Module 2

Module 2: Git Fundamentals, Setting up github repository and AWS login

Week 1 Learning Objectives

At the conclusion of this module, you should be able to:

* Create an AWS EC2 instance, log in and run the ubuntu command line
* Use Linux commands to NAVIGATE in the directory space
* Use Linux commands to inspect files: includes cat, grep, find
* Use Linux editors to modify files
* Use git/github and wget as tools to download content to your AWS instance

Reading

* Reading Assignment:
  + [The Linux Command LineLinks to an external site.](https://learning.oreilly.com/library/view/the-linux-command/9781492071235/), Chapters 1-5 **Assigned at a SKIM (looking through the chapters in a 'would I buy this book' level first, then as a resource) LEVEL**
  + [Mastering Python For BioinformaticsLinks to an external site.](https://learning.oreilly.com/library/view/mastering-python-for/9781098100872/app01.html#idm45666223879840) Appendix A (**REQUIRED reading, and note that is  ONLY Appendix A**)

Lab 1 Makefile, linux, bash

**DS 5111 Data Engineering Fall 2023**

In this lab, you will gather materials from project gutenberg as simple text.  You will set up an automate way to download content, do some very basic operations to count words etc, and put all that in a repository so that someone else can retrace your steps with just one command.  They should also be able to expand, or pause the automation at certain steps to rerun parts of it.  The downloaded data should NOT save to the repo.  When someone clones your repo and runs it, the data should be downloaded from scratch (to save pushing around all the data)

**Lab 1 Instructions**

Please read THROUGH ALL the instructions before you start, because only some of the files we use will be commited in your repository

**Before we get going, let's update the package manager on your linux box.**

* sudo apt update # This is usual and necessary on new instances, but you don't have to do it every time you start, only if you terminate and start a whole new box

**Creating a repository ( 1 point)**

* In your github account, create a repository with the name <UVA ID>\_DS5111su24\_lab\_01, this is important since we'll automate some of the process.
* Make sure the repository is public
* Add EfrainOlivaresUVA and MiaHanzhangYuan as collaborators. [https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-access-to-your-personal-repositories/inviting-collaborators-to-a-personal-repositoryLinks to an external site.](https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-access-to-your-personal-repositories/inviting-collaborators-to-a-personal-repository)
* On your AWS linux box, clone your repository so you can save your work

**Create a branch for your work.  (3 points)**

Using the command line in the repo you just cloned you will create a branch for your work and push it up to github.  This is how we will communicate on your work.

* Create a new branch named project\_gutenberg\_texts and push that to github.
* The rest of your lab work should get checked into and pushed to this branch

**Sidebar: Project Gutenberg Project**

This web page [https://www.gutenberg.orgLinks to an external site.](https://www.gutenberg.org/ebooks/1065) provides the text of many books that no longer have copyrights.  They include a LOT of the classics.  There are several formats you can download the text in.  They are variations of text formats.  We’ll be using the most plain of them all, just a simple txt.

Just like we did in class, you will gather the id’s of 10 books by Edgar Allan Poe.

**Makefile setup (2 pts)**

* Create a  makefile and setup a default: task as shown in the lecture demo, **so it just cats the makefile itself**
* Create a job called `get\_texts'` which should download (wget) the content.

The easiest way to do this once you have the id of a book, is to use the wget command as shown in class `wget [https://www.gutenberg.org/cache/epub/17192/pg17192.txtLinks to an external site.](https://www.gutenberg.org/cache/epub/17192/pg17192.txt)`

Running the `make get\_texts` should download the set of files you’ve selected.

**(2 extra points)**

You get two points for this part if you wrote a line per book.  However, there is another 2 points if you create a file and loop through the book id’s to make it easier to expand.  The makefile job should then run the script, as in `bash get\_the\_books.sh`

.**gitignore to exclude `.txt.` files (2 pts)**

Since we do NOT want the data in the repository, at this point create a .gitignore, or update the existing one to insure that \*.txt files to not get seen or committed by git.   Check in your .gitignore so that it becomes part of your project

**Processing Your Data:**

url to The Raven txt: [https://www.gutenberg.org/cache/epub/17192/pg17192.txtLinks to an external site.](https://www.gutenberg.org/cache/epub/17192/pg17192.txt)

Note that all these should be jobs in your makefile and output to console

* 1 pt: raven\_line\_count  Write a script that uses `wc` to count the number of lines in The Raven
* 1 pt: raven\_word\_count: How many words are there in the raven, use wc again.
* 1 pt: raven\_counts: How many lines in The Raven have the word ‘raven’ in it.  This one should output counts for raven (lowercase), Raven (title case), and a count where case is ignored.
* 2 pt:  total\_lines: how many lines are in all the files you downloaded?
* 2 pt: total\_words: how many words are in all the files you downloaded?

**Creating a Pull Request PR (1 pt)**

When you have some work started, create a Pull Request to merge your code into main.  Request a review from myself and Mia.  At the end of the lab we’ll merge your work into the main branch.

**Round Trip**

By round trip here I mean how your repo performs when some one else checks it out, and runs make.  Did all the jobs run right out of the box?

Please note all the names of the makefile jobs and the names of the repo.  I will automate a git pull and running of the jobs.  And remember your repo should not have the data or the answers in it.  I should be able to git-pull, cd into the repo, and run the makefile jobs one after the other to see the results with no extra work.

This last part is worth **5pts**

More Info from the Module

Some Points for the Makefile lab

I wanted to highlight some of the commands that I think are most useful.   Think of this as a starter 'cheat sheet' from which you can extend as per your use case.

As mentioned in class, and the analogy with a game, some commands represent the basics just to 'be' in the command line.

Seeing/Interacting with environment

* ls is usually one of the first commands I type, it just gets me oriented as to what is available. If it's all directories then all I can do is move, if it's all files then all I can do is act or trigger an action.
* By default ls lists the names. ls -la is wordier and quite useful. l is for list format and includes details about the file like size and permissions. a is for all, and the listing will include even hidden files. Hidden files start with a . by default.
* When you execute ls -la, as note the very first letter in the descriptors. For example dxxwxx..... In this case the d indicates it is a directory. In our game mindset, it's a 'd'oor.

Navigation

* cd is the command to move. cd <directory> takes you into a directory.
* Two important shortcuts, cd .. takes up to the parent directory. The file structure is a tree so it can only be interpreted one way. We'll talk about . also, that one means 'here'
* A good shortcut to know is cd by itself takes you to /users/<username> directory

Interacting

* We already mentioned directories, you can go into and out of them.
* For regular files, whose descriptor does NOT start with a d, look at the right most 3 letters. The triplet will consist of r, w, x in that order. Read, Write, eXecute. If the letter appears you can act on it. So for example, if it's a text file and you see r--, you can cat the contents, or see the contents with nano. A w means you can edit the contents too, otherwise, nano or vim won't let you save the file. Finally, an x means you can execute it. If it has a 'shebang' as we saw in class, it means you can just type ./<filename> and a script will execute. Otherwise, you need to know what language it is written in. So bash <file>.sh runs 'sh'ell scripts. Python <file>.py runs python scripts. etc.

