



Learning Polars with Python

Section 1



Hanjo Odendaal

LEAD DATA SCIENTIST (71POINT4)

ABOUT ME

I lead the advanced data analytics and statistical modelling aspects of the work at 71point4. I am passionate about exploring different methodologies to collect and analyse new and alternative data sets.

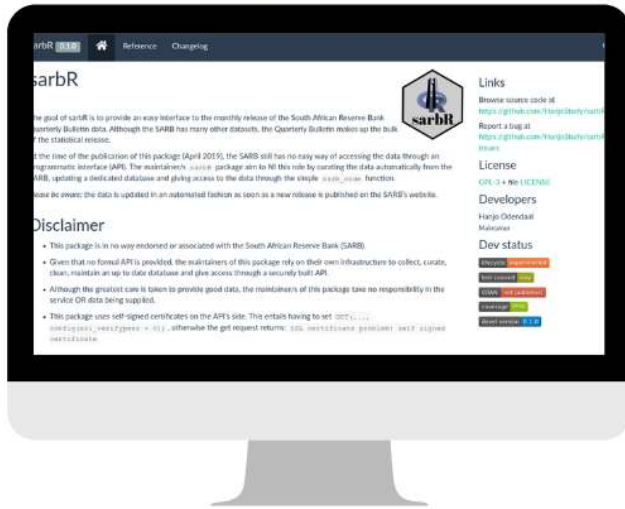
I hold a PhD in Economics from the University of Stellenbosch: News, Sentiment and the Real Economy.



Hanjo Odendaal

LEAD DATA SCIENTIST (71POINT4)

Software
Engineering



High Performance
Cloud Computing



Production Machine
Learning



Web Scraping



Agenda

- 1) About this Course
- 2) Software Requirements
- 3) Why Polars



About this Course



What this course aims to achieve

What the course aims to achieve:

On completion of the workshop, participants should be able to (1) interact with data using polars, (2) use the tidy interface, and utilize the python dbutils package to load data from mysql.

- Very few organizations need machine learning engineers, but all of them need a data team that communicates effectively and have the necessary skills to perform basic data tasks. Getting teams to understand the broader problem each department faces solves 80% of the frictions encountered when delivering insight from data.

What the course does **NOT** aim to achieve:

It will NOT turn individuals with varying backgrounds, skills and motivations into fully-fledged Data Scientists. This course does not cover statistical languages and the interplay between databases and Python

- We wish to elevate people's knowledge and exposure to basic data science principles to help guide them on their data journey.

Key outcomes

You should:

- Basic wrangling in python using `polars`.
- Be able to query a database and do *basic* aggregations.
- Understand how one can build python packages.

We also encourage the following behaviour throughout the course:

- Learn from each other and share knowledge in groups.
- Ask questions during the course - the instructor has a lot of knowledge that you should tap.



Session Breakdown: Day 1 - Linux Environment

Session 1 (08:30 to 10:30) 🧑 & 💻:

- Course introduction.
- Install software for course.
- Linux basics review
- Learning about virtual environments.


Session 2 (11:00 to 13:00) 🧑 & 💻:

- Quarto and VSCode

Session 3 (14:00 to 16:30) 💻

- Basics of Polars 🐻


Session Breakdown: Day 2 - Deeper into polars

Session 1 (08:30 to 10:30) :

- Selecting, filtering interfaces
- Transformations


Session 2 (11:00 to 13:00)  & :

- Transformations
- Aggregations

Session 3 (14:00 to 16:30) :

- Aggregations
- Joins


Session Breakdown: Day 3 - Deeper into polars

Session 1 (08:30 to 10:30) :

- Homework
- Optimizations


Session 2 (11:00 to 13:00)  & :

- Tidypolars


Session 3 (14:00 to 16:30) :

- Tidypolars


Session Breakdown: Day 4 - Application

Session 1 (08:30 to 10:30) :

- Mini project


Session 2 (11:00 to 13:00)  & :

- Presentations



Session 3 (14:00 to 16:30) :

- Development in Python


Session Breakdown: Day 5 - Software

Session 1 (08:30 to 10:30) :

- Gorilla Methods and Classes

Session 2 (11:00 to 13:00)  & :

- Connection and installation

Session 3 (14:00 to 16:30) :

- Basic querying

System Setup

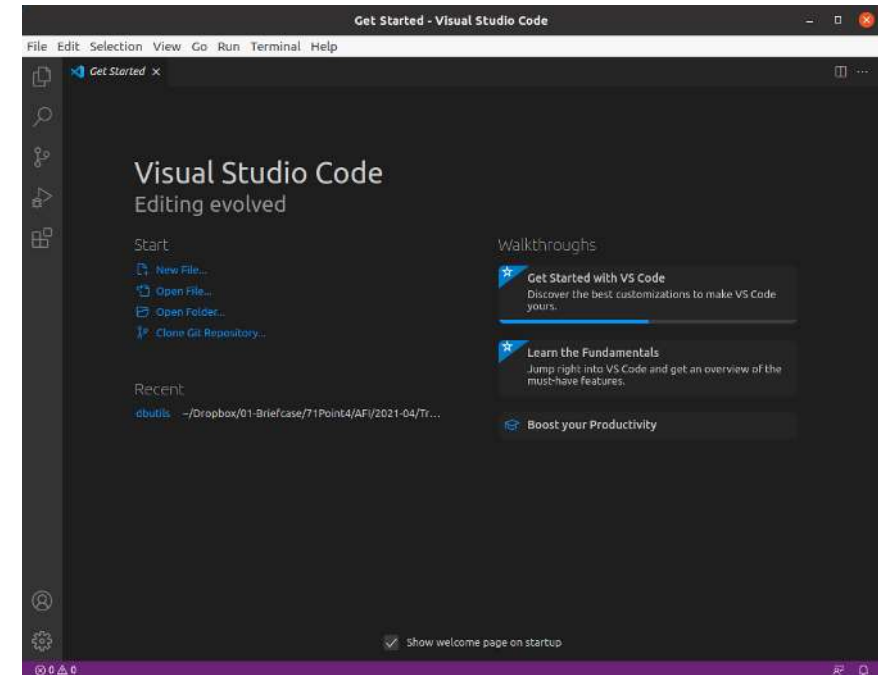


Learning to code in VSCode

Why switch from RStudio to VSCode for SQL development?

The first few things we are gonna do in VSCode is:

- Interact with a remote server
- Connect to database on remote server
- Execute code and download results



Connecting to remote development environment

As in most instances, you will likely be developing code on a remote machine, but would like to use VSCode as your development environment.

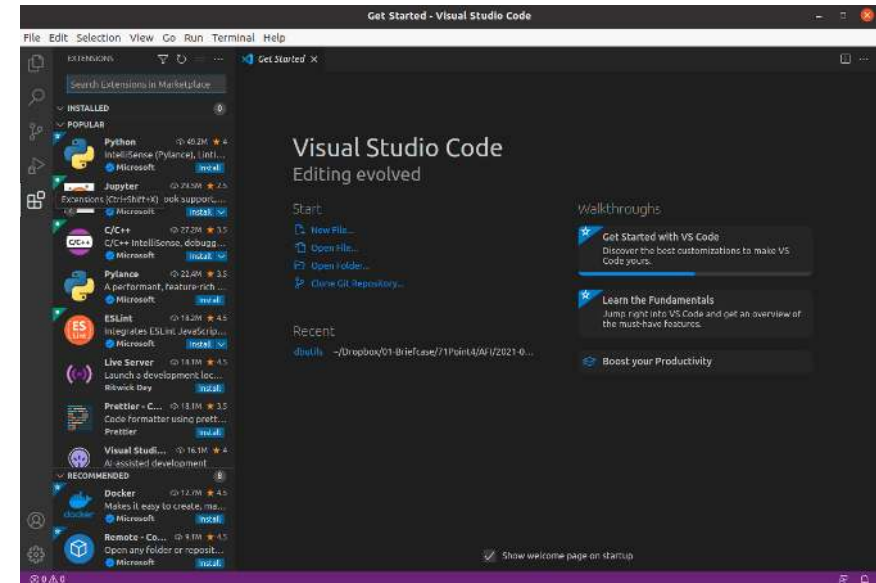
This can easily be achieved using the `Remote-SSH` feature in the IDE. This allows for:

- Develop on the same operating system you deploy to or use larger, faster, or more specialized hardware than your local machine.
- Quickly swap between different, remote development environments and safely make updates without worrying about impacting your local machine.
- Access an existing development environment from multiple machines or locations.
- Debug an application running somewhere else such as a customer site or in the cloud.

Connecting to remote development environment

One of the most used shortcuts in `vSCoDe` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



Connecting to remote development environment

One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

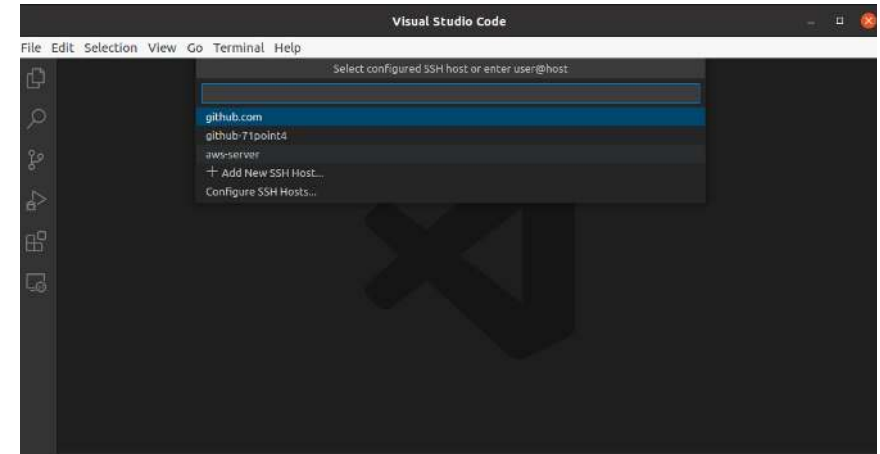
- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file

```
Host aws-server
  HostName
  IdentityFile ~/.ssh/
  User
  IdentitiesOnly yes
```

Connecting to remote development environment

One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

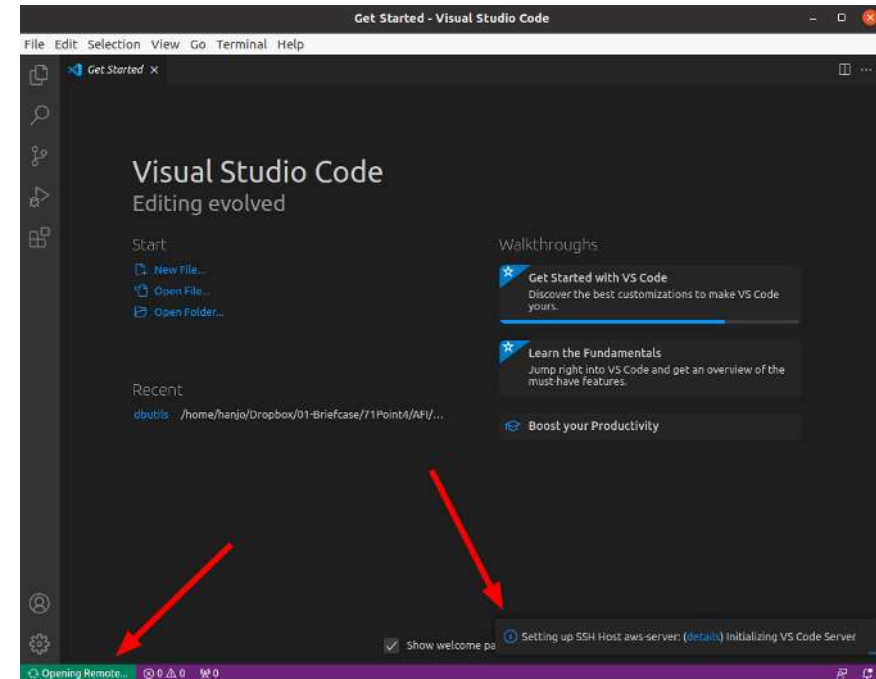
- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



Connecting to remote development environment

One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

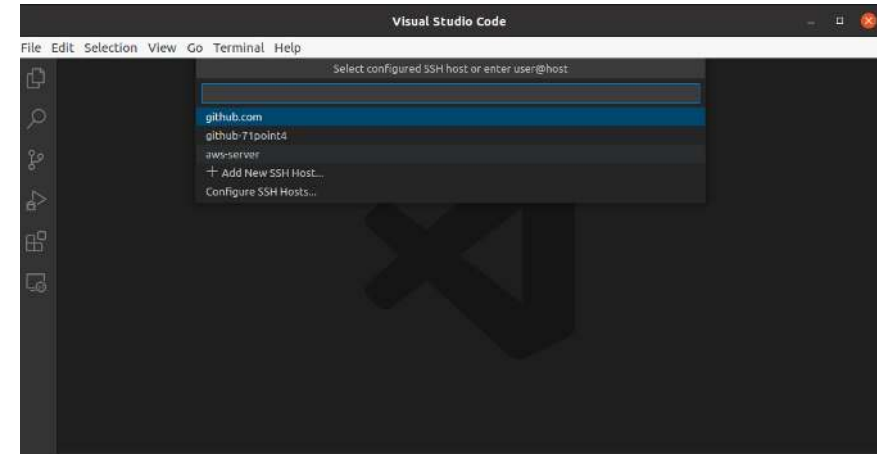
- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



Connecting to remote development environment

One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



Install OpenSSH for Windows

- To install OpenSSH using PowerShell, run PowerShell as an Administrator. To make sure that OpenSSH is available, run the following cmdlet:

```
Get-WindowsCapability -Online | Where-Object Name -like 'OpenSSH*'
```

- Install the OpenSSH Client

```
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
```

- Test the service

```
ssh 183.204.102.12\ubuntu@servername
```



Logging into Server



What is shell?

