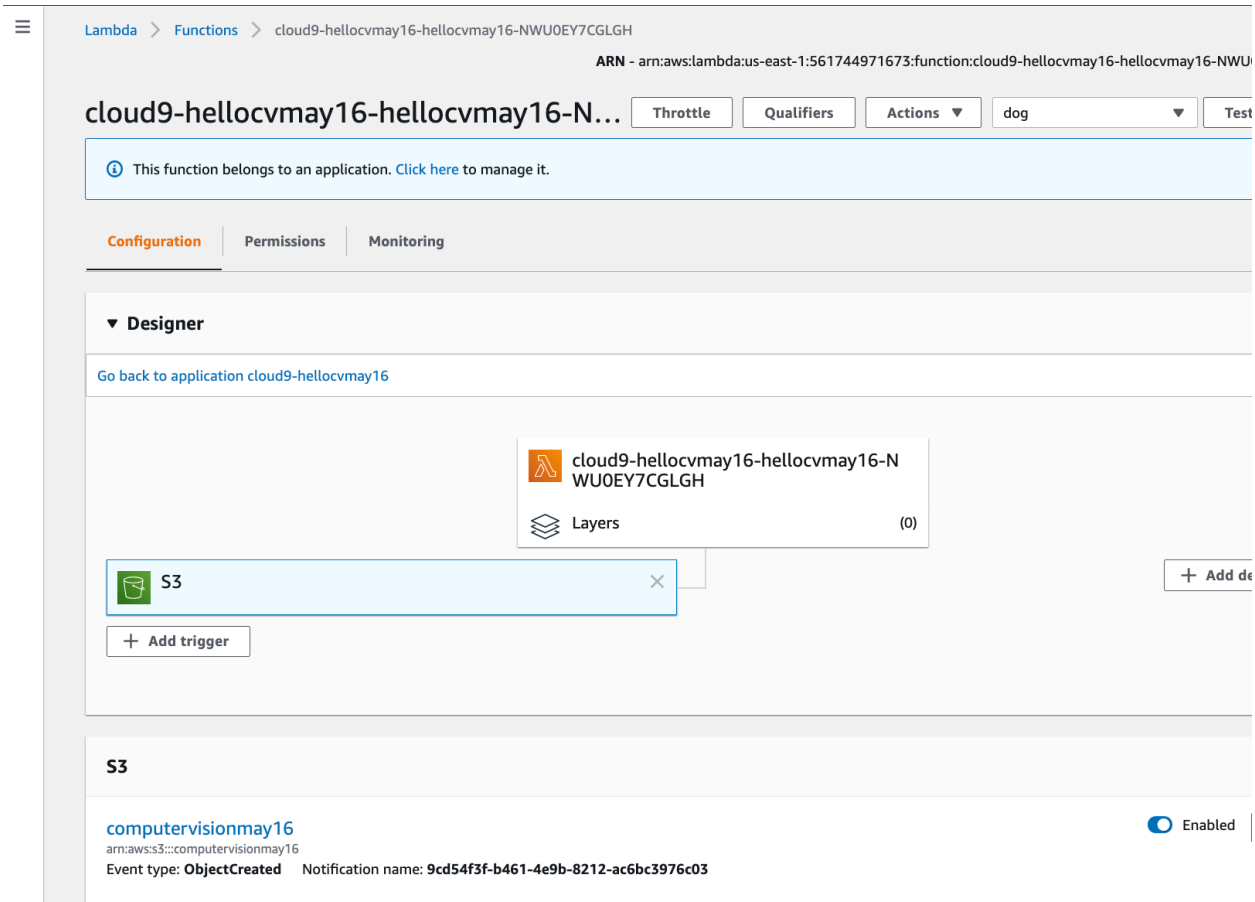
def label_function(bucket, name):
    """This takes an S3 bucket and a image name!"""
    print(f"This is the bucketname {bucket} !")
    print(f"This is the imagename {name} !")
    rekognition = boto3.client("rekognition")
    response = rekognition.detect_labels(
        Image={"S3Object": {"Bucket": bucket, "Name": name,}},
    )
    labels = response["Labels"]
    print(f"I found these labels {labels}")
    return labels


def lambda_handler(event, context):
    """This is a computer vision lambda handler"""
```

```
    print(f"This is my S3 event {event}")
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        print(f"This is my bucket {bucket}")
        key = unquote_plus(record['s3']['object']['key'])
        print(f"This is my key {key}")