Other interactions that are a must know…

* mv , meaning move a file from one location (name), to a new location (new name)
* cp , copy a file
* rm , deletes a file
* Note that by default rm won't work on directories. Linux won't let you remove a directory because it might have files under it, well, it won't let you do it by default. So for directories you need to force the action with rm -rf <directory>.  (r for recursive, f for force)

Data processing/Searching/Modifying

* Remember that you chain commands with the pipe |
* grep, searches for matches to a string. Use the -E and quotes to use regular expressions. For example grep book matches lines containing book. grep -E "(book|magazine)" will match lines containing EITHER book or magazine
* sed, replaces strings on the fly. So if you cat a file with a bunch of lines like 'abcdef' you can use cat <file> | sed 's/d/D/g' and that will turn each line into 'abcDef'. The structure of the command is parsed by the '/'. So 's/d/D/g'… the first thing is 's', for search. The next slot between '/' is 'd', the thing we're searching for. The third slot between '/' is 'D', what we replace when we find a string. In this case the last slot, the 'g' stands for global, meaning 'as many as you find in a line'. You will see this last slot make more sense in more complicated cases where you may only want to replace the first match for example.

About makefiles

Remember the structure….

make-command:

<tab>shell-commands

<empty line>

That <tab> has to be a tab, spaces will break it.

Chaining…

* Once you create a make-command, you can chain them by simply defining make-command2: make-command2 etc.
* When I say make-command, it can be a descriptive command like 'run\_this', BUT given the history of the makefile it is usually a filename. So myfile.csv: implies that the shell commands belonging to this make-command will generate that file.
* This last point is particularly well suited for data pipelining. For example, if you have a make command like download\_raw\_text.log: it would execute if the log does not exist. Otherwise it runs the lines to, presumably download raw text, then create a log. Again, the existance of lack of the file dictates if the command runs or not.
* Say now you have the downloaded text and you need to clean the text, the next step might be clean\_text.log:. Or in some cases, it may be one unified csv with the clean text inside it. (There are many ways this can go, so let's go for the content version). So if we go for the content version we would have clean\_content.csv:
* Now your next step is to run that through some analysis and, you guessed it, create a result.csv file. So the next job would be result.csv:

Putting it all together your makefile now looks like

download\_raw\_text.log:

<bash commands to download>

<bash command to create download\_raw\_text.log, maybe the count of files retrieved is in it>

clean\_content.csv: download\_raw\_text.log

<shell commands to clean text and preserve it all in one file>

result.csv: clean\_content.csv

<shell command to run a model and store results in a file>

So in this case, if you realize you have a bug in your model, you just fix the bug, delete 'result.csv' and run make result.csv. No major downloads or cleaning will be executed.

In the case you realize youre data cleaning scripts have a bug, you delete 'result.csv', AND 'clean\_content.csv' and run the command again. This time, the cleaning will happen.

In the case you realize the content you downloaded is out of date, you delete everything and run again.

And that reminds me of two indispensible commands that should be part of your arsenal…

wget which we saw in class, and curl. Wget is easier to use, but it, as the name implies only 'gets'. Curl is raw, and you can execute pretty much anything against an API, from GET, to PUT, DELETE, pass in data etc….

Link to Cloud Recording

Title: DS5111 Summer 2024 Data Engineering  
Date: May 28, 2024 8:24 PM Eastern Time (US and Canada)

Recording-1(198 MB)  
https://virginia.zoom.us/rec/play/BTENFov3VT3TWqEa\_dpCuzPv9WOTQs6K0H15UhgVeoylE7VZCz0Lgu0SJmyw-iI3SiJLXhbB\_MpZM\_Ht.ad\_k0HoP4IlBhdkr

Passcode: BQ9?2!&@

Learning Objectives

Week 3 Learning Objectives

By the end of this course you should have a basic understanding of

* how git commits are tracked
* what a git branch is and what it means to merge
* why storing data in git/github is generally a bad idea

You should also have been exposed to some ideas on making your code resilient and efficient by

* adding asserts to your code
* separating complex code into functions
* the use of logging to help avoid lengthy debugging
* using code organization, functions and variable names to make comments a 'last resort'

Reading Assignment

[Modern Python LibraryLinks to an external site.](https://learning.oreilly.com/library/view/modern-python-standard/9781788830829/)

* [Counting FrequenciesLinks to an external site.](https://learning.oreilly.com/library/view/modern-python-standard/9781788830829/00759777-ccb8-4558-890b-0f1d50599761.xhtml)
* [Clean Up TextLinks to an external site.](https://learning.oreilly.com/library/view/modern-python-standard/9781788830829/3f3c6c4d-9be0-4c2b-b1a4-f87ca9374f22.xhtml)
* [Logging To FileLinks to an external site.](https://learning.oreilly.com/library/view/modern-python-standard/9781788830829/cf238f15-dcc2-48ed-ac1d-81c36fe9e2c3.xhtml)

Week 3 Lab: python counter, cleaner, tokenizer

# DS 5111 Summer 2024 Lab, Part 2: Writing a tokenizer, word cleaner, word counter

This is part II of our lab, which will span several lectures. In this section of the lab, you will reproduce some of the functionality we saw using wc and grep but using homegrown python.

Overview of the lab:

* Write makefile to download data and quickly check it, runs some stats on it (Week 2, we just did that)
* Writing a tokenizer and word cleaner (this week)
* Lint and test the code
* Automate the testing in github actions and put a badge on our code to show it's working properly
* Package the code for reuse and use the tokenizer, cleaner, counter on totally new data as reusable code.
* Repackage our code in Docker, showing a *second* path to sharing and reusing our code

Details for this weeks lab:

CREATE A BRANCH FROM THE BRANCH OF WEEK 2.  I.e.  Main -> gutenberg\_branch -> this weeks branch.

Name it **tokenizer**

Create a python file and create three functions.