Whenever we talk about *black screen*, *command line* or *shell* we are essentially talking about the interface that takes input from the keyboard and sends it to the operating system (OS).

Almost all Linux distributions supply a shell program from the GNU Project called `bash` that looks like this:

```
hanjo@optimus:~$ penguin
```

This interface is called *shell prompt* and usually contains `username@machinename:directory`. If the last character of the prompt is a hash mark (`#`) rather than a dollar sign (`$`), the terminal session has superuser privileges (a little bit more on this later).

- Pressing the up 👆 arrow on your keyboard goes into your command history.
 - Be aware that history stores about 1,000 commands.

Different type of users

Superuser (root)

With great power comes great responsibility!



Different type of users

Superuser (root)

With great power comes great responsibility!

On a Linux system Superuser refers to the root user, who has unlimited access to the file system with privileges to run all Linux commands.

- This responsibility is mostly given to experienced SysAdmins. The reason being there is no "take-backsies" in linux. Once a command has been executed under `sudo` (superuser do) , there is almost never a way to reverse the execution (ex. deleting a file).
- The Superuser/Root is also responsible for setting up security and thus, limiting the power to a single (or very few individuals is preferred).



Basic shell commands

Try these basic commands:

```
date
```

```
## Thu 30 May 2024 15:56:53 SAST
```

```
free -h
```

```
##          total        used        free      shared  buff/cache   available
## Mem:          62Gi       7.7Gi       41Gi        2.9Gi        13Gi        51Gi
## Swap:         19Gi          0B       19Gi
```

```
cal
```

```
##          May 2024
## Su Mo Tu We Th Fr Sa
##          1  2  3  4
##  5  6  7  8  9 10 11
## 12 13 14 15 16 17 18
## 19 20 21 22 23 24 25
## 26 27 28 29 30 31
##
```

Welcome to your new home

Welcome to your new home, or `127.0.0.1` as I would like to call it.

```
hanjo@optimus:~$ ls -lart
## total 20
## drwxr-xr-x 6 root  root  4096 Dec 19 13:05 ..
## -rw-r--r-- 1 hanjo hanjo  807 Dec 19 13:05 .profile
## -rw-r--r-- 1 hanjo hanjo 3771 Dec 19 13:05 .bashrc
## -rw-r--r-- 1 hanjo hanjo  220 Dec 19 13:05 .bash_logout
## drwxr-xr-x 2 hanjo hanjo 4096 Dec 19 13:05 .
```

- Can anyone tell me what they think the `-rw-r--r--` stands for?

Although we will not go deep into security in this course, it is good to understand some basics.

Permissions

Owners, Group Members, and Everybody Else

One of the fundamentals that were built into Linux systems from the start is the concept of it being a *multiuser* system. This means that multiple users can log into the system at the same time without interfering (mostly) with each others processes and files.

In the Linux security model, a user may *own* files and directories.

- When a user owns a file or directory, the user has control over its access.
- Users can, in turn, belong to a group consisting of one or more users who are given access to files and directories by their owners.
- An owner may also grant some set of access rights to everybody, which in Linux terms is referred to as the world.

Permissions

Owners, Group Members, and Everybody Else

How does this look for the user I just created?

And for Superuser `ubuntu`?

```
ubuntu@optimus:~$ id ubuntu
## uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(au
```



Basic Commands



Listing directories

To find out where in the `tree` you are, we can use a simple command called: `pwd`

```
hanjo@optimus:~$ pwd
## /home/hanjo
```

Upon logging into a system, the terminal will always set your working directory to `home` also known as `~`.

- If you log in as a regular user, your `home` directory is the only place where you will be able to write and create files.

So, now that we are in the system, what directories are in my `home` folder *?

```
hanjo@optimus:~$ ls
## Data Desktop Documents Pictures
```

To list the files and directories in the current working directory, we use the `ls` command. This command is very versatile as you will see in a minute.

* Yours might look a bit different depending whether you are running Linux on a server or a desktop.

Changing the current working directory

Obviously looking at files in your `home` directory doesn't take you very far. We need to be able to navigate the file system in a quick and efficient manner.

The `cd` command in Linux is a powerful way to navigate the tree folder structure that is the file system.

```
hanjo@optimus:~$ cd Data
hanjo@optimus:~/Data$
```

The two main methods for traversing the tree is: (1) Absolute Paths and (2) Relative Paths:

- **Absolute Paths** begins with the root redirectory `/` and expands to the folder you are interested in: `/home/hanjo/Data`
- **Relative Paths** starts at the working directory and starts navigation from there. These paths have a special notation, a single dot (`.`) and a dot dot (`..`). The `.` notation refers to the working directory, and the `..` notation refers to the working directory's parent directory.

Changing the current working directory

Lets see an example of the **absolute** and **relative** path in action. Start by navigating the `/usr/bin` directory and listing all the files.

```
hanjo@optimus:~$ cd /usr/bin
hanjo@optimus:/usr/bin$ pwd
#/usr/bin
hanjo@optimus:/usr/bin$ ls
## 2to3-2.7 funzip mpiCC splitfon ...
```

Now lets move to the `/usr` directory from our working directory `/usr/bin`. There are two ways to do this, either **absolute** (`cd /usr`) or **relative**. Let us practice using the **relative** method.

```
hanjo@optimus:~$ cd /usr/bin
hanjo@optimus:/usr/bin$ cd ..
hanjo@optimus:/usr$ pwd
# /usr
hanjo@optimus:/usr$ ls
# bin/ games/ include/ lib/ lib32/ local/ sbin/ share/ src/
```

Changing the current working directory

There are also some nice shortcuts to be aware of:

- Change the working directory to your home directory: `cd ~`
- Change the working directory to the previous working dir: `cd -`
- Change the working directory to a specific user: `cd ~ubuntu`

Notes about filenames in Linux

Filenames in Linux are quite special and if you have worked closely with someone who works in Linux, you would have noticed some things. First and foremost:

- NEVER use a space in filenames use an underscore (`_`) instead - thank me later ;-)
 - ex. `this file Name SUCKS 1/30/23.txt` where `this_is_much_better.txt`
- Filenames that start with a `.` are hidden files. The `ls` command will not list these unless you use a *parameter* `ls -a`. These files usually relate to configuration settings.
 - ex. `.bashrc`.
- CASE MATTERS, so dont ever use Capitals for folders or filenames - it gets confusing.
 - ex. `This/path/IS/different/`. from `/this/path/is/different/`
- Linux does not have any concept of "file extensions". So remember to name your files in an appropriate manner if you would like them to be readable by the correct application.
 - ex. `mypdffile` and `mypdffile.pdf` is the same

See [this presentation](#) by Dr. Anna Krystalli for further tips on file naming.

Getting to know 'ls'

The `ls` command is probably one of the most used commands that any Linux user will encounter from day to day. As you will come to see, it is also one of the most powerful commands.

Let's start by listing the contents of `/usr` while our working directory is `~`:

```
hanjo@optimus:~$ ls /usr
# bin/  games/  include/  lib/  lib32/  local/  sbin/  share/  src/
```

You can also ask for multiple directories in a single line:

```
hanjo@optimus:~$ ls /usr ~
## /home/hanjo:
## Data Desktop Documents Pictures
##
## /usr:
## bin games include lib lib32 local sbin share src
```

Options and Arguments

By now you should have noticed once or twice that I have added an *options* parameter to my commands: `command -options arguments`. Type `man ls` to see all options for the `ls` command.

```
hanjo@optimus:~$ ls -l
## total 16
## drwxrwxr-x 2 hanjo hanjo 4096 Dec 20 09:23 Data
## drwxrwxr-x 2 hanjo hanjo 4096 Dec 20 09:23 Documents
```

My favourite command is `ls -lart` which stands for "list ALL the contents in REVERSE order SORT BY TIME".

```
hanjo@optimus0:~$ ls -lart
## total 40
## drwxr-xr-x 6 root root 4096 Dec 19 13:05 ..
## -rw-r--r-- 1 hanjo hanjo 807 Dec 19 13:05 .profile
## -rw-r--r-- 1 hanjo hanjo 3771 Dec 19 13:05 .bashrc
## -rw-r--r-- 1 hanjo hanjo 220 Dec 19 13:05 .bash_logout
## -rw----- 1 hanjo hanjo 26 Dec 19 13:35 .bash_history
## drwxrwxr-x 2 hanjo hanjo 4096 Dec 20 09:23 Documents
## drwxrwxr-x 2 hanjo hanjo 4096 Dec 20 09:23 Data
## drwxr-xr-x 6 hanjo hanjo 4096 Dec 20 09:23 .
```

Creating files and folders

Apart from knowing how to navigate folders, we must also know how to create files and folders.

The basic commands for this is:

- Create folder

```
mkdir scripts  
mkdir scripts data analysis
```

- Create file

```
touch analysis.R
```

90% of the time you will be using the basic versions of these commands. But they can also do some pretty interesting things.

Tricks and Tips for mkdir

- Create folders within folders that do not already exist (recursively create).

```
mkdir project/analysis/scripts
# mkdir: cannot create directory 'project/analysis/scripts': No such file or directory

# Correct usage
mkdir -p project/analysis/scripts
```

- What if I wanted to create a `data`, `scripts` and `output` folder in a single line?

```
# Note, there is NO spaces in the array
mkdir -p project/analysis/scripts/{data,scripts,output}
cd project/analysis/scripts/ && ll
```

- Current date in directory name

```
mkdir `date '+%Y%m%d'`
```

Viewing contents of files

To view the contents of a file, we use a program called `less`.

The `less` program was designed as an improved replacement of an earlier Unix program called `more`. The name `less` is a play on the phrase "*less is more*" — a motto of modernist architects and designers.

`more` (developed in 1978) was replaced by `less` in 1983, first and foremost because `more` could only scroll forwards through a text file. `less` was written by Mark Nudelman and is currently being maintained by him to this day!

- Backwards movement
- Searching and highlighting
- Multiple files
 - Less allows you to switch between any number of different files, remembering your position in each file. You can also do a single search which spans all the files you are working with.
- Advanced features
 - You can change key bindings, set different tab stops, set up filters to view compressed data or other file types, customize the prompt, display line numbers, use "tag" files, and more.

<http://www.greenwoodsoftware.com/less/faq.html#mail>

Viewing contents of files

Lets start by looking at the users on the system:

```
hanjo@optimus0:~$ less /etc/passwd
```

Navigation:

- **G** - Move to the end of the text file
- **g** - Move to the beginning of the text file
- **10g** - Move to the nth line
- **q** - Exit

Forward Search:

- **/characters** - Search forward
- **n** - Search forward
- **N** - Search backwards

Useful options for Less

Squeeze Blank Lines:

- The `-s` (squeeze blank lines) option removes a series of blank lines and replaces them with a single blank line.

Viewing Multiple Files:

- `less file1.txt file2.txt`
- To view the next file, press `:` and then hit `n`.

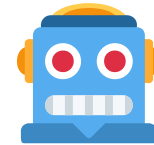
Mark places:

- Press `m` and then a letter, example: `a`. To return to that mark press apostrophe `'` and `a`.

Switch to editor:

- Pressing `v` while in `less` pushes you to default editor.
- `sudo update-alternatives --config editor`

Redirection in Linux



Redirection & Piping

This is maybe one of the coolest features of command line that you will learn: *Redirecting* or *piping* your results into another command. The *Input/Output* allows us to chain together commands and build pipelines of instructions.

- I/O redirection (`>`) allows us to change where output goes and where input comes from.
 - A good example of this would be the `ls` command we learned earlier.

```
hanjo@optimus0:~$ ls -l
hanjo@optimus0:~$ ls -l > all_files.txt
hanjo@optimus0:~$ less all_files.txt
```

We can also append a file using (`>>`):

```
hanjo@optimus0:~$ ls -l >> all_files.txt
hanjo@optimus0:~$ ls -l >> all_files.txt
hanjo@optimus0:~$ ls -l >> all_files.txt
hanjo@optimus0:~$ less all_files.txt
```

Redirection & Piping

In the previous examples we redirected only the `stdout` of the command. But, we sometimes also want to redirect the errors or Standard Error (`stderr`).

To do this we add an additional command to the end of the line (`2>&1`):

```
hanjo@optimus0:~$ ls -l > all_files.txt 2>&1
hanjo@optimus0:~$ less all_files.txt
```

We redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation `2>&1`.

