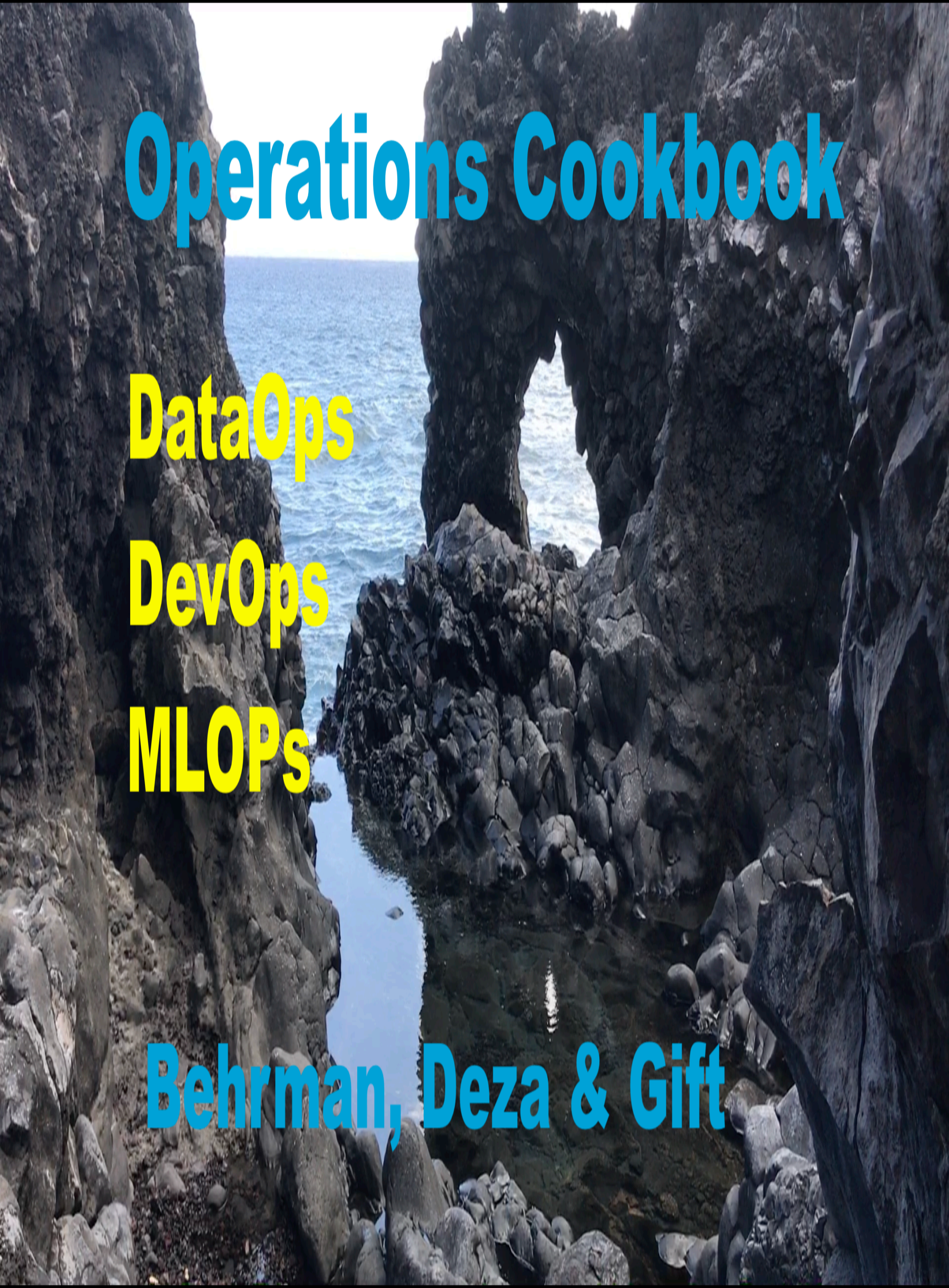
Operations Cookbook

DataOps

DevOps

MLOPs

Behrman, Deza & Gift



Operations Cookbook: DevOps, DataOps and MLOps

Learn to operationalize data, machine learning and software

Noah Gift and Kennedy Behrman

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

This version was published on 2021-12-14

Contents

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/operationscookbookdevopsdataopsandmlops.

About the cover

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/operationscookbookdevopsdataopsandmlops.

Chapter 1: Python, Pandas, Git and Local Development Setup for Data Engineering

Kennedy Behrman

Code Organization.