* clean\_text, should take a string, and should return all lowercase words, and throw out any punctuation
* tokenize, should take a string and return a python list, where each item is a word in the file
* count\_words, should take a string and return a dictionary with the words as keys, and their counts as value

In your makefile:

* Add a job to python3 -m venv env; pip install --upgrade pip; pip install -r requirements.txt
* Add a requirements.txt file which will install pytest and pylint

In this lab you will focus on juxtaposing writing CLEAR code against writing as LITTLE comments as possible. Use some of the skills highlighted in the reading to assist your code writing

Grading: Each of these is worth a point each except for the first which is 4 points

* create a new branch from your with the gutenberg text lab
* use assert to check inputs to functions
* use assert to check returns, (type?, None? ….)
* use descriptive variable names (no single letter names)
* use spacing to show coherent thoughts (this paragraph checks input, this one processes, this next one returns)
* use logging with descriptive messages to help you debug
* use comments to clarify anything that can't be made clear

# **Link to Cloud Recording June 4th, 2024**

https://canvas.its.virginia.edu/courses/109172/external\_tools/9

# **Slides used in lecture**

I've set up a repository to store our class materials.  To see the slides please follow this link.

If the link does not work for you, it means you have not accepted the invitation to the repository, please check your email.

[https://github.com/DS5111-Su24/DS5111\_Summer\_24\_content/blob/main/Week3SWSkillsI.mdLinks to an external site.](https://github.com/DS5111-Su24/DS5111_Summer_24_content/blob/main/Week3SWSkillsI.md)

Week 4

# **Learning Objectives**

# **Introduction to the Module**

The activity of testing is part of and inseparable from engineering. Our focus will be how to use testing to be efficient at writing code.

Mastering a few basic practices will save you a lot of time and headaches.

In this module, you will develop your own tests to accompany the code you wrote in the previous module.

## **Learning Objectives**

At the conclusion of this module, you should be able to:

1. Have a basic intuition about what to test, when to test, and how to test
2. Have a clear 'starter path' to set up test automation for new project
3. Rely on automation to test as you code

# **Reading**

# **Testing with pytest**

The reading assignment this week is intended to be processed like this.

1. Read through and get the ideas from chapters 1-6 of [Python Testing With Pytest, Second EditionLinks to an external site.](https://learning.oreilly.com/library/view/python-testing-with/9781680509427/)
2. Combine the ideas in the book with the material that will be presented in class.
3. You don't need to do the exercises in the book. Reserve execution of pytest using the examples in the book to apply them to YOUR code from the previous lab. I'll request some specific points in the lab, so it should be fairly easy to then go back and use the book as reference.
4. As you read, you want to pay particular attention to the features the tool provides. Testing is something you 'do', like walking. You can't really read about it and then just do it. Keep that in mind and then just 'engage' with your own code from the perspective of what is expected from it, and the features will make more sense after you get done with the lab.

Here's some particular points to stop and notice along the way.

* [Chapter 2 Structure Test FunctionsLinks to an external site.](https://learning.oreilly.com/library/view/python-testing-with/9781680509427/f_0026.xhtml#structure)- Please **NOTE** that you should strive to have all your tests have a Given/When/Then header. It will make it much easier to give you feedback during code review.
* [Chapter 2 Running a Subset of TestsLinks to an external site.](https://learning.oreilly.com/library/view/python-testing-with/9781680509427/f_0028.xhtml#subset)- We usually use all these methods, but pay particular attention to the **-k** flag.
* [Chapter 6 MarkersLinks to an external site.](https://learning.oreilly.com/library/view/python-testing-with/9781680509427/f_0064.xhtml)- Markers are a great feature of the pytest tool. At their core, they are python 'decorators', something we'll cover in class for a deeper understanding, but you don't need to know the internals to get familar with them in pytest.
* [Chapter 5 ParametrizationLinks to an external site.](https://learning.oreilly.com/library/view/python-testing-with/9781680509427/f_0057.xhtml#param_func)- Actually, you may skim the first few pages, what we're really interested in here is the example ch5/test\_func\_param.py. Take note of how the @pytest.mark.parametrize decorator is used. If you only look at the example and can pick up what to do from there, that is good enough.

## **Again, speed-read this through to get the concepts, then put it in action for your tests while you use the book for reference**

Week 4: Lab: Testing your functions

# **Week 4: Testing**

This week the reading and lab are focused on getting some hands-on test writing with some of Pytests most important features.

## **Testing LAB**

In the previous lab, you created three functions, clean\_text, tokenize and count\_words. You wrote them using some of the linux functions as a guide for comparison. Now it is time to formalize what you expect from those functions concretely by creating and definign tests around them.

* In your repository, create a new branch from week3. Let's be a little descriptive this time and use week4/testing\_word\_processors for the name of the branch.
* Create a new directory tests. This is in keeping with good practices so the tests are kept in their own dedicated folder.
* And continuing with best practices, create a file for each of your functions, i.e. tests/test\_tokenizer.py, (note that the file names **must** start with test\_)
* At this point, we need to reduce some of the degrees of freedom in our lab so we can get repeatable results. What I mean is that we are now going to fix which files are used in testing so we compare apples to apples. - 17192 The Raven
  + 932 Fall of the House of Usher
  + 1063 Cask of Amontillado
  + 10031 The Poems
  + 14082 And Le Corbeau. So we see how the tokenizer stands up to accented characters.
* We'll run some tests on the Raven single file, all 4 english titles, the French version of the Raven, and finally all of them together.
* I'm going to give you a passing test to get you started. I'll use a snippet from the Raven. You can follow the same pattern as you go along with your tests. Start with a sentence or two hard coded in the test, then raise that to a full file read in subsequent tests. Finally you can write tests for a set of files.

### **If my simple tokenizer in tokenizer.py is**

def tokenize(somestr):

return len(somestr.split())

### **Then my test in my tests/test\_tokenizer.py file might look like**

def test\_tokenize():

# Given a string \_text\_ of text with words

# When I pass \_text\_ to the `tokenize()` function

# I should get an int as return representing the number of blocks of continuous text in the string

text = 'philosophical prose poem of "Eureka," which he deemed the crowning work'

assert isinstance(tokenize(text), list), f"Tokenizer failed on sample text: {text}"

### **Similarly…**

def test\_word\_count():

# fill in the Given/When/Then

text = 'philosophical prose poem of "Eureka," which he deemed the crowning work'

assert 11 = word\_count(text), f"Tokenizer failed on sample text: {text}"

Note, that depending on what you require of your tests, and how you implement, one of these function may end up being a simple pass through, i.e. if your tokenizer is a list, maybe word count is just the length. But what if your tokenizer split off punctuation on their own…? The interplay between cleaning, tokenizing and counting should be clarified in your Given/When/Then.