Once we know the concept of standard output and input, we can start stringing commands together. These are called *pipelines* and it looks this, `command1` pipes into `command2`:

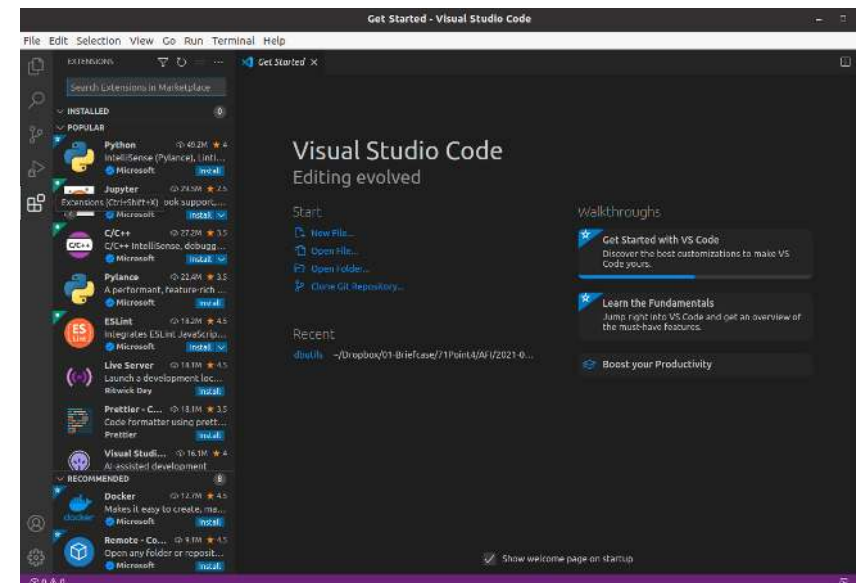
```
command1 | command2
```

Here we can see that `command2` takes `command1`'s output as its input. As you get more comfortable with the terminal, these become core concepts you will use every day.

Installing the recommended Extension

Installing *Extensions* in VSCode is pretty straight forward. Just navigation to the search tab using GUI. Then search and install the following:

- Remote -SSH
- Rainbow CSV
- autopep8
- R Extension for Visual Studio Code
- Spelling Checker for Visual Studio Code
- SQLTools



- Linux shortcut

```
wget -O extentions.sh https://bit.ly/3GrF5kn  
bash extentions.sh
```

Getting ready for

We will be working in `VSCode` using what's called `Workspaces`. But the first step is to setup your folder structure.

```
hanjo@optimus0:~$ mkdir -p ~/projects/polars
```

Next open the folder:

- File > Open Folder

How is that for setup?



Markdown



What is Rmarkdown/Quarto?



R Markdown wizard monsters creating a R Markdown document from a recipe. Art by [Allison Horst](#)

What is markdown?

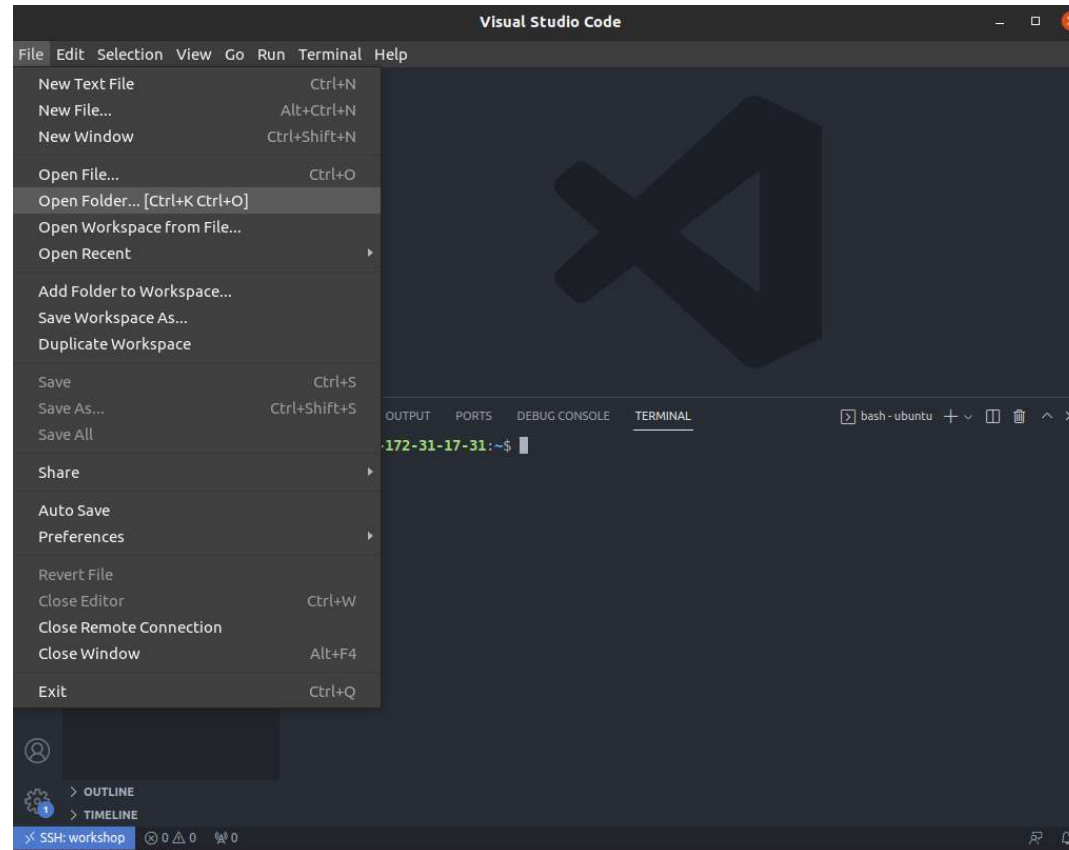
Markdown is a lightweight markup language for creating formatted text using a plain-text editor. *John Gruber* and *Aaron Swartz* created Markdown in 2004 as a markup language that is appealing to human readers in its source code form. Markdown is widely used in blogging, instant messaging, online forums, collaborative software, documentation pages, and readme files.

— *Wikipedia*

- Abstraction layer *above* certain compiling formats such as PDF, HTML, Word (XML).
 - This is pretty cool as you only have to learn the very basic syntax of markdown to be able to convert your document to any of the formats.
- `Rstudio` uses a productive notebook interface (called *Rmarkdown*) to weave together narrative text and code to produce elegantly formatted output.
 - Great thing is it supports over 51 languages. Main ones are `R`, `python`, `shell` and `SQL`.
- *Rmarkdown* has recently been 'replaced' with *Quarto* which works in VSCode!

Understanding markdown in VSCode

Start by opening a new Quarto file (`.qmd`) in a folder called `projects/polars`.



Understanding markdown in VSCode

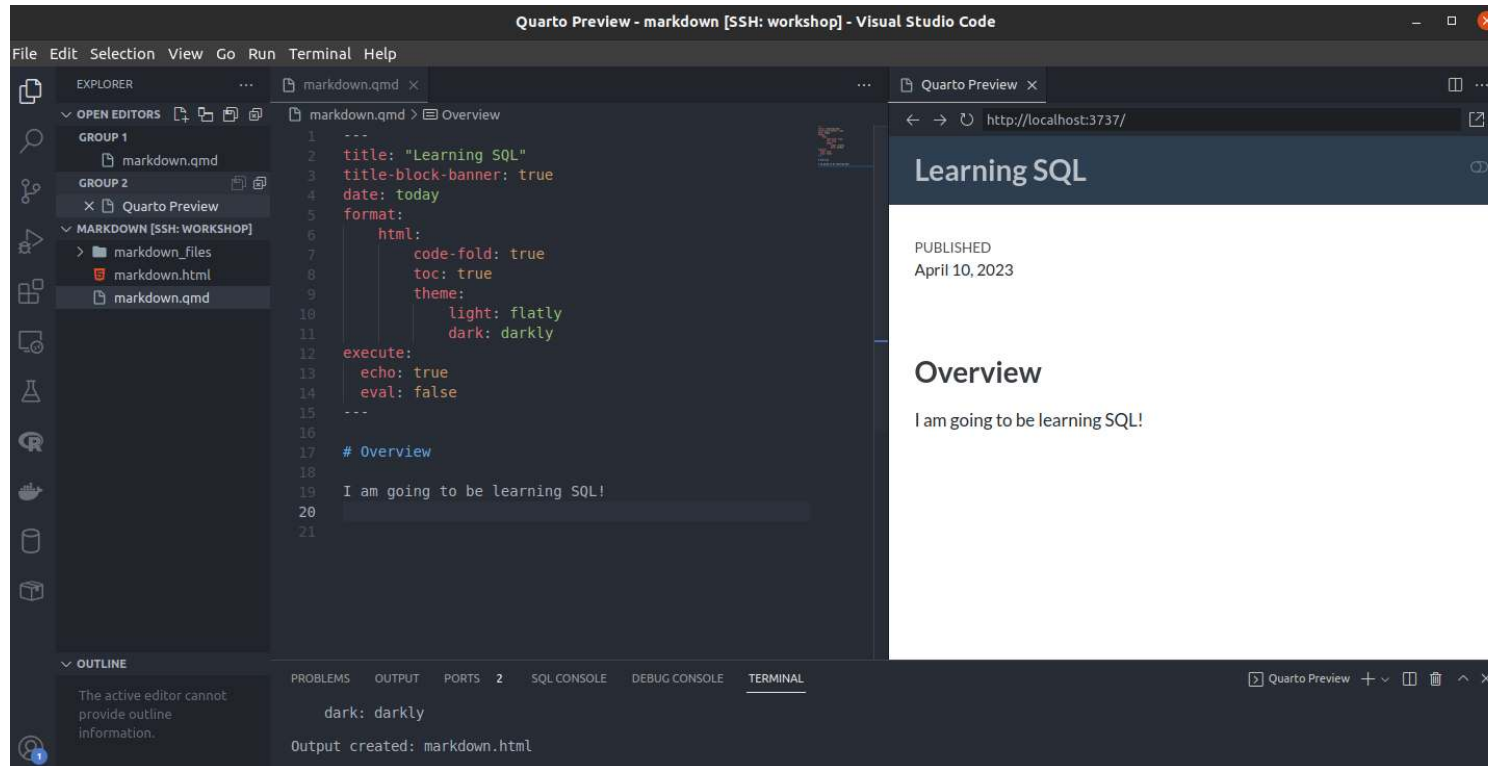
Add to a new file: README.qmd

```
---  
title: "Learning Polars"  
title-block-banner: true  
date: today  
format:  
  html:  
    code-fold: true  
    toc: true  
    theme:  
      light: flatly  
      dark: darkly  
execute:  
  echo: true  
  eval: false  
---  
  
# Overview  
  
I am going to be learning Polars!
```

Understanding markdown in VSCode

We need to `render` our documents in order to produce the output.

- Press the `render` button at the top OR (be cool) and use `CTRL + SHIFT + k!`



Components of markdown

The image shows a Visual Studio Code editor window titled "markdown.qmd - markdown [SSH: workshop] - Visual Studio Code". The editor displays a file named "markdown.qmd" with the following content:

```
1 ---
2 title: "Learning SQL"
3 title-block-banner: true
4 date: today
5 format:
6   html:
7     code-fold: true
8     toc: true
9     theme:
10      light: flatty
11      dark: darkly
12
13 execute:
14   echo: true
15   eval: false
16 ---
17 # Overview
18
19 I am going to be learning SQL!
20
21 ```{sql}
22 SELECT * FROM some table
23 ```
24
25 ▶ RunCell
26 ```{r}
27 #| eval: true
28 #| fig.height: 3
29 #| fig.width: 3
30 plot(rnorm(100))
31 ```
```

Annotations in the image identify three components of the file:

- YAML:** Lines 1 through 16, which define the document's metadata and execution options.
- Markdown:** Lines 17 through 20, which contain the main text of the document.
- Code Chunk:** Lines 21 through 31, which contain a code block for a SQL query and an R plot.

The right-hand pane shows a preview of the rendered document at <http://localhost:4593/>. The preview displays the following content:

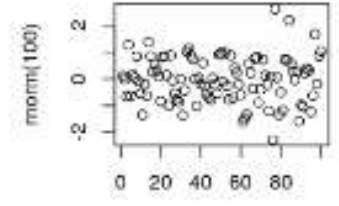
Learning SQL

PUBLISHED
April 10, 2023

Overview

I am going to be learning SQL!

- ▶ Code
- ▶ Code



Components of markdown: YAML

YAML: YAML Ain't Markup Language

The YAML component specifies the metadata of the file:

- Type of output to produce
- Formatting preferences of things like tables
- Other metadata such as document title, author, and date.

YAML is dependent on indentation so be careful:

```
---
title: "Learning Polars"
title-block-banner: true
date: today
format:
  html:
    code-fold: true
    toc: true
    theme:
      light: flatly
      dark: darkly
execute:
  echo: true
  eval: false
---
```


Components of markdown: Code Chunks

Code Chunks are the sections of the document where you will write your code that you wish to include into your document.

For now, we will only use the code chunks as a documentation tool for any code that we write. Later on in the course we will actually be executing the code to produce tables and plots in a document!

Each chunk is opened with a line that starts with three back-ticks, and curly brackets that contain parameters for the chunk (`{ }`). The chunk ends with three more back-ticks.

😊 use shortcut (`CTRL + ALT + i`) to open chunk

```
Run Cell
```{r}
#| eval: true
#| fig.height: 3
#| fig.width: 3

plot(rnorm(100))
```
```

Components of markdown: Code Chunks

What do we mean by parameters in the `{ }` brackets? Lets start with the programming language specification.