A Python program consists of a series of statements and states. The state is the current environment of the running software. This includes what variables and functins have been defined, and any values that have been set. In a general sense the job of the statements is to define and change the state. Python statements can be run either in an interactive session or from a file. Any Python installation comes equiped with a built in Python Interpreter. By typing `python` with no arguments in a shell, the interpreter will open an interactive session. If you type a satement in an interactive session, it will run once you press enter. Any output will be displayed below the line. This is a great way to start playing around with the language as the feedback is immediate. Experienced programmers often open an interactive session to try out ideas and test syntax.

Managing a Python Environment

Libraries
A module is a series of Python statements saved in a file and meant for resuse. A package is a group of files bundled together to appear as a module. A module or package, can be imported into an interactive Python session, into a Python script, or even into another module. Once it is imported, any functionality defined in the module is available for use. When Python is installed, it comes bundled with collection of modules known as the Python Standard Library[1]. These modules are available for use in any Python program. A great deal of the success and power of Python comes in the use of third-party modules and packages.

What is a python environment? libraries, code available to use
Importance of segregating project environments

- packages installed globbally by default
- different python version
- different library versions
- sharing project with other people
- deploying project

direnv
pyenv

- system python vs project python(s)
- `pyenv virtualenv`
- local

pipenv

[1]https://docs.python.org/3/library/

Evaluating to True or False

Python has a number of operations that evaluate to the special values `True` or `False`. These include comparision operations, boolean operations, and object evaluations.

Comparisons yield boolean True or False
can be chained

Comparision Operations

Comparison can be made either by value or identity. Comparing by value are more generalized than camparison by value.
The comparison operators that compare by value are:

- == Equals
- `!=` Not Equals
- < Less Than
- <= Less Than or Equal
- > Greater Than
- >= Greater Than or Equal

In most cases, two objects that are of different type will always evaluate as not equal. So the comparison `1 == 'b'` will evaluate as `False`, and `1 != 'b'` will evaluate as `True`. One notable exception to this is numeric types, such as integars and floating point numbers. The comparison `1 == 1.0` will evaluate to `True`, and `1 != 1.0` will evaluate to `False`.
The order comparisons, those that test greater or less values, will generally raise an error if the objects compared are of different types. A notable exception, again, is numberic types. The comparision `3.0 >= 2` will result in `True`. The order of objects depends on the type of objects being consisdered. For example text (type string), uses lexographic order and numeric types use numeric order.

Order comparison can be chained with multiple operators `1 < 2 <= 3` will result in `True`.

The comparison operators which compare by identity are `is` and `is not`. They are most commonly use to compare against the special object `None`.

Membership Operations

Some objects in Python can contain others. For example the word (of type string) `"Henry"` contains the letter `"r"` (also a string). The `in` operator tests for this type of membership. The expression `"r" in "Henry"` will return `True`, and `"b" in "Henry"` will return `False`.

Boolean Operations

Boolean operatins are based on boolean math, which you may have learned in a mathematics or philosophy course. The operators are `and`, `or`, and `not`. The first two take two operands, the last one. The `and` operator returns true if both of it's operands evaluate as `True` and `False` if either evaluates to `False`. The `or` operator evaluates to `True` if either of its operands evaluates as `True` and `False` if they are both `False`. The `not` operator returns `False` if it's operand evaluates to `True` and `True` otherwise.
You can make more complex logical operations by nesting boolean operations in parenthesis. The expression `(False and False) or (not False)` evaluates to `True`, as `not False` is `True`.

Object Evaluations

All objects (everything) in Python evaluates as `True` or `False`. This means you can use them in the places where you would test for `True` or `False`, such as in Boolean operations. Generally, most Python objects evaluate as `True`. The exceptions are:

1. Numeric types that equal zero, such as `0`, or `0.0`.
2. The constants `False` and `None`.
3. Anything that has a length of zero. This includes the empty string, `""`.

Chapter 2: Linux and Bash for Data Engineering

Noah Gift

Notes on Bash

Project Goal: Truncate file with Bash

For this project you will create a shell pipeline that truncates a file via random shuffling, then verifies the correct number of lines. Many times large files are so big that traditional data science libraries like `pandas` or `jupyter` cannot process them. One approach to dealing with this problem is to sample the file by truncating and shuffling the results.

Part 1: Count the lines in the file and inspect the contents

1. Run `wc -l nba_2017.csv`
2. How many lines are in the file?
3. Run the `head nba_2017.csv` and inspect the first few rows of the file.

Part 2: Truncate and shuffle the file

1. Truncate and shuffle the file `shuf -n 100 nba_2017.csv > small_-nba_2017.csv`
2. Count the number of lines. How many are there?
3. If you inspect the first few lines what do you see? ‘head nba_-2017.csv

Part 3: Remove Column Names Before Shuffle

1. What happens when you run `tail -n +2 nba_2017.csv | head`?
2. How could use this approach to remove the column heads before shuffling?
3. Why would want to do this and how could you append them back on after you shuffle?