# **And now, for the fun part**

* Use text = But the Raven, sitting lonely on the placid bust, spoke only That one word, as if his soul in that one word he did outpour. and write a test for each of your functions. - Write tests with Given/When/Then to describe what is expected from the function… all to upper case? lower? remove dashes? Can you do this in one test? do you need a test for different aspects?
  + Try to keep to one, two, maybe three asserts per test. In this case the less the better to keep things clear.
  + When you're done you should have at least a handful of tests for each of your functions against the text.
* Use a decorator and write a test for each of your functions against that one text string that is intended to **fail on purpose**
* Write a test for each of your functions now working against the whole 'The Raven' file
* Expand that, by following the parametrizing procedure in the book so you can pass in the list of files to a test to run each of your English files. I.e. a parameter to the test is a list of the file names, and for each name a test is run independently.
* Now write a test for ALL the English files together.
* Finally, write a test for Le Corbeau for each of your functions using the following as the text

\_Mais le Corbeau, perché solitairement sur ce buste placide, parla

ce seul mot comme si, son âme, en ce seul mot, il la répandait. Je ne

proférai donc rien de plus: il n'agita donc pas de plume--jusqu'à ce

que je fis à peine davantage que marmotter «D'autres amis déjà ont

pris leur vol--demain il me laissera comme mes Espérances déjà ont

pris leur vol.» Alors l'oiseau dit: «Jamais plus.»\_

The lab if you write working tests is 10 points. The following are worth a point each for a total of 15

* use the skip decorator for a test that hypothetically is expected to pass but can't be run, say we will eventually run a Japanese version but we are not ready yet…
* make a test continional on your OS, so if the tests are run on a different OS they fail to warn you have not tested on that OS
* Make a similar test for the version of python used when running the tests
* Write a test that uses bash/linux to get a result on a test string and compare it against your functions. The function should pass if the results are the same.
* Add a short description to your README to give an synopsis of what your functions do, and provide one example. Think of someone who has been pointed to your code, (not just a random person), and is looking to get a clear short direct description of what your functions do and how to use them.

# **When done:**

Please submit a LINK TO YOUR PULL REQUEST, with the properly named branch.

This is more than just tiny paperwork.  A lot of modern workplace automation ties github pull requests to your work tasks.  For example, your ticketing system, say 'Jira', may have a feature to write tests with an ID in it's database as FEAT-240610.  Most automation will key off the start of the branch name.  Further more, humans don't work that way, so in this case a branch name like FEAT-240610/add\_tests\_to\_tokenizer would be a good balance.  The start of the branch name would automatically tie the work item to your branch, and give an update to your leads and program managers.  In our case, our ticket number is simply week4

# **Link to Cloud Recording Office Hours, June 12**

Title: 24Su Data Engineering OFFICE HOUR  
Date: Jun 12, 2024 8:22 PM Eastern Time (US and Canada)

Recording-1(285 MB)  
https://virginia.zoom.us/rec/play/s3kK6NSu651M-hFdqgPaY\_ys7Sf8FZtIK11LAJArAjBzxFxsoR4PC8qljMgUTm9zYITP3c5nhVJdujmE.o\_QYcmq0S02pljQx

Passcode: Y9mA&.7p

**Week 5**

# **Learning Objectives**

# **CI/CD with Github Actions**

## **Introduction**

From the very early days of Agile programming, automation has been a core component. The automation of repetitive tasks helps reduce errors. It does so not just by reliably reproducing a long set of complex tasks, but also by removing the tedium of monotonous tasks.

Those two seemingly unrelated goals, to reproduce reliably and to releive us from monotony, are behind the almost fractal nature of CI/CD. Meaning, Continuous Integration and Condinuous Delivery.

Both these practices have one common goal, the delivery of quality software in the shortest possible time.

In this lecture we'll take a look at the large end to end loop of automation, and zoom in on the fractals down to the extreme in TDD, (Test Driven Development).

## **Learning Objectives**

At the conclusion of this module, you should understand:

* the large automation loops deliver fully functional software
* the small automation loops that test and integrate the separate pieces of sofware
* the way both the large and small loops are interconnected into a full CI/CD system
* how similar automation tools are to the makefile

At the conclusion of this module, you should be able to:

* Activate a GitHub repo to start the Github Actions/Workflows automation to set up your own CI/CD automation
* Add the tests you created in the previous lab to a Github Workflow so they always run
* Be able to run local tests in a 'short' loop during development, while having longer running tests run when you update your repository

# **Reading**

# **Week 5, CI/CD Github Actions Reading Assignment**

## [**Learning Github Actions, Brent LasterLinks to an external site.**](https://learning.oreilly.com/library/view/learning-github-actions/9781098131067/)

Chapters 2, 3, and Chapter 4, up to and including 'Committing the Initial Workflow'.

That's roughly 45ish pages in the printed book. Note that you should 'read to understand' chapters 2 and 3. When you get to chapter 4, zoom in your focus since that will be directly related to the hands-on part of the lab this week.

## **For reference:**

* https://docs.github.com/en/actions

Week 5- Github Workflow Automation

# **Week 5 Lab: Setting up your tests to run in Github Actions**

In this weeks lab we will take the test automation we created with pytest and the makefile and allow a Github Workflow to run it in demand.

During the lecture, I'll demo these steps briefly, so you'll be following my steps for the basic setup. Feel free to add more elaborate constructs, but this should get your foot in the door with creating a basic workflow.

## **Set up a quick github workflow to run our linter and tests**

* In the first part, we'll take the shortest path to get a workflow running.
* In the second part, we will add a badge to demostrate your tests are passing.

### **Procedure:**

Chapter 4 of the reading also focuses on this and has other variations.

* Create a directory .github/workflows, this is the default location where Github Actions will look for a yaml or yml file with instructions.
* Create file .githhub/workflows/validations.yml and put this content in it

name: Python package

on:

push:

workflow\_dispatch:

jobs:

build:

runs-on: ubuntu-latest

strategy:

matrix:

python-version: ["3.7"]

steps:

- uses: actions/checkout@v4

- name: Set up Python ${{ matrix.python-version }}

uses: actions/setup-python@v4

with:

python-version: ${{ matrix.python-version }}

- name: Install dependencies

run: |

make env

if [ -f requirements.txt ]; then pip install -r requirements.txt; fi

- name: Test with pytest

run: |

pytest -vvx tests/

* Go ahead at this time and check in all your code and push to the branch you are working in. Look up the docs here: https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-ac tions to learn more about github actions.
* Now go to your repository page in github, look in the options in the top menu. You will see Code, Pull Requests and then -> Actions. Click on that.
* If all went well, you should see an action triggered and running your make env, and make tests jobs. Explore a bit, dig into the output. You should eventually be able the output for your tests.
* If you don't see that and get stuck… let me know, we want to get everyone to this point soon so we can add more tests.

\*\*Up to this point in the lab is 5 pts.

## **Adding a badge. (1 Pts)**

* Follow the procedure in your reading, (at the end of the required reading), and add a badge to your repository's README.

## **Adding groups of tests (4 pts)**

In this section we'll combine what we've learned about pytest and github actions to customize our test setup.

* Add two more complicated tests that use your functions as a group. Perhaps tests that download, clean, tokenize and then take a count for a handful of common words.
* Add a pytest marker to label them as integration tests.
* On your makefile, update your test job so it runs ONLY the NON integration tests.
* Now add a step in your github workflow to run the NON integration tests, and a separate step that runs the INTEGRATION steps. So you get the full set, in two steps.