- They start with `r` to indicate that the language name within the chunk is `R` (we can also do `python` or `sql` etc.)
- After the `r` you can optionally write a chunk "name" - good practice for debugging later on

The chunk can include other options too, written as `tag:value`, such as:

- `eval: false` to not run the R code.
- `echo: false` to not print the chunk's `R` source code in the output document.
- `warning: false` to not print warnings produced by code.
- `message: false` to not print any messages produced by code.
- `include: true/false` whether to include chunk outputs (e.g. plots) in the document.
- `out.width` and `out.height` provide in style `out.width: "75%"`.
- `fig.align: "center"` adjust how a figure is aligned across the page.
- `fig.show: 'hold'` if your chunk prints multiple figures and you want them printed next to each other (pair with `out.width: c("33%", "67%")`). Can also set `animate` to concatenate multiple into an animation.

Components of markdown: Markdown Text

Markdown Text is what makes using it as a lab-book (and writing journal articles) so versatile.

📖 Would you believe that these slides were all made in using `Rmarkdown`?

So lets start with some basics: *Headings* and *Formatting*

```
# Header 1
```

```
## Header 2
```

```
### Header 3
```

So *how* would `this` text `look`?

```
So how would this text `look`?
```

Components of markdown: Markdown Text

Unordered list items start with `*`, `-`, or `+`, and you can nest one list within another list by indenting the sub-list:

```
- Fruits
- Vegetables
  * Carrot
  * Spinach
```

- Fruits
- Vegetables
 - Carrot
 - Spinach

```
1. Dog
  - German Shepherd #(two spaces)
  - Belgian Shepherd #(two spaces)
2. Cat
  - Siberian #(two spaces)
  - Siamese #(two spaces)
```

1. Dog
 - German Shepherd #(two spaces)
 - Belgian Shepherd #(two spaces)
2. Cat
 - Siberian #(two spaces)
 - Siamese #(two spaces)

Your turn!

Can you produce the following document?

Learning SQL & Markdown

PUBLISHED
April 10, 2023

About me

My name is *Hanjo Odendaal* and I am a **Principal** Data Scientist at [71point4](#)

My favourite food is:

- Steak and Salad

Coding languages

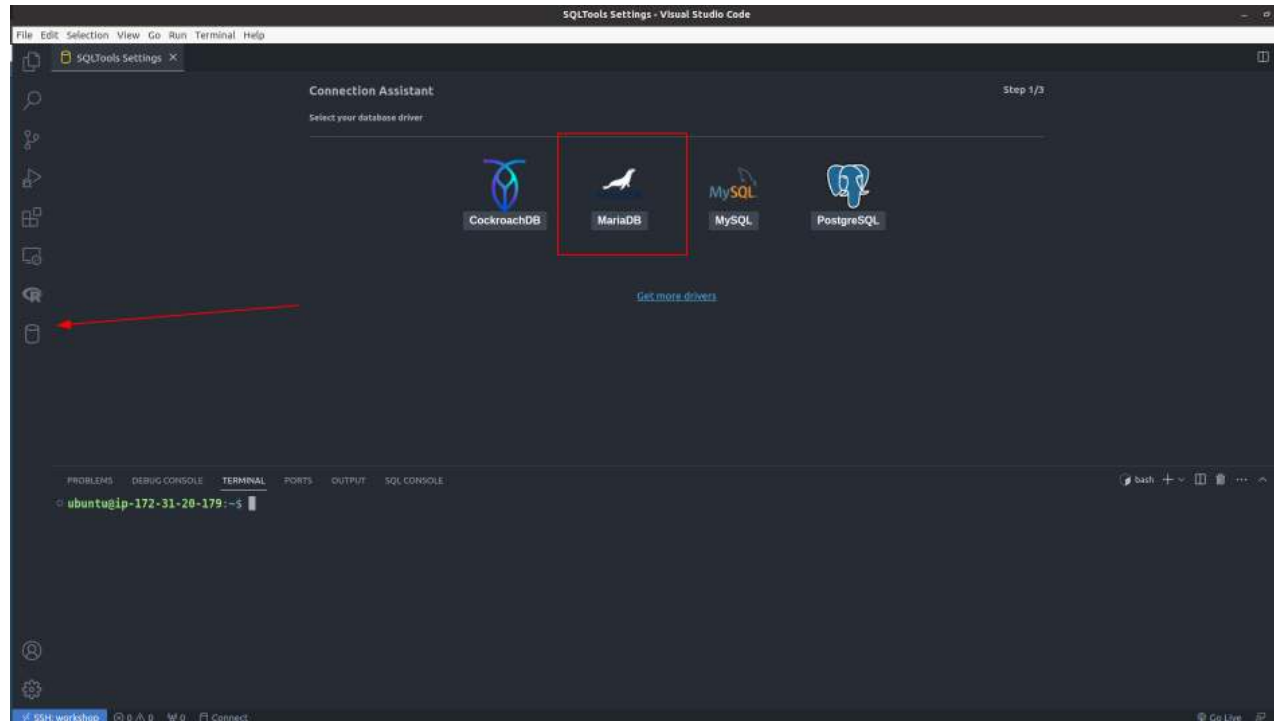
I code in

- `R`, `SQL` and `python`

10:00

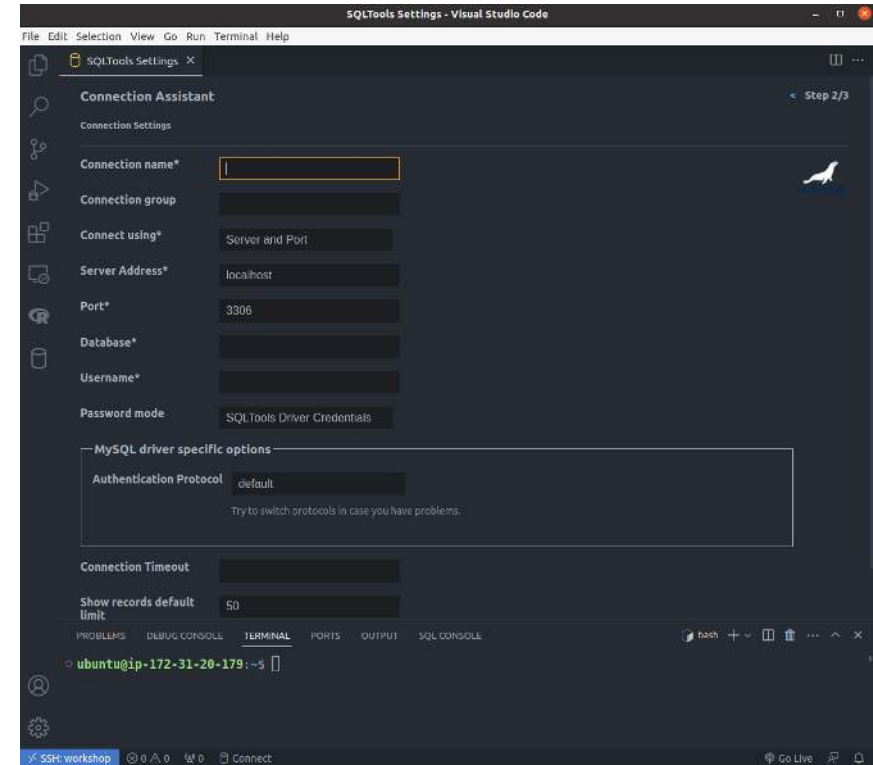
Connection from VScode

Start your notebook for this section

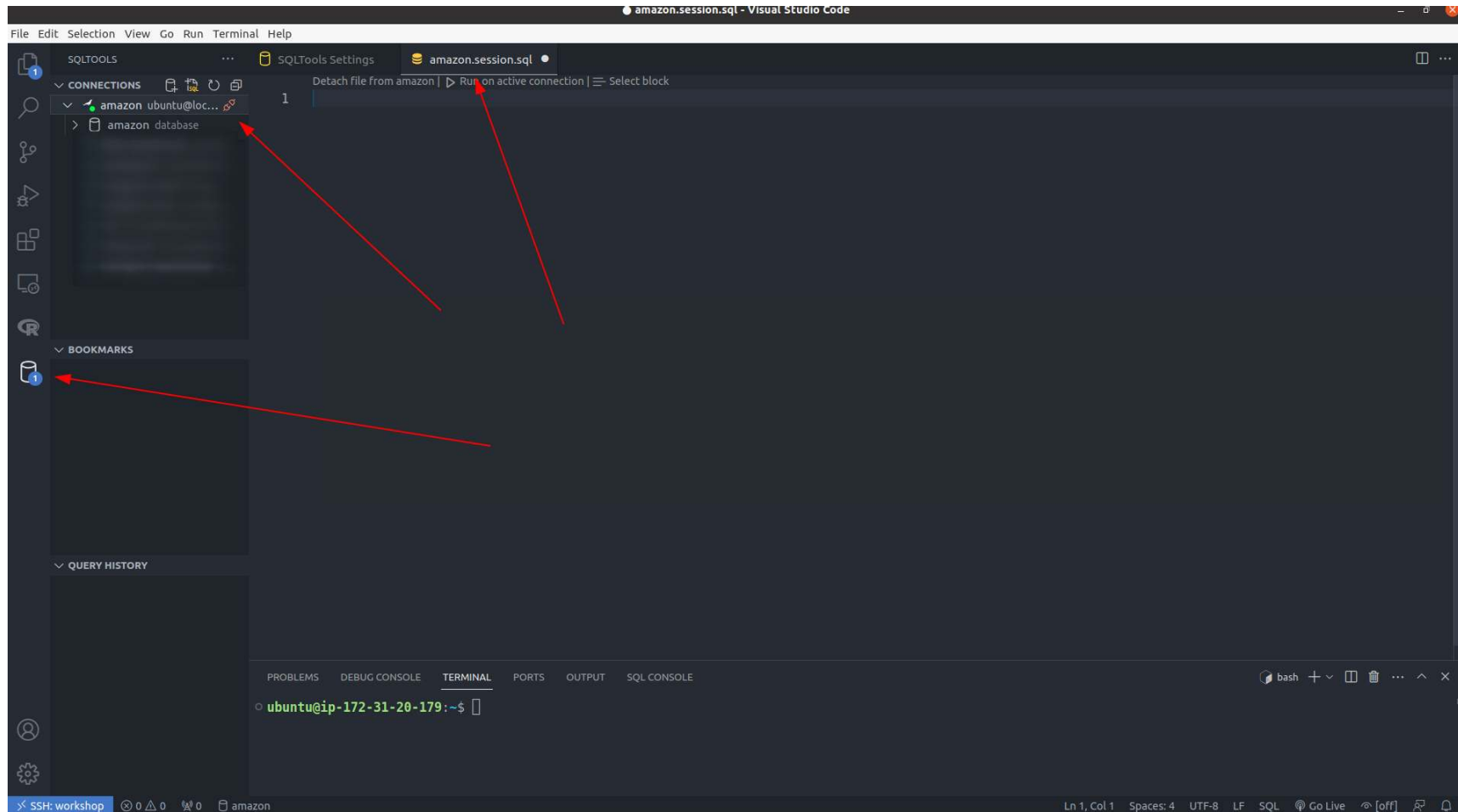


Fill in the information

- port: 3306
- database: amazon
- user: ubuntu
- 0c32348ad0361269b



Connect to database





Polars



Why Polars?

Pandas is definitely the most used data wrangling library in Python, but recently there is a new kid on the block: Polars

Its built on Rust and has some amazing features that I believe will make it the *de facto* library in years to come. It relies on the Apache Arrow Memory Model and Wes McKinney, the creator of Pandas, is heavily involved in Arrow.

<https://pola-rs.github.io/polars/py-polars/html/reference/index.html>

Expression API:

- Allows for manipulation on selection
- Parallel manipulation if multiple columns being mutated
- Query optimization for `lazy`



Why Polars?

I really like the fact that `python` and `R` is starting to play nice and taking the best from each language. I never learned pandas because it reminded me of `base R` which I found very difficult as an economist when I started out. But oh, how things have changed!!

Can you tell me what the following code does?

```
(  
  df.select(  
    [  
      pl.col("Col1"),  
      pl.col("Col2").str.to_lowercase(),  
      pl.col("Col3").round()  
    ]  
  )  
)
```

Data Types & Apache Arrow

We mentioned Arrow earlier. Arrow is an Apache project where they look to best represent tabular data in memory.

- A specification for how data should be represented in memory (Rust)
- A set of libraries in different languages that implement this specification (R AND python)
- Sharing data without copying
- Fast vectorized calculations
- Consistent representation of missing data

Using environments

Before we kick off with some analysis. Lets create an environment. What is it you ask? Imagine you have some of puzzles that you like playing with:

- A virtual environment is like a separate box for each puzzle you're working on.
- Inside this box, you put all the pieces (packages) you need for that puzzle.
- This keeps everything organized and prevents mix-ups between different puzzles.
- You can switch between these boxes (or 'environments') depending on the puzzle you're working on.

Its especially nice to ensure you dont mix up namespaces and its a MUST when developing your own packages. More on this later the week.