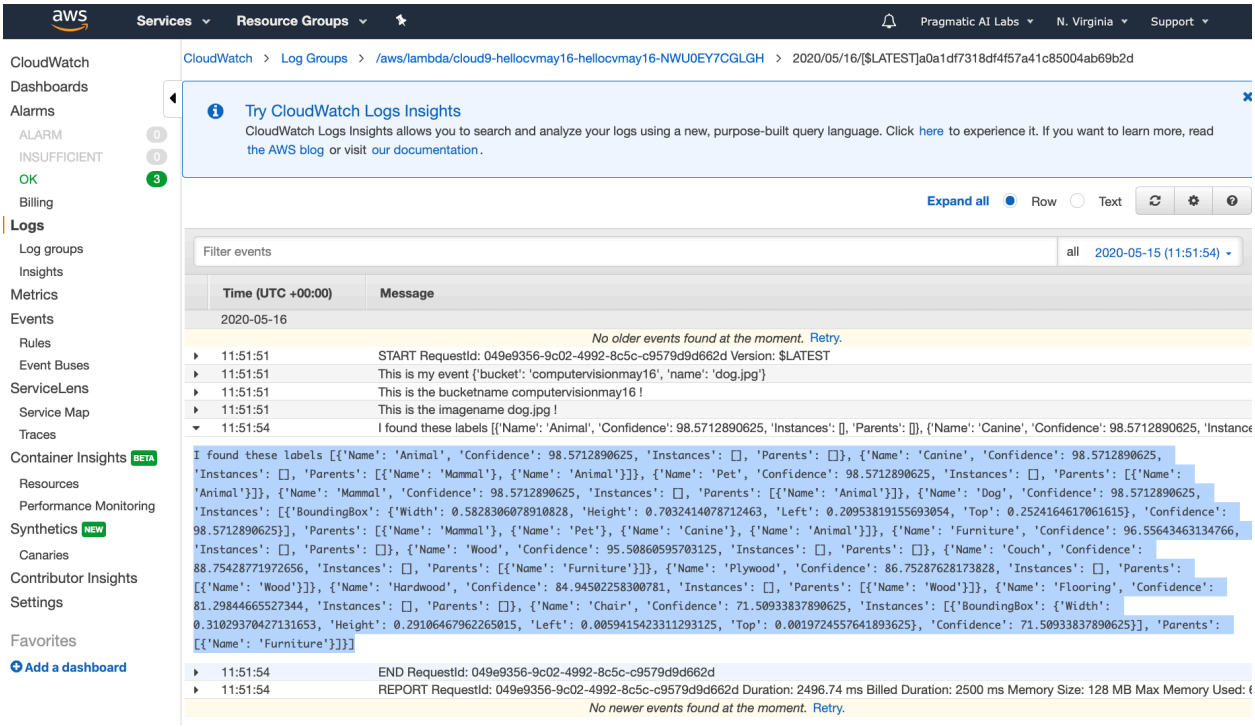
    my_labels = label_function(bucket=bucket,
        name=key)
    return my_labels
```

You can see the trigger set up in the AWS Lambda designer.



Screen Shot 2020-05-17 at 3 00 03 PM

Finally, the S3 event generates a call to the AWS Lambda function. The cloud watch logs show the label events.



Screen Shot 2020-05-17 at 3 04 05 PM

What are the next steps? The lambda function could store data in DynamoDB, or pass the results to another lambda function via AWS Step Functions.

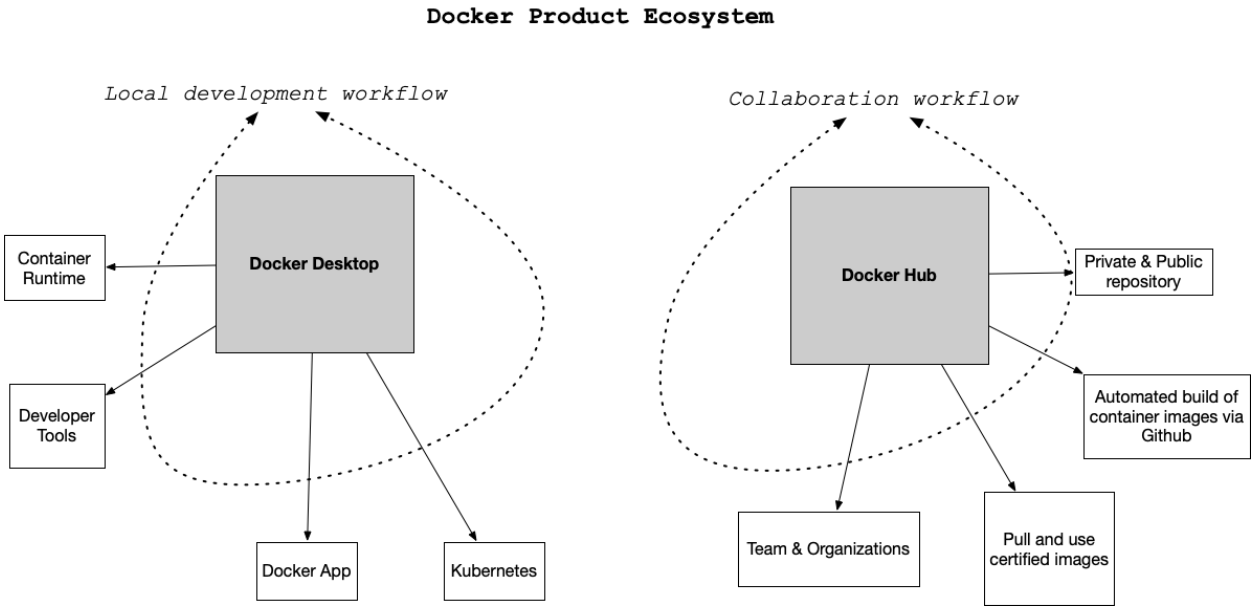
Chapter 9: Case Studies

This chapter contains a collection of recipes that have come up through the years in building command-line tools.

Distribute a containerized click application to DockerHub

Getting started with Docker

There are two primary components of Docker: Docker Desktop[57] and Docker Hub[58].



docker-ecosystem

Docker Desktop Overview

The desktop application contains the container runtime, which allows containers to execute. The Docker App itself orchestrates the local development workflow, including the ability to use Kubernetes[59], which is an open-source system for managing containerized applications that came out of Google.

[57] https://www.docker.com/products/docker-desktop
[58] https://www.docker.com/products/docker-hub
[59] https://github.com/kubernetes/kubernetes

Docker Hub Overview

So what is Docker Hub, and what problem does it solve? Just as the `git`[60] source code ecosystem has local developer tools like vim[61], emacs[62], Visual Studio Code[63] or XCode[64] that work with it, Docker Desktop works with Docker containers and allows for local use and development.

When collaborating with `git` outside of the local environment, developers often use platforms like Github[65] or Gitlab[66] to communicate with other parties and share code. Docker Hub[67] works similarly. Docker Hub allows developers to share docker containers that can serve as a base image for building new solutions.

These base images can be built by experts and certified to be high quality: i.e., the official Python developers have a base image[68]. This step allows a developer to leverage the expertise of the real expert on a particular software component and improve their container's overall quality. This process is a similar concept to using a library developed by another developer versus writing it yourself.

Why Docker Containers vs. Virtual Machines?

What is the difference between a container and a virtual machine? Here is a breakdown:

- Size: Containers are much smaller than Virtual Machines (VM) and run as isolated processes versus virtualized hardware. VMs can be GBs, while containers can be MBs.
- Speed: Virtual Machines can be slow to boot and take minutes to launch. A container can spawn much more quickly, typically in seconds.
- Composability: Containers programmatically build. They are defined as source code in an Infrastructure as Code project (IaC). Virtual Machines are often replicas of a manually created system. Containers make IaC workflows possible because they are defined as a file and checked into source control alongside the project's source code.

Real-World Examples of Containers