## **Going further (5 points)**

Now you can experiment to the fullest. Any one of these is another point. You can also do something outside of this group and just let us know to claim another point.

* Update the version of python used for the test, and add a pytest that will fail if another version is used
* Add an input to the workflow\_dispatch event so that you can toggle running the non-integration tests
* Execute a pip list on your local machine and get the versions of your python packages. Now add them to your requirements.txt.
* Add a test that fails on purpose, it should show the assert string for feedback. In other words, it should't just fail, but you should see some expectation that was not met as feedback.
* Add a new step, this step should only run if a previous step failed, (like the one you put in to fail on purpose), and in response, it should cat the logs to the screen to help in debugging.

# As usual, branch off last assignments branch and label this branch `WEEK-05/github\_automation`.  As mentioned before, branch naming is used for automation and human readability, so please follow that format.  I'll knock a point off for branches named `week5`, `github\_lab`, etc.   I'm looking for the `<PROJECT>-<ID>/<description>` format so often found in real settings.  Thanks!

# **Link to Cloud Recording: June 18 '24**

Title: DS5111 Summer 2024 Data Engineering  
Date: Jun 18, 2024 8:22 PM Eastern Time (US and Canada)

Recording-1(257 MB)  
https://virginia.zoom.us/rec/play/y4IoLbsMpnsSGnOADtXDxVR\_9H9ZYH3ldDhuO-rHEJu-CvDL3qSU8QI7\_cQ-RyD-ymfsnPIkUA8\_9GRq.wo81g8bMaLHytdbO

Passcode: dO0B\*Tc6

# **Office Hour June 20**

Title: 24Su OFFICE HOUR  
Date: Jun 20, 2024 8:20 PM Eastern Time (US and Canada)

Recording-1(133 MB)  
https://virginia.zoom.us/rec/play/OiriBZnCN14LygtQBCq-keiraJMCwTetjM1EKorqy20\_JO75tTZPfN-o\_O34xOMcgV\_AGVRPYjquUnfg.01QfBXYEE4Dv4W-U

Passcode: yCu.9P#k

WEEK 6

# **Learning Objectives**

# **Virtual Environments, Creating pip package**

We have been using virtual environments in the class but we have not really taken a close look at how they work. There are a handful of reasons why it's worth unpacking how they work. First, it is a great skill to have to be able to create a package that can be easily shared. The use case of sharing a common set of text cleaning functions was mentioned in class. Secondly, understanding how they work helps you develop faster by giving you insight to debug package conflicts. Finally, understanding the python virtual environment makes it easy, (by analogy), to understanding how linux security works, and subsequently, docker.

## **Introduction**

In this module:

* We'll review how python's venv and pip work under the hood to create your virtual environment.
* You'll see a demo of how to convert your functions from the ongoing lab into a pip installable package.

## **Learning Objectives**

At the conclusion of this module, you should understand:

* How python's virtual environment is structured; how it's shimmed in place; and where to find the code.
* The importance of versioning your packages, and python to help avoid breaking software
* The basic files needed to convert your code into a shareable package.

## **The Lab**

This weeks lab will be to add the files and structure necessary to convert your repo into a package installable by adding a line in a requirements.txt file

# **Reading**

# **Week 6, Virtual Envirnments, and creating a package Reading**

## [**Dead Symple Python, Jason McDonaldLinks to an external site.**](https://learning.oreilly.com/library/view/dead-simple-python/9781098156671/c02.xhtml#h1-500920c02-0003)

* Chapter 2 from **Packages and Virtual Environments** pg 18 to 36 (18 pages)
* Chapter 4 from **The Python Module Search Path** to end of chapter (3 pages only)

Week 6: Lab: Installable Package

# **In this weeks lab we will create a shareable package**

What you've written so far in you repo is a set of tested functions, that are using gh actions to rerun those tests at will.

Our refactoring of your code will result in a new structure. Your function code will end up in a new directory. This directory will be equipped with an \_\_init\_\_.py file which will be your interface to the external users of your package. You will also add a setup.py file, which will contain metadata about your newly formed package. After setting up the infrastructure, you'll then test it by installing via pip a test project. The user will be Mia and me, as well setup some code that will use your functions. We should be able to swap who's library we're using just by swapping out a line in the requirements.txt

## **How to do it.**

* Use https://github.com/EfrainOlivaresUVA/dpy8wq\_DS5111su24\_lab\_01 as a model
* Convert your existing repository into an installable package
* A good reference: https://packaging.python.org/en/latest/tutorials/packaging-projects/ There will be a series of steps and configuration, but it should not be that much.

1. After refactoring you should something like this:

.

├── README.md

├── makefile

├── pyproject.toml

├── requirements.txt

├── src

│ └── pkg\_dpy8wq

│ ├── \_\_init\_\_.py

│ └── example.py

└── tests

└── test\_dpy8wq.py

Note the new files \_\_init\_\_.py and setup.py

We'll go over this in class as a live demo. So you'll be able to follow along.

1. After you are done refactoring your repository

* Use this format in ANOTHER repository's virtual environment to load your package git+http://github.com/<github user id>/<repo name>@main

# **HOWEVER when you do release the package, use tags… never use a 'moving' pointer like 'main'.**

# **Week 7-Decorators, Comprehensions, Dicts and some OOP**

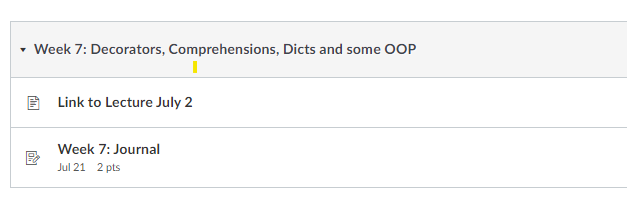
# **Link to Lecture July 2**

Title: DS5111 Summer 2024 Data Engineering  
Date: Jul 2, 2024 8:23 PM Eastern Time (US and Canada)

Recording-1(118 MB)  
https://virginia.zoom.us/rec/play/FEu-Snxx8zppEry12MqiUk0BNXOcQGiElMDh\_3NhSuxJfqS0Z7q0ggDh7J3B\_A7ck62mZeROEEl\_wAVy.EbESaECyFrM7OxQ1

Passcode: ciE$N5\*^

**Week 7:** Decorators, Comprehensions, Dicts and some OOP



**Week 8**: Linting, Refactoring, Intro Design Patterns

# **Design Patterns**

Design patterns are a higher level concept that can really make you code efficiently. At it's most basic Design Patterns are a language you use to define a problem and a known solution. It's a really powerful concept, and like all powerful things, if it is misunderstood they can cause more harm than good. So it is worth spending a little time just focusing on the concept, before even applying any coding techniques.