Using environments

Now that you now basic Linux, its easy to create a `venv`:

- Step 1: Create the necessary folders:

```
mkdir ~/.virtualenvs/  
cd ~/.virtualenvs/  
sudo apt install python3.10-venv
```

Step 2: Create the environment and activate it.

```
python3 -m venv polars  
source ~/.virtualenvs/polars/bin/activate
```

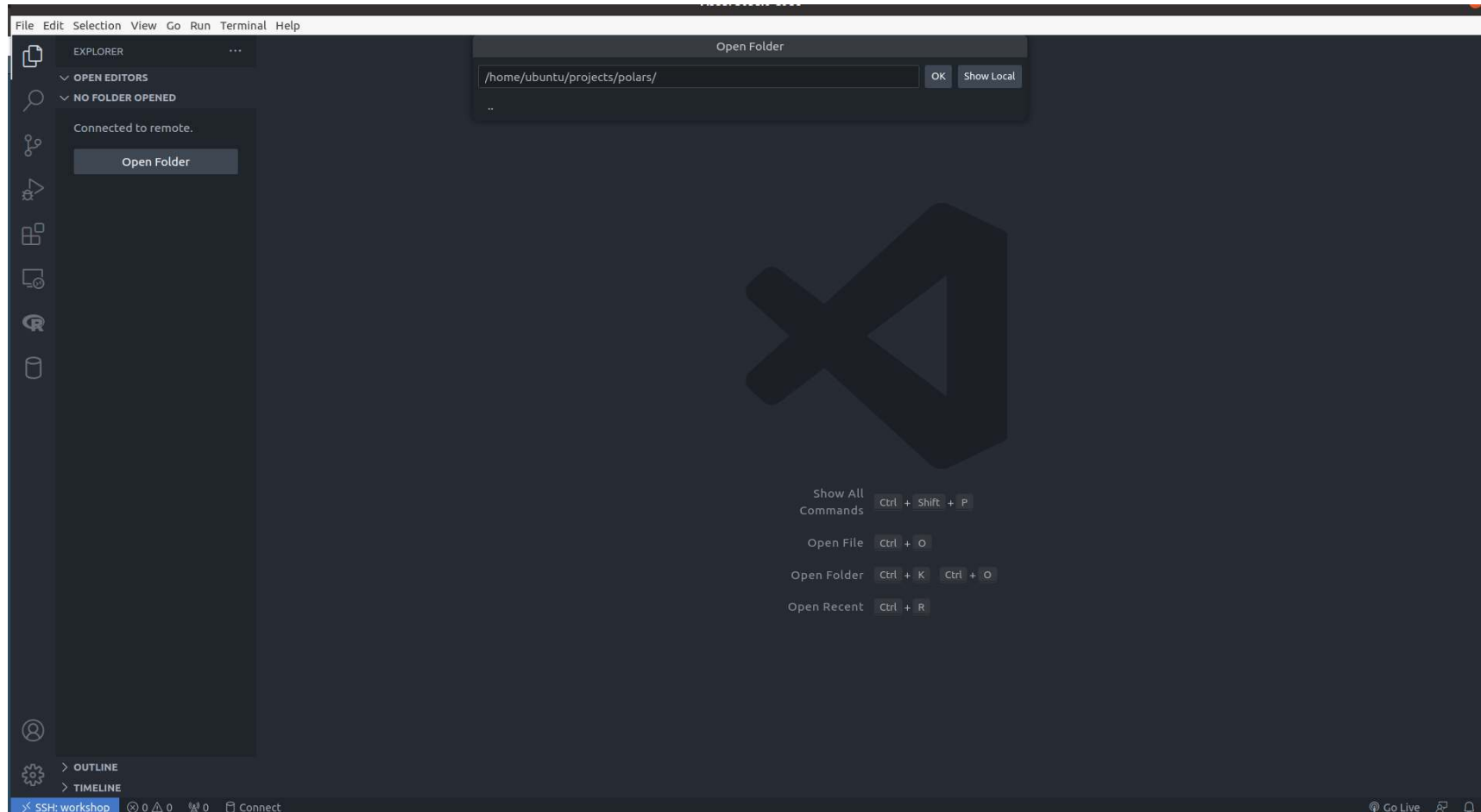
Using venv in VSCode

- add to `vim ~/.bashrc`

```
function py_activate() {  
  source ~/.virtualenvs/$1/bin/activate  
}
```

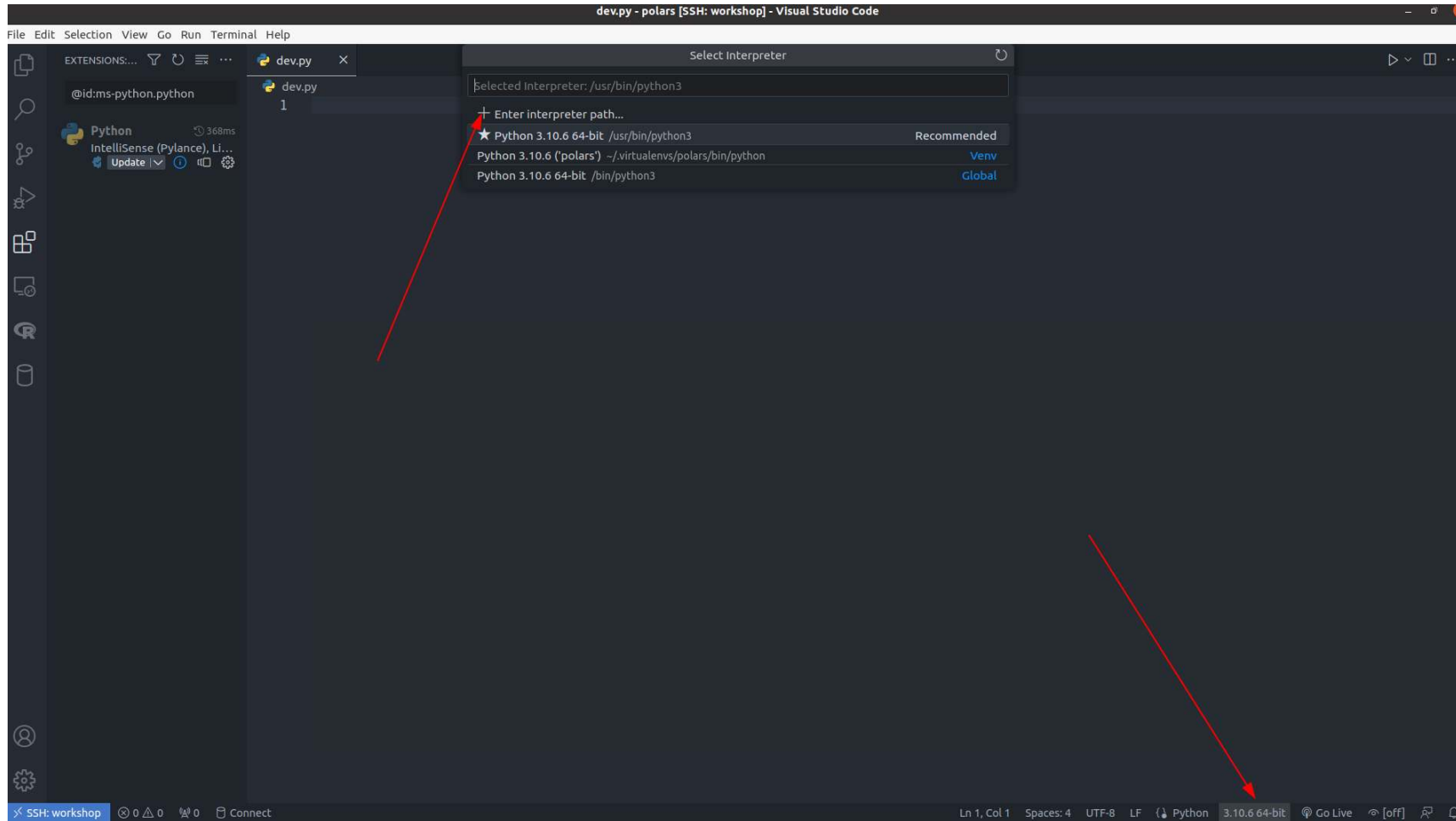

Using venv in VSCode

Open project folder:



Using venv in VSCode

Create `dev.py` and set python interpreter to `~/virtualenvs/polars/bin/python3`



Basics of polars

⚠ Remember to always set your python interpreter OR use workspaces in VSCode!

```
pip install polars
```

```
(polars) ubuntu@ip-172-31-20-179:~/projects/polars$ pip install polars
Collecting polars
  Downloading polars-0.18.0-cp37-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (18.3 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 18.3/18.3 MB 47.7 MB/s eta 0:00:00
Installing collected packages: polars
Successfully installed polars-0.18.0
(polars) ubuntu@ip-172-31-20-179:~/projects/polars$
```

Basics of polars

We are going to start with basic things and load the `~/worldcup.csv` dataset found on the server:

- In `dev.py`

```
import polars as pl

csvfile = '~/data/worldcup.csv'
df = pl.read_csv(csvfile)

df.head()
df.glimpse()
```

Basics of polars

We are going to start with basic things and load the `~/worldcup.csv` dataset found on the server:

- In `dev.py`

```
df.head()
# >>> df.head()
# shape: (5, 10)
#
```

| year | host | winner | second | ... | goals_scored | teams | games | attendance |
|------|-------------|--------------|----------------|-----|--------------|-------|-------|------------|
| --- | --- | --- | --- | | --- | --- | --- | --- |
| i64 | str | str | str | | i64 | i64 | i64 | i64 |
| 1930 | Uruguay | Uruguay | Argentina | ... | 70 | 13 | 18 | 434000 |
| 1934 | Italy | Italy | Czechoslovakia | ... | 70 | 16 | 17 | 395000 |
| 1938 | France | Italy | Hungary | ... | 84 | 15 | 18 | 483000 |
| 1950 | Brazil | Uruguay | Brazil | ... | 88 | 13 | 22 | 1337000 |
| 1954 | Switzerland | West Germany | Hungary | ... | 140 | 16 | 26 | 943000 |

```
#
```

Basics of polars

We are going to start with basic things and load the `~/worldcup.csv` dataset found on the server:

- In `dev.py`

```
df.glimpse()

# >>> df.glimpse()
# Rows: 21
# Columns: 10
# $ year      <i64> 1930, 1934, 1938, 1950, 1954, 1958, 1962, 1966, 1970, 1974
# $ host      <str> Uruguay, Italy, France, Brazil, Switzerland, Sweden, Chile, England, Mexico, German
# $ winner    <str> Uruguay, Italy, Italy, Uruguay, West Germany, Brazil, Brazil, England, Brazil, West
# $ second    <str> Argentina, Czechoslovakia, Hungary, Brazil, Hungary, Sweden, Czechoslovakia, West G
# $ third     <str> USA, Germany, Brazil, Sweden, Austria, France, Chile, Portugal, West Germany, Polan
# $ fourth    <str> Yugoslavia, Austria, Sweden, Spain, Uruguay, West Germany, Yugoslavia, Soviet Union
# $ goals_scored <i64> 70, 70, 84, 88, 140, 126, 89, 89, 95, 97
# $ teams     <i64> 13, 16, 15, 13, 16, 16, 16, 16, 16, 16
# $ games     <i64> 18, 17, 18, 22, 26, 35, 32, 32, 32, 38
# $ attendance <i64> 434000, 395000, 483000, 1337000, 943000, 868000, 776000, 1614677, 1673975, 1774022
```

Nice configs for polars

```
pl.Config.set_tbl_rows(100)  
pl.Config.set_tbl_cols(100)
```

```
dir(pl.Config)  
#[ '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__',
```

Data Types

Lets what data types and schemas the data contains:

```
df.schema # better
df.dtypes
df["Name"].dtype

# >>> df.schema # better
# {'year': Int64, 'host': Utf8, 'winner': Utf8, 'second': Utf8, 'third': Utf8, 'fourth': Utf8, 'goals_scor
# >>> df.dtypes
# [Int64, Utf8, Utf8, Utf8, Utf8, Utf8, Int64, Int64, Int64, Int64]
# >>> df["goals_scored"].dtype
# Int64
```


Data Types

There are multiple types of data types in `polars`, its nice to know some of the basic types because once you move into 'high' performance analytics. These matter a lot. But for now, we gonna use them in selecting columns 😊

- `polars.DataType`
- `polars.Decimal`
- `polars.Float32`
- `polars.Float64`
- `polars.Int8`
- `polars.Int16`
- `polars.Int32`
- `polars.Int64`
- `polars.UInt8`
- `polars.UInt16`
- `polars.UInt32`
- `polars.UInt64`
- `polars.Date`
- `polars.Datetime`
- `polars.Duration`
- `polars.Time`
- `polars.Array`
- `polars.List`
- `polars.Struct`
- `polars.Boolean`
- `polars.Binary`
- `polars.Categorical`
- `polars.Null`
- `polars.Object`
- `polars.Utf8`
- `polars.Unknown`

Selecting Columns

To select columns, we wrap the selection in `pl.col` using the `.select()` method on a data frame:

```
(
  df.select(
    [
      pl.col("host"),
      pl.col("year")
    ]
  )
)
```

We can also apply some transformation in the `select`:

```
(
  df.select(
    [
      pl.col("host").str.to_lowercase(),
      pl.col("year").alias('year_game_played')
    ]
  )
)
```

Your turn!

Select the winner and goals scored columns for me!

Then

Select attendance and rename to spectators

10:00

Putting your first script into production

Start by creating a new folder:

- `~/projects/production/python`

Then create a python file: `say_hello.py`

```
#!/usr/bin/python3

import datetime

now = datetime.datetime.now()

if __name__ == "__main__":
    print(f'[{now}] - Hello, World!')
```

- The `shebang` is a special kind of comment that you may include in your source code to tell the operating system's shell where to find the interpreter for the rest of the file. (especially useful when using environments)

It's not uncommon to combine a shebang with the `name-main` idiom:

- Prevents the main block of code from running when someone imports the file from another module

Last step, make the file executable: `chmod +x say_hello.py`

Learning basics of VIM

Why learn VIM?