What problem does Docker format containers[69] solve? In a nutshell, the operating system runtime can package along with the code, which explains a particularly complicated problem with a long history. There is a famous meme that goes, "It works on my machine!". While this is often told as a joke to illustrate the complexity of deploying software, it is also true. Containers solve this exact problem. If the code works in a box, then the container configuration can be checked in as code.

[60] https://git-scm.com/
[61] https://www.vim.org/
[62] https://www.gnu.org/software/emacs/
[63] https://code.visualstudio.com/
[64] https://developer.apple.com/xcode/
[65] https://github.com/
[66] https://about.gitlab.com/
[67] https://hub.docker.com/
[68] https://hub.docker.com/_/python
[69] https://docs.docker.com/engine/docker-overview/

Another way to describe this concept is that the actual Infrastructure runs as "code." This process is called IaC (Infrastructure as Code).

Here are a few specific examples:

Developer Shares Local Project

A developer can work on a web application that uses `flask` (a popular Python web framework). The Docker container file handles the installation and configuration of the underlying operating system. Another team member can check out the code and use `docker run` to run the project. This process eliminates the multi-day problem of configuring a laptop correctly to run a software project.

Data Scientist shares Jupyter Notebook with a Researcher at another University

A data scientist working with jupyter[70] style notebooks wants to share a complex data science project with multiple dependencies on C, Fortran, R, and Python code. They package up the runtime as a Docker container and eliminate the back and forth over several weeks that occurs when sharing a project like this.

A Machine Learning Engineer Load Tests a Production Machine Learning Model

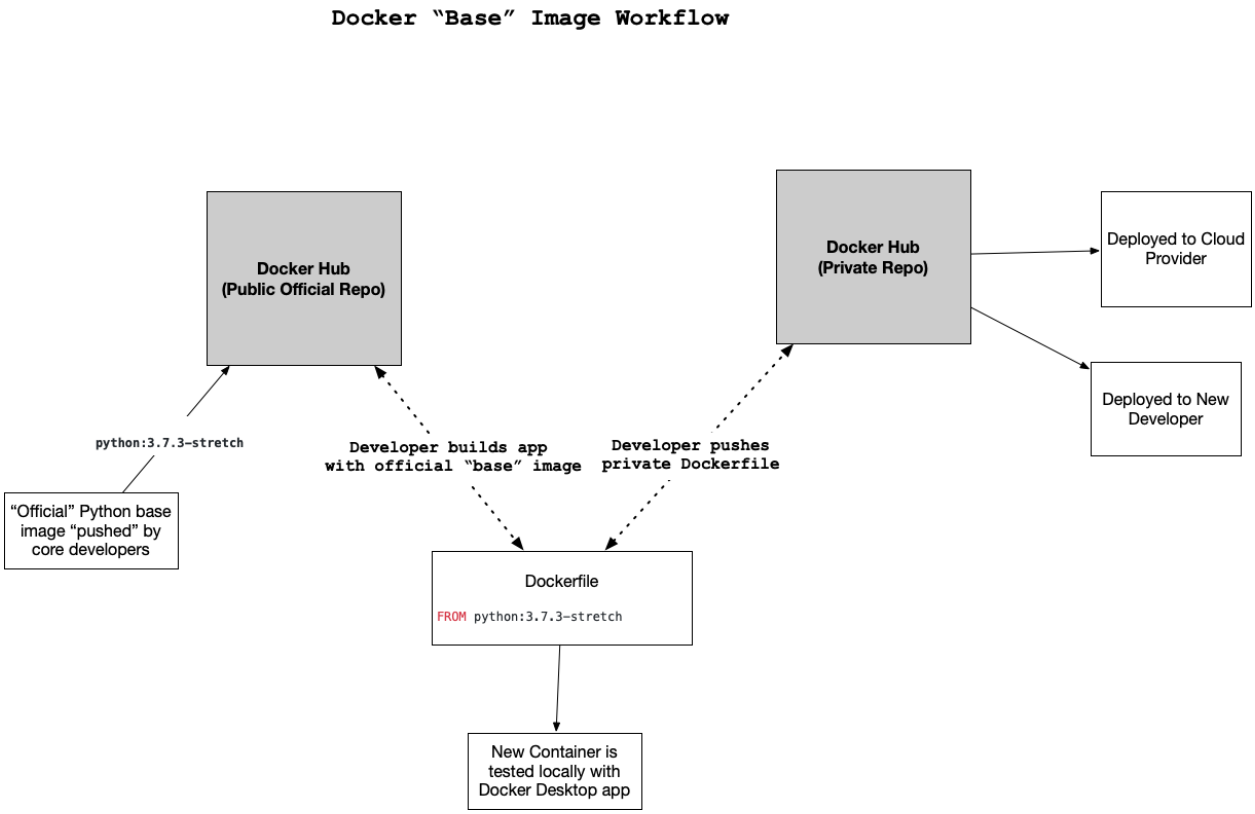
A Machine learning engineer needs to take a new model and deploy it to production. Previously, they were concerned about how to accurately test the new model's accuracy before committing to it. The model recommends products to pay customers and, if it is inaccurate, costs the company a lot of money. By using containers, it is possible to deploy the model to a fraction of the customers, only 10%, and if there are problems, it can be quickly reverted. If the model performs well, it can soon replace the existing models.

Running Docker Containers

Using "base" images

One of the advantages of the Docker workflow for developers is the ability to use certified containers from the "official" development teams. In this diagram, a developer uses the official Python base image developed by the core Python developers. This step is accomplished by the `FROM` statement, which loads in a previously created container image.

[70] https://jupyter.org/



docker-base-image

As the developer changes to the `Dockerfile`, they test locally and then push the changes to a private Docker Hub repo. After this, the changes can be used by a deployment process to a Cloud or by another developer.

Common Issues Running a Docker Container

There are a few common issues that crop up when starting a container or building one for the first time. Let's walk through each problem and then present a solution for them.

- What goes in a `Dockerfile` if you need to write to the host filesystem[71]? In the following example, the `docker volume` command creates a volume, which mounts to the container.

[71] https://docs.docker.com/storage/volumes/