Project Goal: Build Bash Command Line Tool that accepts two arguments

You have learned to build Bash command-line tools. This exercise helps you master this concept.

Let’s create a Bash Command Line Tool that accepts two arguments. The current example accepts two arguments `--count` and `--phrase`.

To run the command you get the following output:

```
./cli.sh --count 5 --phrase "hello world"
hello world
hello world
hello world
hello world
hello world
```

Part 1: Perform the following steps

1. Change the `--count` option to be named `--number`.
2. Change the the `--phrase` option to be name `--message`.
3. Rerun the command-line tool

Part 2: Change the phrase generate

1. The `phrase_generator()` bash function prints out the phrase you entered as the second argument via `echo "$2"`. Change function so it will print out the phrase with the following appended to the front: `You entered phrase:`.

Project Goal: Search the filesystem with the find command

Now that you have learned to search with the `find` command let's put this to use.

Part 1: Search with find

1. Search the /bin directory and count the number of file you find `find /usr/bin | wc -l`
2. Find the location of python versions on your system: `find /usr/bin | grep python`

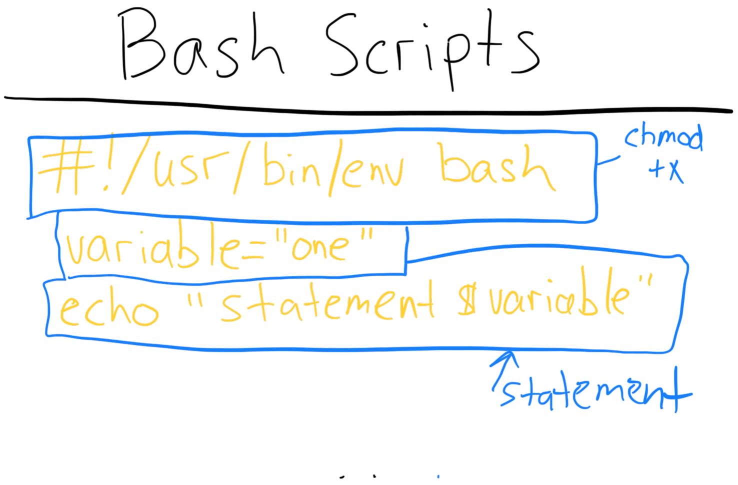
3. How could you change the last command to find a specific version of Python?

Part 2: Search for file types

1. Search for all of the csv files in `docs` directory: `find -name "*.csv"`
2. How many did you find?
3. Can you search for files with the `.txt` extension? How many do you find?

bash-scripts

Examples of Bash Scripts



Bash Scripts

Basic Bash Script

Core components:

- shebang line
- debug modes
- statements and variables

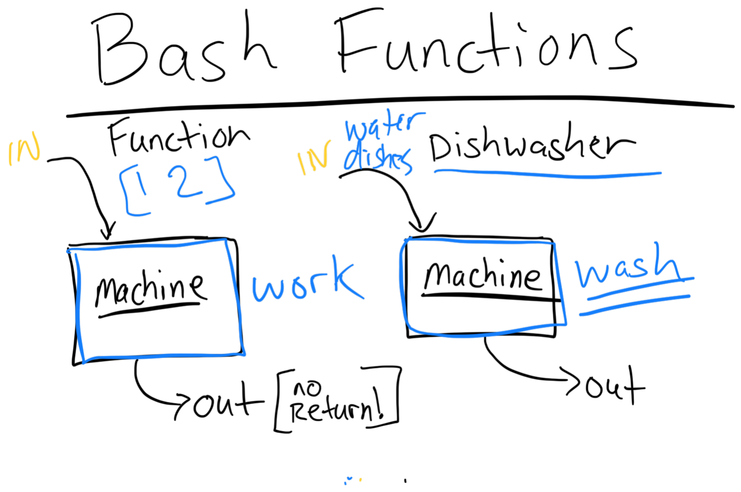
Core components breakdown

- bash-script-basics.sh[2]

```
#!/usr/bin/env bash
#
# This is where comments go
# It can be useful to explain the purpose of your code
## Note you can also start your script with #!/usr/bin/ba\
sh -xv for verbose debugging
## https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_02\
_03.html

# Set strict mode.  Causes shell to exit when a command f\
ails
set -e

# Enables printing of shell input lines as they are read
#set -v

# Enables print of command traces before executing command
#set -x

# Set a variable
variable="one"

echo "This is a script with a variable: $variable"
```