There is an old joke about a visitor to New York City asking a passerby for directions to the city's famous classical music venue.

Visitor: Excuse me, how do I get to Carnegie Hall?

Passerby: Practice, practice, practice!

Learning the Linux command line, like becoming an accomplished pianist, is not something that we pick up in an afternoon. It takes years of practice. In this chapter, we will introduce the `vi` (pronounced “vee eye”) text editor, one of the core programs in the Unix tradition. `vi` is somewhat notorious for its difficult user interface, but when we see a master sit down at the keyboard and begin to “play,” we will indeed be witness to some great art. We won’t become masters in this chapter, but when we are done, we will know how to play the equivalent of “Chopsticks” in `vi`.

Linux Command Line, 2nd Edition - Jr. William E. Shotts

Why learn VIM?

So why learn VIM when you have something like `VSCode` or `Rstudio` as an IDE?

- VIM is ubiquitous on all Unix systems, which mean you will always have access to an editor, even if your front-end crashes.
- VIM is powerful and fast. Sometimes you just need to change a simple config file and VIM works best for these times. Also, you may not have GUI access when doing routine SysAdmin tasks as root.
- Once you have mastered VIM¹, then there are few way to be more efficient in typing up code or changing files since you never need a mouse.
- We don't want other Linux and Unix users to think we are cowards.²

¹ No one on earth can say that they have mastered VIM.

² Its a joke from *Linux Command Line, 2nd Edition*, but still true 😊

First things first, how to exit

Stackoverflow, helping people exit vim since 2011.



The screenshot shows a Stack Overflow question titled "How do I exit Vim?". The question is 10 years, 8 months old, modified 26 days ago, and has been viewed 2.8 million times. The question text is "I am stuck and cannot escape. It says: type :quit<Enter> to quit VIM". The answer text is "But when I type that it simply appears in the object body." The question is tagged with "vim" and "vi". The question was asked by jclancy on Aug 6, 2012 at 12:25. The answer was edited by UnrealApex on Jun 4, 2022 at 11:47.

To exit we enter the *editor* with `:`, then type `q` and a `!` (the exclamation, `!`, means to force close):

```
:q!
```


Basics of editing a file

⚠ Follow my commands before typing. Do not type anything yet!

Remember, if something bad happens just press ESC a couple of times and then exit VIM with `:q!`

```
hanjo@optimus0:~$ vim owner_information.txt
```

- In VIM, every keystroke is a specific command, this type of editor is known as a *modal editor*.
 - VIM starts by going into *command mode*, which means it expects commands, NOT input text.

To type something we must go to *Insert Mode*. To do this, type `i`. You should see the following at the bottom:

```
-- INSERT --
```

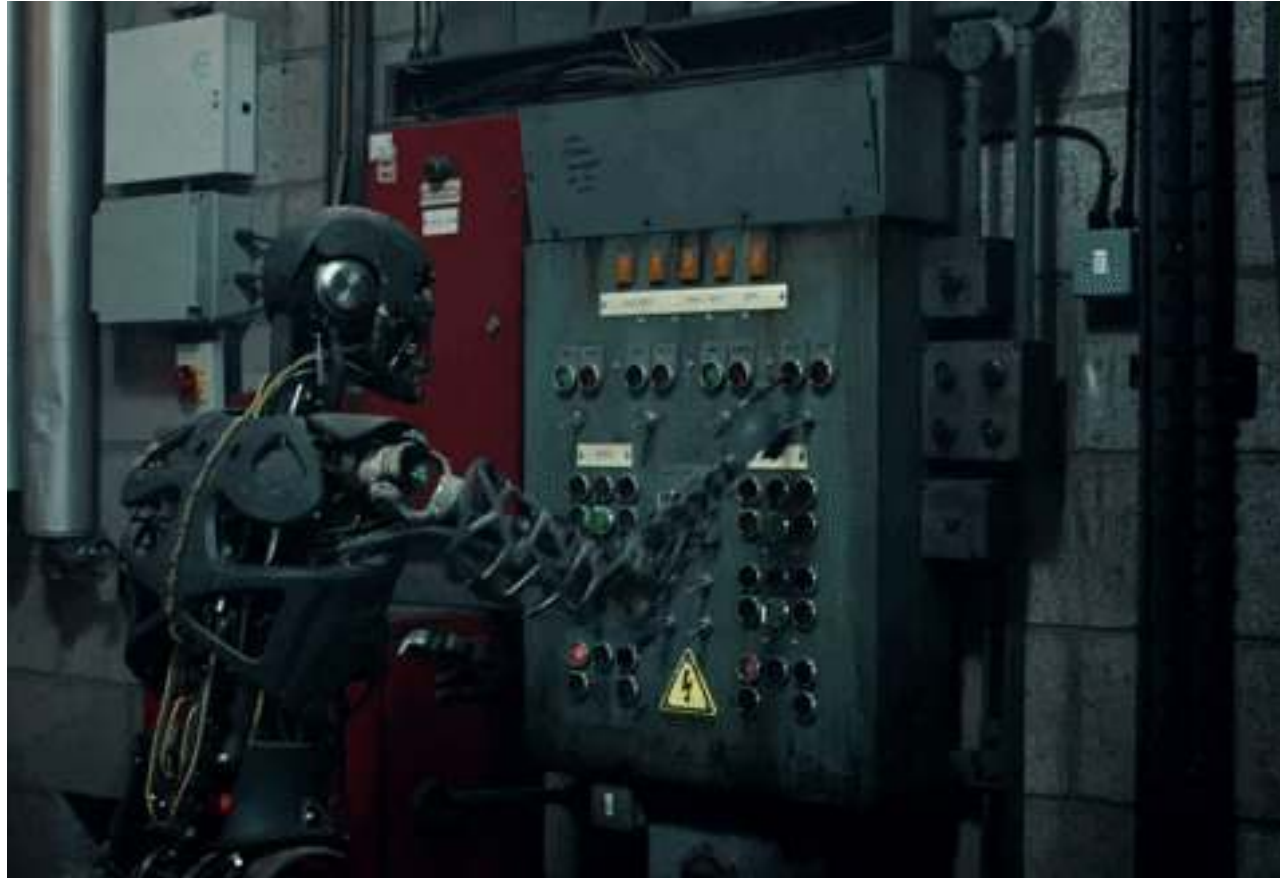
Now, type the following:

```
[owner] Hanjo Odendaal
```

Save and exit by pressing `ESC` and `:wq`

Putting your first script into production

The time has come!



Cronjobs and crontabs

What is cron?

The cron command-line utility, also known as cron job is a job scheduler on Unix-like operating systems.

Lets open crontab:

```
hanjo@optimus0:~$ crontab -e
```

```
# For details see man 4 crontabs  
  
# Example of job definition:  
# .----- minute (0 - 59)  
# | .----- hour (0 - 23)  
# | | .----- day of month (1 - 31)  
# | | | .----- month (1 - 12) OR jan,feb,mar,apr ...  
# | | | | .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat  
# | | | | |  
# * * * * * user-name command to be executed
```

Cronjobs and crontabs

Lets say that the script must run every minute and output to a file in a folder called `logs`:

```
# For example, you can run a backup of all your user accounts  
# at 5 a.m every week with:  
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/  
#  
# For more information see the manual pages of crontab(5) and cron(8)  
#  
# m h dom mon dow  command  
  
* * * * ~/projects/production/python/say_hello.py >> ~/logs/python_logs.log 2>&1
```

The end



Production

Implement the following script in production. I want this to be in our production folder, call it `football_watcher.py`. This has to run every MONDAY morning at 08:00 for me.

💡 Go to <https://crontab.guru/>

```
import polars as pl
csvfile = '~/data/worldcup.csv'

df = pl.read_csv(csvfile)
df.head()
```

30:00



Learning Polars with Python

Section 2

Selection expression

What you just used was called an 'expression'. Its using `verbs` to *express* what you need from code in English.

You can also do in the more 'classic' way, but its not recommended. When we get to `LazyFrames` and `chaining` you will see why:

```
df[range(1, 5), "winner"]  
df.select(pl.col('winner'))
```


Selection expression

Lets see some more examples:

```
(  
  df  
  .select('winner')  
  .to_series()  
  .head(3)  
)  
  
(  
  df  
  .select(['winner', 'second'])  
  .head(3)  
  .to_series()  
)
```

Being smarter on your select

Polars has some really nice smart selector helpers. Lets explore some.

- `.all` + `.exclude`

```
(
  df.select(
    pl.all()
  )
)

(
  df.select(
    pl.all().exclude(["games", "spectators"])
  )
)
```

Being smarter on your select

Polars has some really nice smart selector helpers. Lets explore some.

- Regex (for the brave 🐉)
- Has to have "^something\$"

```
(  
  df.select(  
    pl.col("^goals.*$")  
  )  
)  
  
(  
  df.select(  
    pl.col("^goals.*$").max()  
  )  
)
```

Being smarter on your select

Polars has some really nice smart selector helpers. Lets explore some.

- Select on type

```
(  
  df.select(  
    pl.col(pl.Int64)  
  )  
  .head(3)  
)
```

Your turn!

Select all besides the the team that came third

Only select string columns for me and transform to lowercase

Select attendance and tell me what minimum and median and mean attendance was (Tip: Google is your friend)

25:00

Rename

Often times the column names are not easy to work with. When this happens, we need to rename. This can easily be done by using the simple `.rename` method:

```
(
  df
  .rename({"PassengerId": "ID"})
  .head(2)
)
```

Can also mass a list:

```
(
  df
  .rename(
    {
      "second": "runner_up",
      "attendance": "people",
    }
  )
  .head(2)
)
```

Drop

Just like `.rename`, you can use `.drop` to get rid of unnecessary columns

```
(  
  df  
  .drop(  
    [  
      "fourth",  
      "host"  
    ]  
  )  
  .head(2)  
)
```

Missing Values

In Pandas a missing value can be represented with a `null`, `NaN` or `None` value depending on the `dtype` of the column. Polars also allows `NaN` values for floating point columns as we will see.

```
df = pl.DataFrame(  
    {  
        'col1': [0, None, 2],  
        "col2": [None, None, 5]  
    }  
)  
df
```

Polars stores a count of how many `null` values there are. We can access this with the `null_count` method on a single column or on all the columns

```
df.null_count()
```


Missing Values

We use the `is_null` expression to find out whether each value is `null` and `is_not_null` for the opposite. In the following section you will see how we can use it in a filtering expression.

```
(
  df
  .select(
    [
      pl.col("col1"),
      pl.col("col1").is_null().alias("is_null"),
      pl.col("col1").is_not_null().alias("is_not_null")
    ]
  )
)
```

Filtering

Selecting multiple rows using `list`, `slice`, `range`, but NOT boolean! Lets start with the basics again. Read in the `worldcup` data set.

- `list`
 - We can pass a list of integers `[]`

```
df[[1, 3]]
```

- `slice`
 - we can use slice notation

```
df[:3]
```

Filtering

Selecting multiple rows using `list`, `slice`, `range`, but NOT boolean! Lets start with the basics again. Read in the `worldcup` data set.

- `range`
 - range of integers

```
df[range(1, 5)]
```

- Boolean list not accepted!

```
df[df["Age"] > 30]
```

Filtering

Although we can use `list`, `slice` and `range`, its *much* easier to use the expression API.

```
csvfile = '~/data/worldcup.csv'  
df = pl.read_csv(csvfile)  
(  
    df  
    .filter(  
        pl.col("winner") == 'France'  
    )  
)
```

Filtering

Replaces `.loc` from `pandas` we can use `.with_row_count` method.

```
(
  df
  .with_row_count( name = "row_nr")
  .filter(
    pl.col("row_nr") > 10
  )
)

(
  df
  .with_row_count( name = 'row_nr')
  .filter(
    pl.col("row_nr").is_between(4, 10)
  )
)
```

What if we want a sample of the data set to work with:

```
(
  df.sample(n = 10)
)
```

Filtering

- Winner is France and year is 2018?

```
(  
  df  
  .filter(pl.col('winner') = 'France')  
  .filter(pl.col('year') = 2018)  
)  
.glimpse()
```

BUT

```
(  
  df  
  .filter(  
    (pl.col('winner') = 'France') &  
    (pl.col('year') = 2018)  
  )  
)  
.glimpse()
```

```
(  
  df  
  .filter(  
    (pl.col('winner') = 'France') |
```

Filtering

Filter where the teams are greater than 18

We can also be creative. How do we filter where the attendance is larger than the mean?