```
>  /tmp docker volume create docker-data
docker-data
>  /tmp docker volume ls
DRIVER              VOLUME NAME
local               docker-data
>  /tmp docker run -d \
  --name devtest \
  --mount source=docker-data,target=/app \
  ubuntu:latest
6cef681d9d3b06788d0f461665919b3bf2d32e6c6cc62e2dbab02b05e77769f4
```

- How do you configure logging[72] for a Docker container?

You can configure logging for a Docker container by selecting the type of log driver, in this example, `json-file` and whether it is blocking or non-blocking. This example shows a configuration that uses `json-file` and `mode=non-blocking` for an ubuntu container. The `non-blocking` mode ensures that the application won't fail in a non-deterministic manner. Make sure to read the Docker logging guide[73] on different logging options.

```
>  /tmp docker run -it --log-driver json-file --log-opt mode=non-blocking ubuntu
root@551f89012f30:/#
```

- How do you map ports to the external host?

The Docker container has an internal set of ports that must be exposed to the host and mapped[74]. One of the easiest ways to see what ports present to the host is by running the `docker port <container name>` command. Here is an example of what that looks like against a `foo` named container.

```
$ docker port foo
7000/tcp -> 0.0.0.0:2000
9000/tcp -> 0.0.0.0:3000
```

What about actually mapping the ports? You can do that using the `-p` flag, as shown. You can read more about Docker `run` flags here[75].

```
$ docker run -p 127.0.0.1:80:9999/tcp ubuntu bash
```

- What about configuring Memory, CPU, and GPU?

You can configure docker `run` to accept flags for setting Memory, CPU, and GPU. You can read more about it here[76] in the official documentation. Here is a brief example of setting the CPU.

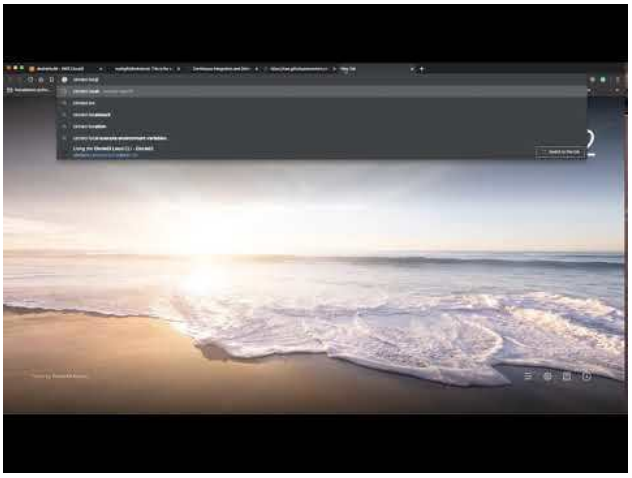
[72] https://docs.docker.com/config/containers/logging/configure/
[73] https://docs.docker.com/config/containers/logging/configure/
[74] https://docs.docker.com/engine/reference/commandline/port/
[75] https://docs.docker.com/engine/reference/commandline/run/
[76] https://docs.docker.com/config/containers/resource_constraints/

```
docker run -it --cpus=".25" ubuntu /bin/bash
```

This step tells this container to use at max only 25% of the CPU every second.

Build containerized application from Zero on AWS Cloud9

Screencast



View this Video at https://youtu.be/WVifwRIwSmo[77].

Docker Python from Zero in Cloud9!

- Docker Project Source Code[78]

1. Launch AWS Cloud9
2. Create Github repo
3. Create ssh keys and upload to Github
4. Git clone
5. Create a structure
6. Create a local python virtual environment and source *MUST HAVE!*:

```
$ python 3 -m venv ~/.dockerproj && source ~/.dockerproj/bin/activate
```

- `Dockerfile`

[78] https://github.com/noahgift/dockerproj

```
FROM Python:3.7.3-stretch

# Working Directory
WORKDIR /app

# Copy source code to working directory
COPY . app.py /app/

# Install packages from requirements.txt
# hadolint ignore=DL3013
RUN pip install --upgrade pip &&\
    pip install --trusted-host pypi.python.org -r requirements.txt
```

- requirements.txt
- Makefile

```
setup:
    python3 -m venv ~/.dockerproj

install:
    pip install --upgrade pip &&\
        pip install -r requirements.txt

test:
    #python -m pytest -vv --cov=myrepolib tests/*.py
    #python -m pytest --nbval notebook.ipynb

validate-circleci:
    # See https://circleci.com/docs/2.0/local-cli/#processing-a-config
    circleci config process .circleci/config.yml

run-circleci-local:
    # See https://circleci.com/docs/2.0/local-cli/#running-a-job
    circleci local execute


lint:
    hadolint Dockerfile
    pylint --disable=R,C,W1203 app.py

all: install lint test
```

- `app.py`

6. Install hadolint
(you may want to become root: i.e. `sudo su -`.
run this command then exit by typing `exit`.)

```
wget -O /bin/hadolint \
  https://github.com/hadolint/hadolint/releases/download/\
  v1.17.5/hadolint-Linux-x86_64 &&\
                chmod +x /bin/hadolint
```

7. Create cirleci config

```
# Python CircleCI 2.0 configuration file
#
# Check https://circleci.com/docs/2.0/language-python/ for more details
#
version: 2
jobs:
  build:
    docker:
    # Use the same Docker base as the project
      - image: python:3.7.3-stretch

    working_directory: ~/repo

    steps:
      - checkout

      # Download and cache dependencies
      - restore_cache:
          keys:
            - v1-dependencies-{{ checksum "requirements.txt" }}
            # fallback to using the latest cache if no exact match is found
            - v1-dependencies-

      - run:
          name: install dependencies
          command: |
            python3 -m venv venv
            . venv/bin/activate
            make install
```

```
            # Install hadolint
            wget -O /bin/hadolint\
              https://github.com/hadolint/hadolint/releases/download/v1.17.5/hadolin\
t-Linux-x86_64 &&\
                chmod +x /bin/hadolint

      - save_cache:
          paths:
            - ./venv
          key: v1-dependencies-{{ checksum "requirements.txt" }}

      # run lint!
      - run:
          name: run lint
          command: |
            . venv/bin/activate
            make lint
```

8. Install local circleci[79] (optional)
9. setup requirements.txt

```
pylint
click
```

10. Create app.py

```
#!/usr/bin/env python
import click

@click.command()
def hello():
    click.echo('Hello World!')

if __name__ == '__main__':
    hello()
```

11. Run in container

```
$ docker build --tag=app .
```

[79] https://circleci.com/docs/2.0/local-cli/

```
$ docker run -it app bash
```

12. Test app in shell

REMEMBER Virtualenv:

```
$ python3 -m venv ~/.dockerproj && source ~/.dockerproj/bin/activate
```

And then run `python app.py` or `chmod +x && ./app.py`

13. Test local circleci and local make lint and then configure circleci.