## **Introduction**

In this module:

* A quick discussion of what Design Patterns are and their origin
* A 'thought experiment' using an example from an unrelated field to clarify the meaning
* A demo of some fundamental design patterns implemented in python

## **Learning Objectives**

At the conclusion of this module, you should understand:

* What are Design Patterns
* The origin of Design Patterns, specifically that they are not native to Engineering
* Understand how the Singleton, Factory, and Template design patterns are work as examples

## **The Lab**

In this weeks lab we'll round out our existing lab by adding the linter to our growing repo. We will also plug it into the workflow.

We'll reserve working with some a design pattern for the **next** lab after you've had a chance to do the reading and have gone through the lecture.

# **Week 8. Data Engineering**

| :--: | | Data Engineering | | School of Data Science | | University of Virginia |

# **Announcements/Questions**

* Everything should be in canvas… I never had the time to setup a new repo
* This week office hours on THURSDAY (travelling on Wed night)
* \*

## **Questions?**

# **This weeks lecture**

## **REFACTORING (overview)**

* Quick overview of putting it all together for refactoring (LAB will focus on this)
  + Including linter and python versions

## **DESIGN PATTERNS (Focus rest of time)**

* What are they
* Some examples
* We'll do lab next week on these.

## **Questions?**

# **Linting**

## **Several tools available**

* pylint is very common and we'll use that.
* [blackLinks to an external site.](https://github.com/psf/black) is an example of a VERY opinionated linter

Linting = automating code cleanup and redability

* Refactoring = The practice of reviewing editing code withOUT changing behaviour

# **Refactoring**

## **REFACTORING**

The practice of reviewing editing code **WITHOUT** changing behaviour

Should be done periodically as part of

* make it run
* make it right
* make it light

**This cycle is made possible because of…. tests**

# **This weeks lab**

## **Combine tools and practices for refactoring**

* tests
* linter
* git checkins to keep FORWARD motion
* makefile for automation local code
* workflow for checking different versions of python

Note also, that we'll run our code in multiple versions of Python.

## **A LOT of bugs come in via version conflicts**

# **Design Patterns: In SW Eng**

## **Design Patterns book and "GoF"**

* [Elements of Reusable Object-Oriented SoftwareLinks to an external site.](https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=sr_1_2?crid=1TLEU3V5Q8N6L&dib=eyJ2IjoiMSJ9.mTRaTOPYqsPcUsGD8aznte8IMQZZiYjv7_Xwyyi1iS1e3pMr5B1lRIQbQQnfxyaU6oenbGuFExxxQadmnE6VmxNoJFeNxnaGIXzla_3hfuqD9JVmS45BlxftfhLBvYM5fsPwAEApT0XjB-ZB2ALoiYPkJpU1y9J-FnR0e7CK5OpUtqFzS6oPMoXPJql_3O48mOzrRptaV-8S97G-5diP01aQBLXaSI7_7Q2QZqTbRgQ.aoeDhKOssvVpnqpI9FNqmJdVttPVwNHRi7tYgV02NE8&dib_tag=se&keywords=design+patterns&qid=1720230242&sprefix=design+pattern%2Caps%2C183&sr=8-2)
  + Gamma
  + Helm
  + Johnson
  + Vlissides

GoF => Gang of Four

# **Design Patterns: What they are NOT**

# **Design Patterns are NOT**

* Object Oriented Design
* Reusable 'chunks of code'
* Software inheritance
* An original concept of Software Engineering

# **Design Patterns: Origins**

* [Timeles Way of BuildingLinks to an external site.](https://www.amazon.com/gp/product/0195024028/ref=ewc_pr_img_1?smid=ATVPDKIKX0DER&psc=1)
* [Wikipedia: Design PatternLinks to an external site.](https://en.wikipedia.org/wiki/Design_pattern#:~:text=A%20design%20pattern%20is%20the,other%20disciplines%2C%20particularly%20software%20engineering.)
  + A design pattern is the re-usable form of a solution to a design problem.

# **Design Patterns: An Example From Biology**

* Counter Example: Why do all Birds have wings?
  + This is inheritance
  + Yes, a solition is re-used but in same domain, so technically … maybe
* Why do Birds and Bats have wings?!
  + Birds and Bats have ancestors with no wings
  + They don't mate, which rules out inheritance

So… taking a closer look

* The ancestors of Birds and Bats encountered similar 'forces'
* Evolution reacted by providing a solution
  + shaped by the forces

# **Design Patterns: A Documented force-solution set**

* Document the context
  + hostile ground
  + need to stay in isolation for safety but get nourishment at distance
* Document the forces
  + cover distance without touching ground
  + shape limbs to leverage aerodynacs

Solution: Wings

That is a Design Pattern

# **Design Patterns: A Documented force-solution set**

A Design Pattern is not LITERAL

* You can't "paste" batwings on a bird or viceversa
* There are many ways to implement wings
  + Flying squirrels for example

Back to the Technical side

* They are not intended for 'cut & paste'
* Can be implemented with or without OOP
* Don't apply to a specific language (implementations, yes, but not the pattern)
  + [python implementationsLinks to an external site.](https://www.amazon.com/s?k=design+patterns+in+python&i=stripbooks&crid=3KLR82JZH53F5&sprefix=design+patterns+in+python%2Cstripbooks%2C178&ref=nb_sb_noss_2)
  + [javascriptLinks to an external site.](https://www.amazon.com/s?k=design+patterns+in+javascript&i=stripbooks&crid=2SI0DSLR0BLC4&sprefix=design+patterns+in+javascript%2Cstripbooks%2C174&ref=nb_sb_noss_1)
  + [rubyLinks to an external site.](https://www.amazon.com/s?k=design+patterns+in+ruby&i=stripbooks&crid=SBS5SMSZTGYR&sprefix=design+patterns+in+ruby%2Cstripbooks%2C172&ref=nb_sb_noss_1)
  + golang, java, nodejs etc etc

# **Design Patterns: Gang of Four foundation**

# **Design Patterns: Close up on some Patterns**

* Proxy

# **PROXY**

When you need an object to "negotiate" as an intermediary

Use Cases:

* Loading LOTS of images on a web page
* Answering API calls
* **Accessing a database**

proxy\_sample.py

\_\_author\_\_ = 'Chetan'

"""

Learning Python Design Patterns Second Edition (Packt Publishing)