25:00

Sorting

```
df.sort("attendance")
```

```
df.sort(["attendance", "games"], descending = True, nulls_last = True)
```


Mutating columns

What do we mean by 'mutating columns'? These are row operations and can be that we want to add columns or perhaps change them in some manner.

```
df.with_columns(  
  (pl.col('goals_scored')/ pl.col('games')).alias('average_goals')  
)
```

Lets change the format:

```
(  
  df.with_columns(  
    (pl.col('goals_scored')/ pl.col('games'))  
    .round(2)  
    .alias('average_goals')  
  )  
)
```

Mutating columns

We can also add a constant to the data frame:

```
(
  df.with_columns(
    (pl.col('goals_scored')/ pl.col('games'))
    .round(2)
    .alias('average_goals')
  ).with_columns(
    pl.lit('football').alias('type')
  )
)
```

```
(
  df.with_columns(
    [
      (pl.col('goals_scored')/ pl.col('games'))
      .round(2)
      .alias('average_goals'),
      pl.lit('football').alias('type')
    ]
  )
)
```

Mutate on type

One of the best ways to mutate multiple columns that might need cleaning is using the `dtype`.

```
(
  df
  .with_columns(
    pl.col(pl.Utf8).str.to_uppercase()
  )
  .select(
    pl.col(pl.Utf8)
  )
  .head(2)
)
```

```
(
  df
  .with_columns(
    (pl.col('goals_scored') / pl.col('games'))
    .alias('average_goals'),
  )
  .with_columns(
    pl.col(pl.Float64).cast(pl.Int64)
  )
  .head(2)
)
```

Mutating columns

```
(
  df
  .with_columns(
    (
      pl.col('goals_scored')/pl.col('games')
    )
    .round(2)
    .alias('average_goals')
  )
  .sort('average_goals', descending = True)
  .select(
    ['year', 'host', 'winner', 'average_goals', 'games', 'goals_scored']
  ).head(3)
)
```

Mutating columns

Another cool feature is to mutate row-wise across multiple columns

```
(  
  df  
  .with_columns(  
    pl.max(  
      [  
        pl.col('teams'), pl.col('games')  
      ]  
    )  
    .alias('max_games_teams')  
  )  
)
```

Mutating frames

Options to join frames:

- 'vertical', 'diagonal', 'horizontal', 'align'

```
seriesA = (  
    df.with_columns(  
        (pl.col('goals_scored') / pl.col('games'))  
        .round(2)  
        .alias('average_goals')  
    ).with_columns(  
        pl.lit('seriesA').alias('type')  
    )  
)
```

```
seriesB = (  
    df.with_columns(  
        (pl.col('goals_scored') / pl.col('games'))  
        .round(2)  
        .alias('average_goals')  
    ).with_columns(  
        pl.lit('seriesA').alias('type')  
    )  
)
```

```
pl.concat([seriesA, seriesB], how="vertical")  
pl.concat([seriesA, seriesB], how="diagonal")
```

Lets play with a bit with Amazon reviews!

Lets play with a bit with Amazon reviews

```
import polars as pl

csvfile = '~/data/amazon/amazon_reviews_us_Watches_v1_00.tsv'
df = pl.read_csv(csvfile)
```



😎 What do you see?

Group By

Lets play with a bit with Amazon reviews

```
import polars as pl

csvfile = '~/data/amazon/amazon_reviews_us_Watches_v1_00.tsv'
df = pl.read_csv(csvfile, separator = '\t', ignore_errors = True)

df.glimpse()
```


Group By

Time to play with aggregations! Think of it as Pivot Tables. So, `mean` by, or `count` by or `n_unique` in a number of groups. The methods we can all on `GroupBy` in mode are:

- `first` get the first element of each group
 - `last` get the last element of each group
 - `n_unique` get the number of unique elements in each group
 - `count` get the number of elements in each group
 - `sum` sum the elements in each group
 - `min` get the smallest element in each group
 - `max` get the largest element in each group
 - `mean` get the average of elements in each group
 - `median` get the median in each group
 - `quantile` calculate quantiles in each group

```
(  
    df  
    .groupby("verified_purchase")  
)  
# <polars.dataframe.groupby.GroupBy object at 0x7f54a
```

Group By

```
(  
    df  
    .groupby("host")  
    .count()  
)  
  
(  
    df  
    .groupby("attendance")  
    .mean()  
)  
  
(  
    df  
    .groupby("winner")  
    .n_unique()  
)
```

Group By

- We can also use our nice column selectors to pull out the correct columns to remove the `Null` columns

```
(
  df
  .groupby('verified_purchase')
  .mean()
  .select(
    [
      pl.col('verified_purchase'),
      pl.col(pl.Float64)
    ]
  )
)
```

Group By

- We can pass a `list` to `.agg` to set out different aggregations.

```
(
  df
  .groupby('verified_purchase')
  .agg(
    [
      pl.col('star_rating').mean(),
      pl.col('total_votes').max()
    ]
  )
)
```

Group By

- What if we want to do aggregations across columns?

```
(  
  df  
  .groupby('verified_purchase')  
  .agg(  
    [  
      pl.col(colName).max()  
      for colName in  
      ['helpful_votes', 'total_votes']  
    ]  
  )  
)
```

Analyzing Amazon watches

- Lets analyse Rolex watches!

💡 We can use `filter` and `contains` to look for Rolex in the `product_title`:

```
(  
  df  
  .filter(  
    pl.col('product_title')  
    .str.contains('Rolex')  
  )  
)
```

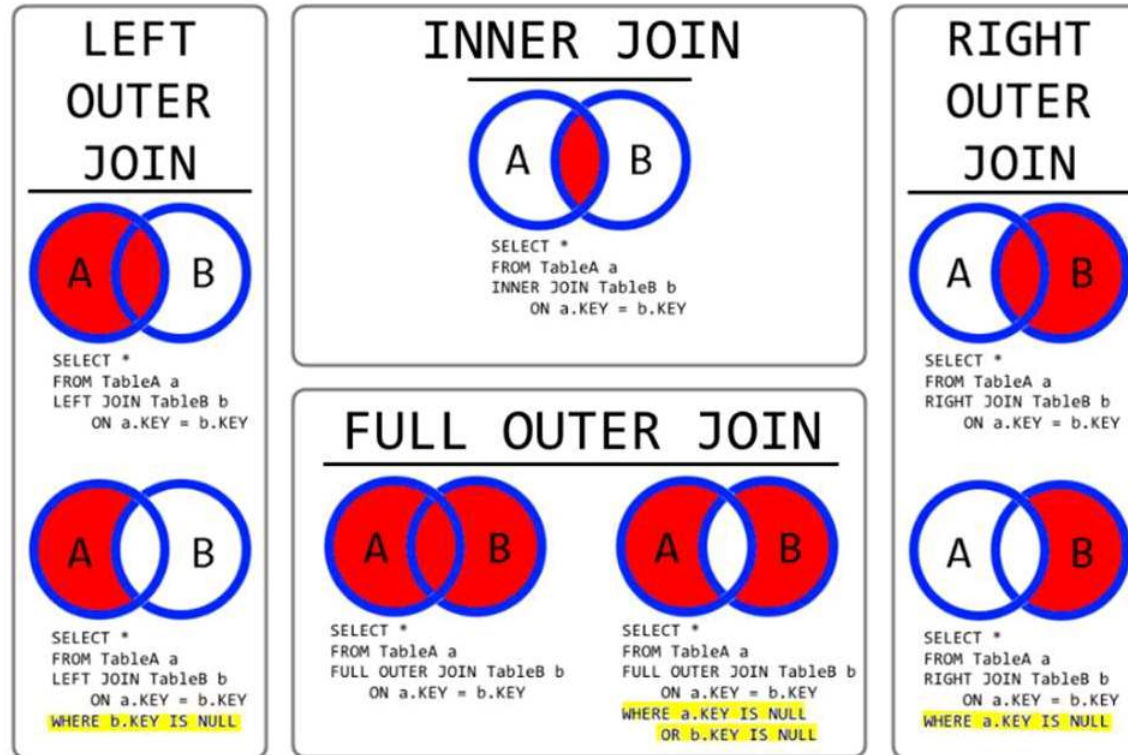
```
(  
  df  
  .with_columns(  
    pl.when(  
      pl.col('product_title')  
      .str  
      .contains('Rolex')  
    )  
    .then(1).otherwise(0)  
    .alias('Rolex')  
  )  
)
```

Analyzing Amazon watches

- What percentage of reviews were helpful?
- Are the Rolex `star_ratings` higher than the rest of the watches?
- Is there a person who has made more than one Rolex review?

45:00

SQL JOINS



Joins

To understand joins, let's start by breaking up amazon into two data sets and then put them back together.

- Create `review_ratings` and `review_body`
- Let's now put it back together using `review_id`!

```
review_ratings = (  
  df  
  .select(  
    [  
      pl.col('review_id'),  
      pl.col('star_rating')  
    ]  
  )  
)
```

```
review_body = (  
  df  
  .select(  
    [  
      pl.col('review_id'),  
      pl.col('review_body')  
    ]  
  )  
)
```

```
review_ratings.join(review_body, on = "review_id", how = "left")
```

Joins

Create two data frames that contain

- DF One: `reviews_id` & `star_ratings` and sample 1000 rows
- DF Two: `reviews_id` & `total_votes`

Then get the `median` for the `star_ratings` for these samples.

💡 You need to use `inner` join.

45:00

Iteration

We can iterate over a single column just as we would do with a Pandas column or a Numpy array. Its nice for pulling out as array.

```
[year* 2 for year in df["year"]]  
[(row[0],row[1]) for row in df.rows()]
```

The `iter_row` method, as the name suggests, allows you to iterate over the rows in the DataFrame. This method returns a generator which you can use in a loop to access each row one by one. This can be useful if you need to process each row in sequence, or if you need to process each row individually and the dataset is too large to fit into memory all at once.

```
[row['host'] for row in df.iter_rows(named=True)]
```

The `rows` method, on the other hand, would allow you to access a particular row directly by index, but this is generally a less common operation in DataFrame-style processing, as operations are usually vectorized (i.e., performed on entire columns at once).

```
[row['host'] for row in df.rows(named=True)]
```

Function & Iteration

Lets see how this iteration could be used in a function:

- Lets about it in a string

```
def printer(row):  
    """  
    Function prints the game  
    """  
  
    res = f"""The host was {row['host']} and winner was {row['winner']}"""  
  
    print(res)  
    return  
  
[printer(row) for row in df.iter_rows(named=True)]
```

Seeing what polars can do: Lazy

They say polars is efficient. So let's start with a small dataset of 1,000,000 rows of amazon reviews. 😊. It can also read from a compressed file!

```
csvfile = "~/data/amazon.tsv.gz"
df = (
    pl.read_csv(csvfile, separator = '\\t', ignore_errors = True)
)
df

df.glimpse()
```

Seeing what polars can do: Lazy

To see what the 'execution plan' is, we can use `.explain` to see what `polars` is going to do.

```
csvfile = "~/data/games.tsv"  
(  
    pl.scan_csv(csvfile, separator = '\\t', ignore_errors = True)  
    .explain()  
)
```

Lazy mode

- Applies optimized query optimization

```
csvfile = "~/data/games.tsv"
df = pl.scan_csv(csvfile, separator = '\\t', ignore_errors = True).fetch(3)
df.glimpse()

(
  pl.scan_csv(csvfile, separator = '\\t', ignore_errors = True)
  .groupby(["star_rating"])
  .agg(pl.col("star_rating").count().alias("counts"))
  .explain
)
```

- Streaming = `True` process in stream (larger than RAM data)

```
(
  pl.scan_csv(csvfile, separator = '\\t', ignore_errors = True)
  .groupby(["star_rating"])
  .agg(pl.col("star_rating").count().alias("counts"))
  .collect(streaming = True)
)
```

Analyzing LARGE data sets

Lets say we have a data set that is around 5GB. This is a very large data set to analyse in memory. Lets now employ lazy mode in `polars`.



Analyzing LARGE data sets

Lets say we have a data set that is around 5GB. This is a very large data set to analyse in memory. Lets now employ lazy mode in `polars`.

- Step 1: How many rows are we talking about?

```
csvfile = "~/data/amazon/*.tsv"
(
    pl.scan_csv(csvfile, separator = '\\t',
                ignore_errors = True)
    .select(
        pl.count()
    )
    .explain()
)
```

```
csvfile = "~/data/amazon/*.tsv"
(
    pl.scan_csv(csvfile, separator = '\\t',
                ignore_errors = True)
    .select(
        pl.count()
    )
    .collect(streaming = True)
)
```

Analyzing LARGE data sets

Lets say we have a data set that is around 5GB. This is a very large data set to analyse in memory. Lets now employ lazy mode in `polars`.