```
ec2-user:~/environment $ sudo su -
[root@ip-172-31-65-112 ~]# curl -fLSs https://circle.ci/cli | bash
Starting installation.
Installing CircleCI CLI v0.1.5879
Installing to /usr/local/bin
/usr/local/bin/circleci
```

14. Setup Docker Hub Account[80] and deploy it!
15. To deploy you will need something like this (bash script)

```
#!/usr/bin/env bash
# This tags and uploads an image to Docker Hub

#Assumes this is built
#docker build --tag=app .


dockerpath="noahgift/app"

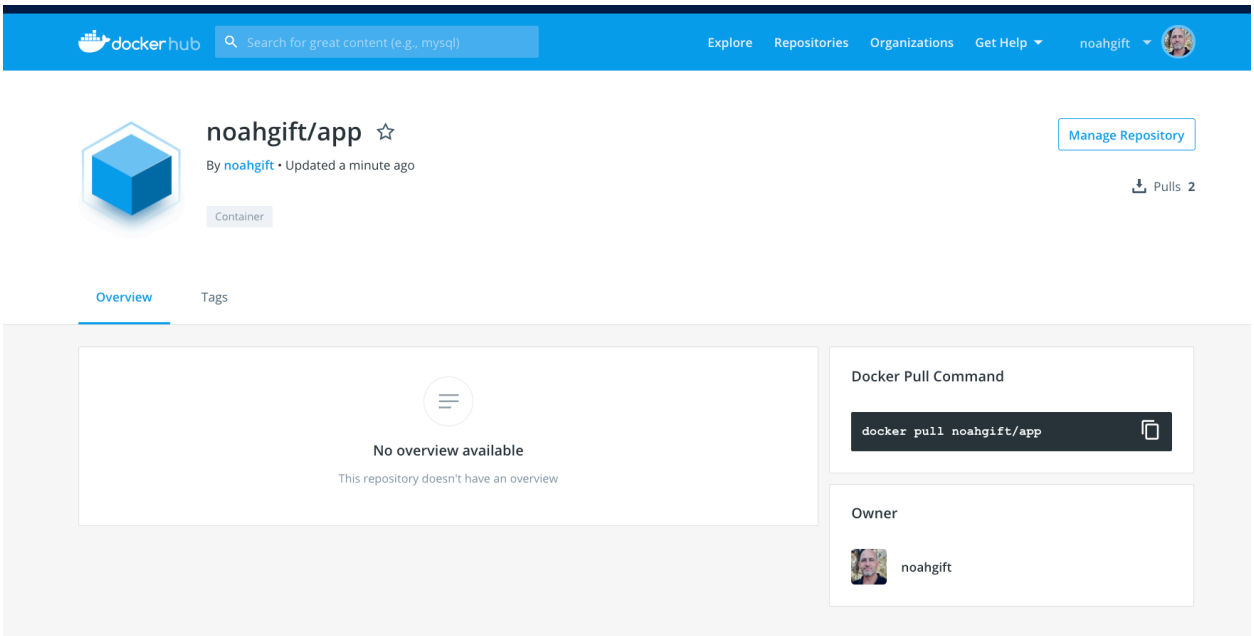
# Authenticate & Tag
echo "Docker ID and Image: $dockerpath"
docker login &&\
    docker image tag app $dockerpath

# Push Image
docker image push $dockerpath
```

Any person can pull now:

[80] https://docs.docker.com/docker-hub/

```
$ docker pull noahgift/app
```



Screen Shot 2020-02-04 at 3 51 15 PM

Exercise

- Topic: Create Hello World Container in AWS Cloud9 and Publish to Docker Hub
- Estimated time: 20-30 minutes
- People: Individual or Final Project Team
- Slack Channel: #noisy-exercise-chatter
- Directions:
  * Part A: Build a hello world Docker container in AWS Cloud9 that uses the official Python base image. You can use the sample command-line tools[81] in this repository for ideas.
  * Part B: Create an account on Docker Hub and publish there
  * Part C: Share your Docker Hub container in slack
  * Part D: Pull down another students container and run it
  * (Optional for the ambitious): Containerize a flask application and publish

Converting a Command-line tool to a Web Service

One item to point out is the feasibility of building command-line tools that get converted to web applications. You can see older idea I toyed with here: Adapt Project[82]

Newer frameworks like chalice[83] may be a secure fit for dual cli + web projects.

[81] https://github.com/noahgift/python-devops-course
[82] https://github.com/noahgift/Adapt/blob/master/serve.py
[83] https://github.com/aws/chalice

Documenting your Project with Sphinx

Sphinx is a tool allowing developers to write documentation in plain text for natural output generation in formats meeting varying needs. This step becomes helpful when using a Version Control System to track changes. Plain-text documentation is also useful for collaborators across different systems. Plain text is one of the most portable formats currently available.

Although Sphinx is written in Python and initially created for the Python language documentation, it is not necessarily language-centric and, in some cases, not even programmer-specific. There are many uses for Sphinx, such as writing entire books and even websites.

Think of Sphinx as a documentation framework abstracting the tedious parts. It offers automatic functionality to solve common problems like title indexing and code highlighting (if showing code examples) with proper syntax highlighting.

Sphinx uses reStructuredText markup syntax (with some additions) to provide document control. You probably already know quite a bit about the language required to be proficient in Sphinx if you have ever written plain-text files.

The markup allows the definition and structure of text for proper output.

```
This is a Title
===============
That has a paragraph about the main subject and sets when the '='
is at least the same length of the title itself.

Subject Subtitle
----------------
Subtitles are set with '-' and are required to have the same length
of the subtitle itself, just like titles.