Chapter 2: Classical Singleton Design Pattern

"""

class Actor(object):

def \_\_init\_\_(self):

self.isBusy = False

def occupied(self):

self.isBusy = True

print(type(self).\_\_name\_\_ , "is occupied with current movie")

def available(self):

self.isBusy = False

print(type(self).\_\_name\_\_ , "is free for the movie")

def getStatus(self):

return self.isBusy

class Agent(object):

def \_\_init\_\_(self):

self.principal = None

def work(self):

self.actor = Actor()

if self.actor.getStatus():

self.actor.occupied()

else:

self.actor.available()

if \_\_name\_\_ == '\_\_main\_\_':

r = Agent()

r.work()

# **SINGLETON**

When you need to make sure you have ONLY ONE of an object

Use Cases:

* **Accessing a database**

### **NB: Don't initialize outside of a function in a package!!**

classical\_singleton.py

\_\_author\_\_ = 'Chetan'

"""

Learning Python Design Patterns Second Edition (Packt Publishing)

Chapter 2: Classical Singleton Design Pattern

"""

class Singleton(object):

def \_\_new\_\_(cls):

if not hasattr(cls, 'instance'):

cls.instance = super(Singleton, cls).\_\_new\_\_(cls)

return cls.instance

s = Singleton()

print("Object created", s)

s1 = Singleton()

print("Object created", s1)

# **STRATEGY**

When you need to vary the algorithm

Use Cases:

* **Classify same data, different algoriithms**

strategy.py

"""

Mastering Python Design Patterns Second Edition

Chapter 13: Strategy Pattern

"""

import time

def pairs(seq):

n = len(seq)

for i in range(n):

yield seq[i], seq[(i + 1) % n]

SLOW = 3 # in seconds

LIMIT = 5 # in characters

WARNING = 'too bad, you picked the slow algorithm :('

def allUniqueSort(s):

if len(s) > LIMIT:

print(WARNING)

time.sleep(SLOW)

srtStr = sorted(s)

for (c1, c2) in pairs(srtStr):

if c1 == c2:

return False

return True

def allUniqueSet(s):

if len(s) < LIMIT:

print(WARNING)

time.sleep(SLOW)

return True if len(set(s)) == len(s) else False

def allUnique(word, strategy):

return strategy(word)

def main():

WORD\_IN\_DESC = 'Insert word (type quit to exit)> '

STRAT\_IN\_DESC = 'Choose strategy: [1] Use a set, [2] Sort and pair> '

while True:

word = None

while not word:

word = input(WORD\_IN\_DESC)

if word == 'quit':

print('bye')

return

strategy\_picked = None

strategies = { '1': allUniqueSet, '2': allUniqueSort }

while strategy\_picked not in strategies.keys():

strategy\_picked = input(STRAT\_IN\_DESC)

try:

strategy = strategies[strategy\_picked]

result = allUnique(word, strategy)

print(f'allUnique({word}): {result}')

except KeyError as err:

print(f'Incorrect option: {strategy\_picked}')

if \_\_name\_\_ == "\_\_main\_\_":

main()

# **TEMPLATE**

When you need to run the same steps to process similar objects

Use Cases:

* **Classify using same algorithm, different data INVERSE OF STRATEGY**

template\_simple.py

\_\_author\_\_ = 'Chetan'

"""

Learning Python Design Patterns Second Edition (Packt Publishing)

Chapter 8 Template Pattern

"""

from abc import ABCMeta, abstractmethod

class AbstractClass(metaclass=ABCMeta):

def \_\_init\_\_(self):

pass

@abstractmethod

def operation1(self):

pass

@abstractmethod

def operation2(self):

pass

def template\_method(self):

print("Defining the Algorithm. Operation1 follows Operation2")

self.operation2()

self.operation1()

class ConcreteClass(AbstractClass):

def operation1(self):

print("My Concrete Operation1")

def operation2(self):

print("Operation 2 remains same")

class Client:

def main(self):

self.concreate = ConcreteClass()

self.concreate.template\_method()

client = Client()

client.main()

# **FACTORY**

When you need to create sets of related objects

Use Cases:

* Replace production resources with 'stand ins'
* data cleaning/prep goes with specific algorityms
  + lemmatizing? or just counting words for example

simple\_factory.py

\_\_author\_\_ = 'Chetan'

"""

Learning Python Design Patterns Second Edition (Packt Publishing)

Chapter 3 Simple Factory Pattern

"""

from abc import ABCMeta, abstractmethod

class Animal(metaclass = ABCMeta):

@abstractmethod

def do\_say(self):

pass

class Dog(Animal):

def do\_say(self):

print("Bhow Bhow!!")

class Cat(Animal):

def do\_say(self):

print("Meow Meow!!")

## forest factory defined

class ForestFactory(object):

def make\_sound(self, object\_type):

return eval(object\_type)().do\_say()

## client code

if \_\_name\_\_ == '\_\_main\_\_':

ff = ForestFactory()

animal = input("Which animal should make\_sound Dog or Cat?")

ff.make\_sound(animal)

## **Design Patterns Reading Assignment**

* [Mastering Python Design Patterns - Second EditionLinks to an external site.](https://learning.oreilly.com/library/view/mastering-python-design/9781788837484/)
  + Preface
  + The Factory Pattern
  + Singleton
* [Design Patterns in Ruby Chapter 3. Varying the Algorithm with the Template MethodLinks to an external site.](https://learning.oreilly.com/library/view/design-patterns-in/9780321490452/ch03.html)

Mastering Python Design Patterns:

# **Preface**

Python is an object-oriented scripting language that is used in a wide range of categories. In software engineering, a design pattern is a solution for solving software design problems. Although they have been around for a while, design patterns remain one of the hot topics in software engineering and are a good source of information for software developers to solve the problems they face on a regular basis.

This book takes you through a variety of design patterns and explains them with real-world examples. You will get to grips with low-level details and concepts that show you how to write Python code, without focusing on common solutions as enabled in Java and C++. You'll also hunt sections on corrections, best practices, system architecture, and its designing aspects.

This book will help you learn the core concepts of design patterns and the way they can be used to resolve software design problems. You'll focus on all the Gang of Four (GoF) design patterns, which are used to solve everyday problems and take your skills to the next level with reactive and functional patterns that help you build resilient, scalable, and robust applications. By the end of the book, you'll be able to efficiently address commonly-faced problems and develop applications, and also be comfortable working on scalable and maintainable projects of any size.

Week 8: Refactoring and linting

**Due: Fri Aug 9, 2024 11:59pmDue: Fri Aug 9, 2024 11:59pm**

Ungraded, 10 Possible Points10 Points Possible

Attempt

In Progress

**NEXT UP: Submit Assignment**

Add Comment

**Unlimited Attempts Allowed**

Available: Jul 9, 2024 12:00am until Aug 9, 2024 11:59pmAvailable: Jul 9, 2024 12:00am until Aug 9, 2024 11:59pm

**Details**

# **Setting up a linter for your code**

In this lab we will set up pylint and we'll **refactor** our code. We'll set up a job in our makefile so we can cycle fast. Our goal will be to get as best a score out of 10 from the linter. At which point we'll set up the workflow to run in at least a couple of python versions.