[2] https://github.com/noahgift/bash-scripts/blob/main/bash-script-basics.sh

Bash Functions



Bash Functions

- Bash Functions[3]

```
#!/usr/bin/env bash
#
## Basic structure
#function_name () {
#    command
#}

# Parameters
mimic() {
    echo "First Parameter: $1"
    echo "Second Parameter: $2"
}
```

[3]https://github.com/noahgift/bash-scripts/blob/main/bash-functions.sh

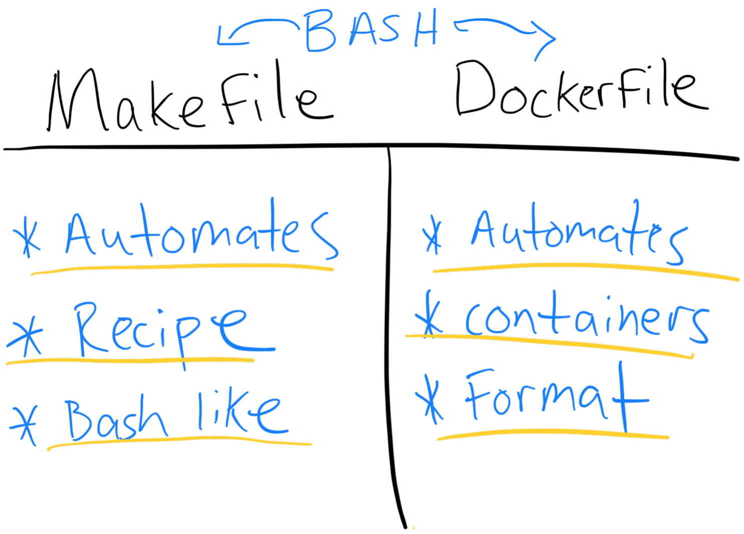
```bash

# Call the function with two parameters
mimic 1 2

# Call it again with two different parameters
mimic 99 100

# Add function
# No return value, so must echo
add() {
    num1=$1
    num2=$2
    result=$((num1 + num2))
    echo $result
}

# will echo three
add 1 2

# capture output of function
# will not echo 14 because I captured it
output=$(add 5 9)

# sent that value into add again
add $output $output

#echo $output
```

Bash CLI

```bash
# Run Script:
        ./cli.sh --count 5 --phrase "hello world"
```

```
* [Bash CLI](cli.sh)

#!/usr/bin/env bash
#output looks like this:
#
# Run Script:
#        ./cli.sh --count 5 --phrase "hello world"
#hello world
#hello world
#hello world
#hello world
#hello world

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

## B. Parses input from the CLI
#Parse Options
while [[ $# -gt 1 ]]
do
key="$1"

case $key in
    -c|--count)
    COUNT="$2"
    shift
    ;;
    -p|--phrase)
    PHRASE="$2"
    shift
    ;;
esac
shift
done

##C.  Pass parsed input to function and run everything
#Run program
phrase_generator "${COUNT}" "${PHRASE}"
```

Bash Makefiles and Dockerfiles



L BASH

Makefiles

Basic formula:

```
target :
    recipe
```

- Good Example is the Linux `Makefile`[4]
- More documentation[5]

Dockerfiles

Builds a container.
You can read more documentation here[6].

To build it: `docker build .`

Find container id: `docker image ls`

```
docker run -it <your-id> bash
````

Then try out `ipython` and `import pandas`

Bash Data Structures: Lists and Hashes

Bash 4.0 and beyond have "hashes", you should check if it\
 is available:
``` bash --version```

Note that modern cloudshell environments have version gre\
ater than 4.0, like Github codespaces and AWS Cloudshell.\
  On OS X, you will need to ```brew install bash``` to up\
grade.

Bash Lists

* [lists](https://github.com/noahgift/bash-scripts/blob/m\
ain/lists.sh)

```bash
```

[4]https://github.com/torvalds/linux/blob/master/Makefile
[5]https://www.gnu.org/software/make/manual/make.html#Overview
[6]https://docs.docker.com/engine/reference/builder/

```
#!/usr/bin/env bash

#This is a bash list/arrary
declare -a array=("apple" "pear" "cherry")

## now loop through the above array
for i in "${array[@]}"
do
   echo "This ${i} is delicious!"
done
```

Bash Hashes

- hashes[7]

```
#!/usr/bin/env bash
# Requires Bash >=4.0

declare -A mealhash=([dinner]="steak" [lunch]="salad" [br\
eakfast]="fruit" )

## now loop through the above hash
for key in "${!mealhash[@]}"; do
   echo "For $key I like to eat: ${mealhash[$key]}"
done
```

[7]https://github.com/noahgift/bash-scripts/blob/main/hashes.sh

Chapter 3: Scripting with Python and SQL for Data Engineering

Chapter 4: Command-Line Tools in Python for Data Engineering