Lists can be unnumbered like:

 * Item Foo
 * Item Bar

Or automatically numbered:

 #. Item 1
 #. Item 2

Inline Markup
-------------
Words can have *emphasis in italics* or be **bold**, and you can define
```

```
code samples with backquotes, like when you talk about a command:
sudo "gives you superuser powers!
```

As you can see, that syntax looks very readable in plain text. When the time comes to create a specific format (like HTML), the title converts to a significant heading, bigger fonts than the subtitle (as it should), and the lists automatically get numbered. Already you have something quite powerful. Adding more items or changing the order in the numbered list doesn't affect the numbering, and titles can change importance by replacing the underline used.

As always, throughout this book, create a virtual environment, activate it, and install the dependencies. Sphinx, in this case:

```
$  pip install Sphinx
Collecting Sphinx
  Downloading https://files.pythonhosted.org/Sphinx-3.0.3-py3-none-any.whl (2.8MB)
  2.8MB 703kB/s
...
Installing collected packages...
Successfully installed sphinx-3.0.3
```

The framework uses a directory structure to have some separation between the source (the plain-text files) and the build (which refers to the output generated). For example, if making a PDF from a documentation source, the data would be placed in the build directory. This behavior can be changed, but for consistency, I use the default format.

To get started, use the `sphinx-quickstart` tool (which should be available after installing Sphinx) to start a new documentation project. The process prompts you with a few questions. Accept all the default values by pressing Enter.

```
$ sphinx-quickstart
Welcome to the Sphinx 3.0.3 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value if given in brackets).
...
```

I chose "My Project" as the project name that gets referenced in several places. Feel free to choose a different name.

After running the sphinx-quickstart command, there should be files in the working directory resembling these:

```
.
|__ Makefile
|__ _build
|__ _static
|__ _templates
|__ conf.py
|__ index.rst
|__ make.bat
```

These are some of the important files that you interact with:

- `Makefile`: Developers who have compiled code should be familiar with this file. If not, think of it as a file containing instructions to build documentation output when using the make command. This file is the one you interact with the most to generate output.
- `_build`: This is the directory where generated files go after a specific output is triggered.
- `_static`: Any files that are not part of the source code (like images) go here, and later linked together in the build directory.
- `conf.py`: This is a Python file holding configuration values for Sphinx, including those preselected initially with the `sphinx-quickstart` command.
- `index.rst`: The root of the documentation project. This file connects to others if the documentation splits into other files.

Be cautious of jumping right into writing documentation. A lack of knowledge about layout and outputs can be confusing and could significantly slow your entire process.

Take a look inside “index.rst. “ There is a significant amount of information and some additional complex syntax.
Right after the main title in the `index.rst` file, there is a content listing with a `toctree` declaration. The `toctree` is the central element to gather all documents into the documentation. If other files are present, but not listed under this directive, those files would not get generated with the documentation at build time. We are now ready to generate output.

Run the `make` command and specify HTML as output. This output can be used directly as a website as it has everything generated, including JavaScript and CSS files.

```
$ make html
sphinx-build -b html -d _build/doctrees   . _build/html
Making output directory...
Running Sphinx v3.0.3
making output directory... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 1 source files that are out of date
updating environment: [new config] 1 added, 0 changed, 0 removed
reading sources... [100%] index
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices...  genindexdone
writing additional pages...  searchdone
copying static files... ... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded.

The HTML pages are in _build/html.
```

With our first pass at generating HTML from the two files, we have a fully functional (static) website.

Inside the `_buildi` directory, you should now have two new directories: `doctrees` and HTML. We are interested in the HTML directory holding all the files needed for the documentation site.

With so little information, Sphinx was able to create a lot. We have a basic layout with some information about the project's documentation, a search section, a table of contents, copyright notices with name and date, and pagination. The search part is interesting because Sphinx has indexed all the files, and with some JavaScript magic, it has created a static site that is searchable. When you are ready to make more modifications, run the `make html` command again to regenerate the files.

If the look and feel of the generated output are not to your liking, Sphinx includes many themes that can be applied to change how the HTML files render the documentation ultimately. Some critical open source projects, such as SqlAlchemy and Ceph, heavily modify the HTML looks by changing the CSS and extending the templates.

Sphinx changed the way I thought about writing documentation. I was excited to easily document almost all of my open-source projects and a few internal ones.

Chapter 10: Command-line Rosetta Stone

This chapter serves as a guidebook for users coming from another language. The source code can be found here[84]. The same style, hello world command-line tool is written in many languages.

R Hello World

This step is an example of a hello world command-line tool in R. The source code is here[85].

```
#!/usr/bin/env Rscript
#Hello World R command-line tool
#


suppressPackageStartupMessages(library("optparse"))
parser <- OptionParser()
parser <- add_option(parser, c("-c", "--count"), type = "integer",
                     help = "Number of times to print phrase",
                     metavar = "number")
parser <- add_option(parser, c("-p", "--phrase"),
                    help = "Phrase to print")

args <- parse_args(parser)


# Function to Generate Phrases
phrasegen <- function(arguments){
    for (count in 1:arguments$count) {
        cat(paste(arguments$phrase, "\n"))
    }
}

#Run the program
phrasegen(args)
```

Depends on https://github.com/trevorld/optparse for R

[84] https://github.com/noahgift/cli-rosetta
[85] https://github.com/noahgift/cli-rosetta/tree/master/R/hello-world

Usage

```
$  hello-world git:(master) $ ./hello-world.R --count 5 --phrase "hello world"
hello world
hello world
hello world
hello world
hello world
```

Bash Hello World

This step is a hello world Bash example. The source code is here[86].