## **Installing pylint**

You should have your virtual environment activated for this part.

* Add pylint to your requirements.txt. (I think we already did previously, but add if you don't see it add it now)
* Go ahead and run an update to make sure the requirements.txt is pip installed
* If you now activate your virtual environment you should have the pylint command, (just as you have the pytest command)

## **Setting up pylint config**

You can find the documentation for pylint here -> [pylint readthedocsLinksLinks to an external site.](https://pylint.readthedocs.io/en/latest/tutorial.html" \t "_blank).

You should be able to just run it directly on the new file, for example `pylint src/pkg\_/tokenizer.py. If you see a list of errors including things like indentation, then it's working!

From the root of your repository, (in the same directory as your makefile), run pylint --generate-rcfile >> pylintrc. This creates a config file that will be read by pylint (it will look for this in your current directory first, then in HOME etc).

If you run the pylint command again on your file it should now be using the config file. To test that your config is actually being used, open the pylintrc file and search for indent-string=' '. That should have 4 spaces in the string on the right of the = sign. Change that to two spaces and run again. You should see a change in the error log depending on the number of spaces. Of course, this is not what we want, so set it back to 4 spaces, we just wanted to be sure pylint is using it.

Make a new job in your makefile, lint: so you can just re-execute that to lint your file. Remember to activate the virtual environment first, just like we did before for the tests. Now link that job so the tests: job depends on it.

## **Getting the code up snuff**

After running the test/lint jobs, at the bottom of the pylint output you should see something like

------------------------------------------------------------------ Your code has

been rated at 2.86/10 (previous run: 3.57/10, -0.71)

Your mission here is to go through this cycle:

* Run the makefile, your tests should be GREEN, but your lint score will be low
* Edit a file and resolve an issue
* Run the makefile again, tests should be GREEN, lint score should be better
* If you see an improvement in the linting, git commit your code and push
* Not start over again, until you get a 10/10 (or as close as you can get)

Use the documentation, https://pylint.readthedocs.io/en/latest/user\_guide/checkers/features.html in pylint so you can make sense of the feedback.

At the end, you should have all your files passing the linter with a Your code has been rated at 10/10.

## **Set up the github workflow**

Just as we did for the tests, set up a step to run the linter as well.

This is an area where you can see the combination of makefile and workflow working together. The makefile made it easier for you to cycle during refactoring. Now the workflow will easily expand this because you can add different versions of python.

# **A LOT OF ISSUES COME IN VIA VERSION CONFLICTS**

This is invaluable when something breaks in production with the SAME version of your code. The environment can cause failures too.

## **POINTS 10**

You should have

* All your tests be GREEN
* Your linter at 10/10
* At LEAST 2 versions of python, (3.7 is deprecated so make that 3.8 and ?)

# **Link to Lecture**

Title: DS5111 Summer 2024 Data Engineering  
Date: Jul 9, 2024 8:20 PM Eastern Time (US and Canada)

Recording-1(144 MB)  
https://virginia.zoom.us/rec/play/\_GuxsvUhnzPPEKzhCQUR6mfxBSiX6rAx2xxHhcf9zpSxIXaRgOhixPrxRTZn9HYSAwfM3YnmJiQnVvsM.qBKzG29ZIwJ1PuR8

Passcode: tjp2zz\*q

]

Week 9 DataBase Design

# **Module 9: Database Design**

## **Introduction to the Module**

Properly designing a database makes it more useful. This module discusses the important considerations and actions to take in designing a database.

Various database components are introduced and discussed, including keys, constraints and indexes.

Since database design for a client is essentially a translation problem, we discuss how to translate the problem into a database that meets the requirements.

## **Learning Objectives**

At the conclusion of this module, you should be able to:

1. Understand what makes something a database (the CRUD properties)
2. Understand how relational and non-relational databases differ
3. Demonstrate how to apply fundamental database objects including tables, keys and constraints
4. Explain the difference and use of a primary key and foreign key
5. Explain how to learn the requirements to build a suitable database
6. Understand how to write a database requirements document

## **To-Do**

* Reading Assignment
* Course Content
* Quiz
* Homework
* Journaling
* Live Session

**Reading Assignment**

### [**Database Design Reading ListLinks to an external site.**](https://github.com/UVADS/data_engineering#database-design-reading-list)

Book: [Beginning Database Design SolutionsLinks to an external site.](https://learning.oreilly.com/library/view/beginning-database-design/9780470385494/)

* Ch1 Goals of Effective DB Design pp3-21
* Ch2 Database Types pp27-30 (optional: entire chapter)
* Ch3 Relational DB Fundamentals pp49-59
* Ch4 Understanding User Needs pp65-86
* Ch5 Translating User Needs into Data Models pp89-117
* Ch7 Normalizing Data pp137-154

MODULE 10 – Snowflake

## **Introduction to the Module**

Snowflake, unlike other tools we've looked at so far, this tool is pure SAAS (Software As A Service). There is no software to install. You sign up on the web to gain access to the Snowflake appication. You don't need to configure a database, or servers. All of that is taken care of behind the scenes. It is this simplicity that has propelled it to be one of the top tools in Data Science/Analysis/Engineering. It is essentialy a database that can expand in both storage and processing power from the start. This week we'll load our data from the database design lab and use Snowflake to process it into a realization of the ERD diagram.

Snowflake offers a two week trial period. But for our lab we'll use a pre-existing account so we can all have views into each others work.

## **Learning Objectives**

At the conclusion of this module, you should be able to:

1. Load, and unload structured csv data into Snowflake.
2. Query data in snowflake.
3. Create and use worksheets in snowflake to manipulate your data

## **Module 10 To-Do**

* Take quiz
* Reading
* Complete Lab
* Review Slides
* Journaling
* Live Session

Reading from Snowflake Essentials

* Chapter 3 Snowflake Data Cloud Architecture
* Chapter 4 Snowflake Web Interface: Classic Console
* Chapter 8 Database Objects
* Chapter 9 Querying and Cloning Data In Snowflake
* Chapter 11 Semi-structured Data in Snowflake
* Chapter 12 Loading Data

# **Lab: Database Design and Build - Part 2**

This weeks lab is a continuation of Database Design and Build from last week.

Use Snowflake to upload your raw csv.

* Please name them with `\_csv` appended to the end of the tables so I can see they are part of the original set.
* Then create new tables by joining or modifying the original tables.  This should follow your ERD.  Please append `\_ERD` to these tables
* Finally, create SQL statements to answer the questions in part 2 or the lab.   Please add the file, `db\_design.sql` to the same repository from last week.

Grading:  Both part 1 and part 2 (this assignment) are graded together.  This is just a continuiation of the DB Design and Build lab.