```
#!/bin/bash
#output looks like this:
#
#  $hello-world git:(master) $ ./hello-world.sh --count 5 --phrase "hello world"
#hello world
#hello world
#hello world
#hello world
#hello world

#Generate phrase "N" times
phrase_generator() {
    for ((i=0; i<$1;i++)); do
        echo "$2"
    done
}

#Parse Options
while [[ $# -gt 1 ]]
do
key="$1"

case $key in
    -c|--count)
    COUNT="$2"
    shift
    ;;
```

[86] https://github.com/noahgift/cli-rosetta/tree/master/bash/hello-world

```
    -p|--phrase)
    PHRASE="$2"
    shift
    ;;
esac
shift
done

#Run program
phrase_generator "${COUNT}" "${PHRASE}"
```

To lint use `make lint`. And now run it:

```
$  hello-world git:(master) $ ./hello-world.rb --count 5 --phrase "hello world"
hello world
hello world
hello world
hello world
hello world
```

Environment

- Installed https://github.com/koalaman/shellcheck in VSCode
- On Homebrew you can install: `brew install shellcheck`

You may need to also do:

```
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.zshenv
$ echo 'eval "$(rbenv init -)"' >> ~/.zshenv
$ echo 'source $HOME/.zshenv' >> ~/.zshrc
$ exec $SHELL
```

Go Hello World

The `go` source code is here[87].

[87] https://github.com/noahgift/cli-rosetta/tree/master/go/hello-world

```
package main

import (
    "fmt"
    "gopkg.in/urfave/cli.v1" // imports as package "cli"
    "os"
)

func main() {
    app := cli.NewApp()
    app.Flags = []cli.Flag{
        cli.StringFlag{
            Name:  "phrase",
            Usage: "Print phrase",
        },
        cli.Int64Flag{
            Name:  "count",
            Usage: "Count to print a phrase",
        },
    }

    app.Action = func(c *cli.Context) error {
        sum := 0
        for i := 0; i < c.Int("count"); i++ {
            sum -= i
            fmt.Println(c.String("phrase"))
        }
        return nil
    }

    app.Run(os.Args)
}
```

Running program

Run `make all`

Then run program:

```
$  hello-world git:(master) $ hello-world --phrase "hello world" --count 5
hello world
hello world
hello world
hello world
hello world
```

Environment

- Setup on mac with homebrew:
  - https://stackoverflow.com/questions/12843063/install-go-with-brew-and-running-the-gotour[88]

Setting these variables:

```
export GOPATH="${HOME}/.go"
export GOROOT="$(brew --prefix golang)/libexec"
export PATH="$PATH:${GOPATH}/bin:${GOROOT}/bin"
export GOBIN=$GOPATH/bin
```

- Write first program with Go
  - https://golang.org/doc/code.html[89]
- Using cli:
  - https://github.com/urfave/cli[90]
- Running lint:
  `make lint`
- Installing dependencies:
  `make install`

Node Hello World

The source code for the `node` examples is here[91]. This project has several components. First, there are the `.js` file.

[88] https://stackoverflow.com/questions/12843063/install-go-with-brew-and-running-the-gotour
[89] https://golang.org/doc/code.html
[90] https://github.com/urfave/cli
[91] https://github.com/noahgift/cli-rosetta/tree/master/node/hello-world

```
#!/usr/bin/env node
"use strict";

/*

Hello World Commandline Tool

node index.js --phrase "hello world" --count 10

hello world hello world hello world

*/

const program = require('commander');
program
  .version('0.0.1')
  .option('-p, --phrase [value]', 'phrase')
  .option('-c, --count <n>', 'Number of Times To Repeat Phrase', parseInt)
  .parse(process.argv);

/**
 * Multiplies string with additional space.
 * @param {string} phrase The phrase.
 * @param {number} count The number of times to repeat
 * @returns {string} The multiplied string
 */
function phraseGenerator (phrase, count) {

    return phrase.concat(" ").repeat(count);

}

// Check to see both options are used
if (typeof program.phrase === 'undefined' ||
    typeof program.count === 'undefined') {

    console.error('ERROR! --phrase and --count options required');
    program.help();
    process.exit(1);

}

// Print Phrase To Standard Out
```

```
console.log(phraseGenerator(
                program.phrase,
                program.count));
```

Next there is a `package.json` file.

```
{
  "name": "nodecli",
  "version": "1.0.0",
  "description": "nodecli",
  "main": "index.js",
  "dependencies": {
    "commander": "^2.10.0"
  },
  "devDependencies": {
    "eslint": "^4.1.1",
    "eslint-config-defaults": "^9.0.0"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/noahgift/nodecli.git"
  },
  "keywords": [
    "cli"
  ],
  "author": "Noah Gift",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/noahgift/nodecli/issues"
  },
  "homepage": "https://github.com/noahgift/nodecli#readme"
}
```

To run the example, you would do the following.

Steps to run:

```
npm install
./hello-world --phrase "hello world" --count 3
```

The output should be:

```
hello world hello world hello world
```

Multi-paradigm Node.js

Getting Started

This application source code is here[92].

To build this project, you need to:

```
npm install
```

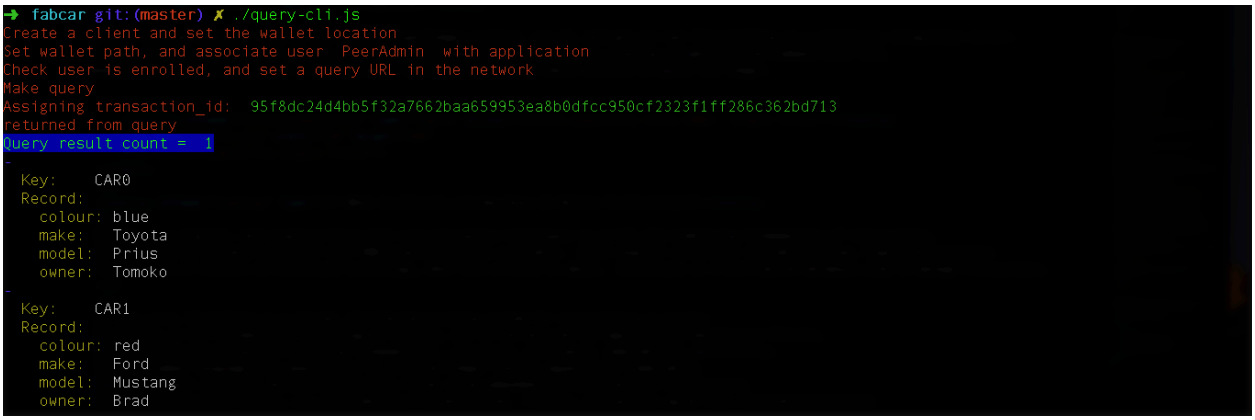
To talk to the blockchain network, please run ./startFabric.sh. Then ./query-cli.js. Further information and background of this forked version can be found here[93]

Features

- Color Output
- JSON Formatting
- Async Network Operations
- Blockchain Integration

Screenshots

* Screenshot No Options:



Output

[92] https://github.com/noahgift/cli-rosetta/tree/master/node/multi-paradigm
[93] http://hyperledger-fabric.readthedocs.io/en/latest/write_first_app.html

* Screenshot One Option:

```
➜ fabcar git:(master) ✗ ./query-cli.js --car "CAR2"
Create a client and set the wallet location
Set wallet path, and associate user  PeerAdmin  with application
Check user is enrolled, and set a query URL in the network
Make query
Assigning transaction_id:  ab679885c5a4fa8a3148a63c253949102e13a3762fe592b4e94a2dc8d1c49e1f
returned from query
Query result count =  1
colour: green
make:   Hyundai
model:  Tucson
owner:  Jin Soo
```

Output

The source is below. You can see how a more sophisticated `node` tool integrates with both blockchain, and colored output works.

```
#!/usr/bin/env node
"use strict";

/*

Hyperledger Query Commandline Tool

./query-cli.js
__dirname = path.resolve();
*/
const Hfc = require('fabric-client'),
    path = require('path'),
    chalk = require('chalk'),
    prettyjson = require('prettyjson'),
    program = require('commander'),
    options = {
        walletPath: path.join(__dirname, './network/creds'),
        userId: 'PeerAdmin',
        channelId: 'mychannel',
        chaincodeId: 'fabcar',
        networkUrl: 'grpc://localhost:7051'
    };
let channel = {},
    transactionId = null,
    client = null,
    jsonResult = null;
program
    .version('0.0.1')
```

```
    .option('-c, --car [value]', 'car to query')
    .parse(process.argv);

/**
 * Queries Blockchain
 * @param {string} chaincodeFunc The chaincode function to query
 * @param {string} car The individual car to query
 * @returns {string} nothing
 */
function queryHelper (chaincodeFunc = 'queryAllCars', car = '') {

    Promise.resolve().then(() => {

        console.log(chalk.red("Create a client and set the wallet location"));
        client = new Hfc();

        return Hfc.newDefaultKeyValueStore({path: options.walletPath});

    })

    .then((wallet) => {

        console.log(chalk.red("Set wallet path, and associate user ",
            options.userId, " with application"));
        client.setStateStore(wallet);

        return client.getUserContext(options.userId, true);

    })

    .then((user) => {

        console.log(
        chalk.red(
            "Check user is enrolled, and set a query URL in the network"
            ));
        if (typeof user === "undefined" || user.isEnrolled() === false) {

            console.error("User not defined, or not enrolled - error");

        }
        channel = client.newChannel(options.channelId);
        channel.addPeer(client.newPeer(options.networkUrl));
```

```
})
.then(() => {

    console.log(chalk.red("Make query"));
    transactionId = client.newTransactionID();
    console.log(chalk.red("Assigning transaction_id: "),
        chalk.green(transactionId._transaction_id));

    // The queryCar - requires 1 argument, ex: args: ['CAR4'],
    // The queryAllCars - requires no arguments , ex: args: [''],
    const request = {
        chaincodeId: options.chaincodeId,
        txId: transactionId,
        fcn: chaincodeFunc,
        args: [car]
    };

    return channel.queryByChaincode(request);

})
.then((queryResponses) => {

    console.log(chalk.red("returned from query"));
    if (typeof queryResponses.length === 'undefined') {

        console.log("No payloads were returned from query");

    } else {

        console.log(
            chalk.bgBlue("Query result count = ", queryResponses.length));

    }
    if (queryResponses[0] instanceof Error) {

        console.error("error from query = ", queryResponses[0]);

    }
    jsonResult = JSON.parse(queryResponses[0].toString());
    console.log(prettyjson.render(jsonResult, {
        keysColor: 'yellow',
        dashColor: 'blue',
```

```
            stringColor: 'white'
        }));

    })
    .catch((err) => {

        console.error("Caught Error", err);

    });

}
// Run The Command line Tool
if (typeof program.car === 'undefined') {

    queryHelper('queryAllCars');

} else {

    queryHelper('queryCar', program.car);

}
```

Python Hello World

Recommended environment

Python 3.6.1

Running

Steps to Run:

Install packages:

```
make install
```

Activate Virtual Env:

```
source ~/.hello-world-py-cli/bin/activate
```

Run Tool:

```
./hello-world.py --phrase "hello world" --count 3
```

The output should be:

```
hello world hello world hello world
```

The ‘Makefile looks like:

```
install:
      mkdir -p ~/.hello-world-py-cli &&\
      python3 -m venv ~/.hello-world-py-cli &&\
      pip install -r requirements.txt


source-cmd:
    echo "Virtualenv source command"
    #source ~/.hello-world-py-cli/bin/activate

lint:
    pylint hello-world.py
```

The requirements.txt file looks like:

```
click
pylint
```

Finally, the `python` code is as follows.

```
#!/usr/bin/env python
import click

@click.version_option("0.1")
@click.group()
def cli():
    """Hello World"""

@cli.command("hello")
@click.option("--phrase", help="phrase to print")
@click.option("--count", help="Number of times to repeat phrase", type=int)
def hello(phrase, count):
    """Hello World Command-line tool"""

    while count:
        count -= 1
        click.echo(phrase)
```

```
if __name__ == '__main__':
    cli()
```

A full example of lint and run:

```
    (.hello-world-py-cli) $  hello-world git:(master) $ ./hello-world.py hello\
        --phrase "hello world" --count 3
    hello world
    hello world
    hello world
    (.hello-world-py-cli) $  hello-world git:(master) $ make lint
    pylint hello-world.py

    --------------------------------------------------------------------
    Your code has been rated at 10.00/10 (previous run: 7.50/10, +2.50)
```

You can find the latest up to date examples in the Github Repo https://github.com/noahgift/cli-rosetta[94].

[94] https://github.com/noahgift/cli-rosetta