- Step 2: Now we can count the number of observations per category

```
csvfile = "~/data/amazon/*.tsv"
(
    pl.scan_csv(csvfile, separator = '\t',
                ignore_errors = True)
    .groupby(["product_category"])
    .agg(pl.col("product_category")
        .count().alias("counts"))
    .explain()
)
```

```
csvfile = "~/data/amazon/*.tsv"
(
    pl.scan_csv(csvfile, separator = '\t',
                ignore_errors = True)
    .groupby(["product_category"])
    .agg(pl.col("product_category")
        .count().alias("counts"))
    )
.collect(streaming = True)
)
```

Analyzing LARGE data sets

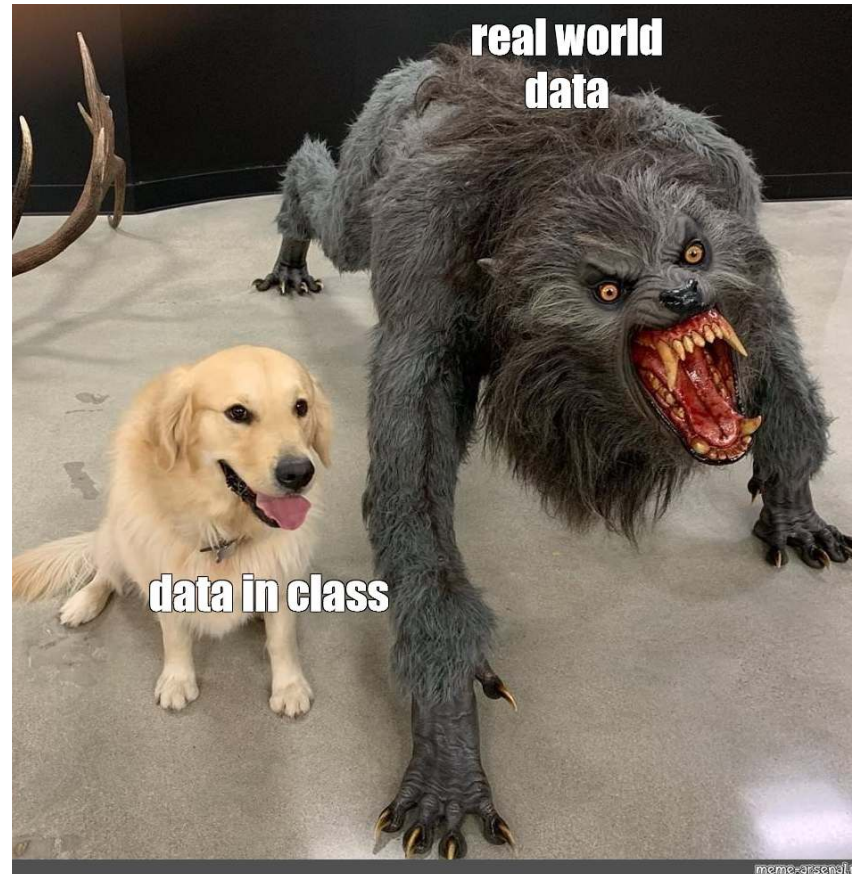
Lets say we have a data set that is around 5GB. This is a very large data set to analyse in memory. Lets now employ lazy mode in `polars`.

- Step 3: Filter and sort the results...
- Step 4: Do the results add up to our initial count?

25:00

Analyzing LARGE data sets

- <https://stackoverflow.com/questions/76391153/python-polars-lazy-frame-row-count-not-equal-wc-l>





Learning Polars with Python

Section 3

Dbutils

Do you have problems connecting to databases? Do you put passwords in plain text in python scripts (Big No no!)?
What if you want to write to BD?

- Well, we have the answer for you!

```
import pathlib #info about where things are stored
from setuptools import setup, find_packages

HERE = pathlib.Path(__file__).parent # anchoring
# print(HERE)

VERSION = '1.2.0'
PACKAGE_NAME = 'dbutils'
AUTHOR = 'JUSTE NYIRIMANA'
AUTHOR_EMAIL = 'justenyirimana@gmail.com'
URL = 'justenyirimana@gmail.com'

LICENSE = 'MIT'
DESCRIPTION = 'DBUTILS is a collection of functi
```



Traditional way of uploading data

In the traditional way of uploading data to a DB, the best way is to (1) copy the CSV to the machine, (2) create the table in the database and (3) then upload the data using the load command.

- Create a DB

```
CREATE DATABASE workshop
```

- Connect to mysql through VSCode and create the table in the database

```
DROP TABLE IF EXISTS property;  
CREATE TABLE property(  
    property_type VARCHAR(255),  
    addresslocality VARCHAR(255),  
    bedrooms INT,  
    bathrooms INT,  
    derived_lcy DOUBLE PRECISION NOT NULL  
);
```

Traditional way of uploading data

In the traditional way of uploading data to a DB, the best way is to (1) copy the CSV to the machine, (2) create the table in the database and (3) then upload the data using the copy command.

- Next upload the data to the database

```
--LOAD DATA INTO DB  
LOAD DATA LOCAL INFILE '/home/ubuntu/data/property/property.csv'  
INTO TABLE workshop.property  
FIELDS TERMINATED BY ','  
;
```


Exercises for SQL

- What is the average and standard deviation of house prices?
- How much more expensive is adding an extra bedroom and going from a 2 to 3 bedroom if I RENT?
- Where are the most expensive houses for sale?

30:00

Now for dbutils!!

What happens if we want to interact with the database through `python`? Well, you can either write your own functions or just use our `dbutils` package! The main class is: `Query`, and it has the following methods:

- `sql_query`
 - Helps to query the database
- `sql_write`
 - Helps to write data from `python` to `mysql` DB
- `sql_execute`
 - Executes raw `sql` commands

Obviously one can extend this package quite a lot, but these are the nice basic functions we use from day to day. Now let's do what we did above, but using `dbutils`

1. Create a folder: `~/projects/analytics/dbutils` and activate your `polars` environment!
2. Download the `.whl` from the website and upload your folder
3. Install the `.whl` using the file: `pip install dbutils.whl` from CLI
4. Install `decouple` so that we don't have plain text password files: `pip install python-decouple`
5. Create a file in the SAME folder called `.env`

Now for dbutils!!

⚠ Its important to never same passwords in plain text on your machine!! Its best to use environment variables for this. In the `.env` file, add the following information:

```
db_port=3306
db_host=localhost
db_user=ubuntu
db_pass=0c32348ad0361269b
```

- `db_port`: Specifies the port your `MySQL` instance is running on: 3306 is the default
- `db_host`: This is the IP address (remember 127.0.0.1?)
- `db_user`: The username
- `db_pass`: The password

The `decouple` package in `python` will then pick up these *environment variables* automatically and can be used as `config('db_port')` within scripts.

Now for dbutils!!

Back to our task at hand!

- Read in the property file in `python` from the data folder in your project folder.

```
import polars as pl

df = pl.read_csv('data/property.csv')

df.glimpse()
```

Now for dbutils!!

Next connect to the DB

```
import polars as pl
from decouple import config
from dbutils import Query

df = pl.read_csv('data/property.csv')

database = Query(
    db_type = 'mysql',
    db_name = 'workshop',
    db_user = config('db_user'),
    db_pass = config('db_pass'),
    db_host = config('db_host'),
    db_port = config('db_port')
)

database.__version__
database.db_type
```

Now for dbutils!!

Once you have connected, its always good practice to test the connection:

```
import polars as pl
from decouple import config
from dbutils import Query

df = pl.read_csv('data/property.csv')

database = Query(
    db_type = 'mysql',
    db_name = 'workshop',
    db_user = config('db_user'),
    db_pass = config('db_pass'),
    db_host = config('db_host'),
    db_port = config('db_port')
)

database.__version__
database.db_type

database.sql_query(sql = "SELECT * FROM property", limits = 2)
```

Now for dbutils!!

Now that we have established a connection, lets `truncate` the database and load in the data through `python` using the `sql_execute` method.

```
database.sql_execute(sql = "TRUNCATE property")
```

Next, we can upload the data to DB using `sql_write`:

```
database.sql_write(df.to_pandas(), table_name = 'property')
```

Exercises for SQL & Polars

Execute the following command in `polars` and then use `dbutils` to execute the commands in `sql`

- What is the average and standard deviation of house prices?
- How much more expensive is adding an extra bedroom and going from a 2 to 3 bedroom if I RENT?
- Where are the most expensive houses for sale?

40:00



Learning Polars with Python

Section 4

```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful **is** better than ugly.

Explicit **is** better than implicit.

Simple **is** better than complex.

Complex **is** better than complicated.

Flat **is** better than nested.

Sparse **is** better than dense.

Readability counts.

Special cases aren't **special enough to break the rules.**

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now **is** better than never.

Although never **is** often better than **right** now.

If the implementation **is** hard to explain, it's **a bad idea.**

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!




Building basic functions



Modules vs Classes vs Function

Module

In the world of python, a `module` can be thought of as a collection of `functions` / `Classes` that reside within a  package.

Class

Logical abstraction layer to organise certain methods to a specific objects. Ex. An object of class `bird` can have `swim` method, but class `fish` cannot have method `fly`.¹ We usually use `Classes` as a blueprint to easily *instantiate* new objects with a set structure. These will attributes, which we can access through *methods*.

Function

Standalone not associated with any `class` (example `add(1,2)`). Invoke by own name, does not require `self`.

¹ Except for those crazy flying fish!

Modules vs Classes vs Function?

```
x = Person(Name = "Hanjo")
```

Class

```
x.jump()
```

method

```
show_menu("vegetarian")
```

function

```
Connect("greenplum")
```

Class



Building our first class



The most important!

- What is the most important thing about writing code?

!!!DOCUMENTATION!!!

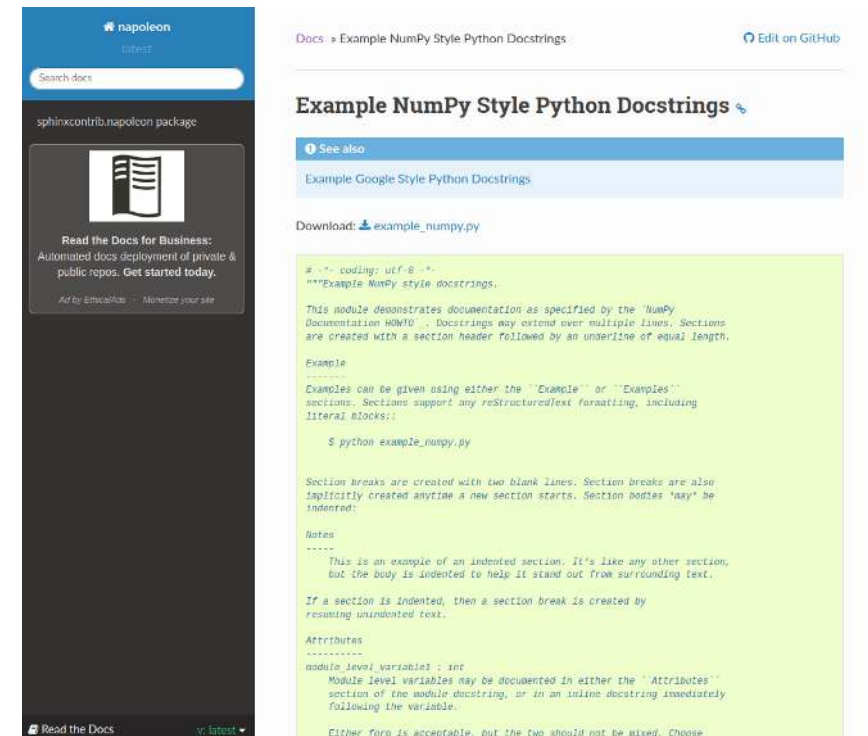


Documentating functions in python

By learning to write your documentation as you write your code, future you and other people will appreciate and enjoy working with you (and your code) a lot more.¹

We start by selecting a *style* of *docstrings*. I certainly prefer writing my code in a more verbose way. So, with that in mind, I prefer **NumPy**.

- It is more verbose.
- Plays well with Sphinx doc generator.
- De-facto standard of the larger projects in python.



The image shows two side-by-side screenshots. The left screenshot is a dark-themed Sphinx documentation page for the 'sphinxcontrib.napoleon' package. It features a search bar, a 'Read the Docs for Business' advertisement, and a 'Read the Docs' footer. The right screenshot is a light-themed Sphinx documentation page titled 'Example NumPy Style Python Docstrings'. It includes a 'See also' section with a link to 'Example Google Style Python Docstrings', a 'Download' section for 'example_numpy.py', and a code block showing a docstring example. The code block contains the following text:

```
# -*- coding: utf-8 -*-
"""Example NumPy style docstrings.

This module demonstrates documentation as specified by the 'NumPy
Documentation HOWTO'. Docstrings may extend over multiple lines. Sections
are created with a section header followed by an underline of equal length.

Example
-----
Examples can be given using either the "Example" or "Examples"
sections. Sections support any reStructuredText formatting, including
literal blocks::

    $ python example_numpy.py

Section breaks are created with two blank lines. Section breaks are also
implicitly created anytime a new section starts. Section bodies may be
indented:

Notes
-----
This is an example of an indented section. It's like any other section,
but the body is indented to help it stand out from surrounding text.

If a section is indented, then a section break is created by
resisting unindented text.

Attributes
-----
module_level_variable1 : int
    Module level variables may be documented in either the "Attributes"
    section of the module docstring, or in an inline docstring immediately
    following the variable.

    Either form is acceptable, but the two should not be mixed. Choose
```

¹If you want to go deep, read up on [PEP 257 -- Docstring Conventions](#)

What should be documented?

Although there is A LOT of things that can be documented, lets start with the two most basics: `Attribute`, `Methods`

- Attributes
 - Think of these as characteristics of the class.
- Methods
 - Operations or actions that the class can perform.

When documenting, please be very aware of capital letters, spaces and ":". All of these things are parsed by our document parser which you will see later on 🙌.

Code is more often read than written.

— Guido van Rossum

What should be documented?

```
class Boilerplate:
```

```
    """
```

```
    Description of class
```

```
    Attributes
```

```
    -----
```

```
    attr_1 : type  
           description
```

```
    attr_2 : type  
           description
```

```
    Methods
```

```
    -----
```

```
    method_1(param=None)  
           Description
```

```
    """
```

```
    attr_1 = "Thanks for loading class with {attr_1}"
```

Take some time and build be a Gorilla `class`!



15:00

My Gorilla Class?

In python we use the object `self` to represent the instance of a class. By using the `self` keyword we access the `attributes` and `methods` of the class we created and we use `__init__` to mean the *initiator*:

```
class Gorilla:
    """
    A class used to represent a Gorilla

    Attributes
    -----
    name : str
        the name of the gorilla
    weight : int
        how much does the gorilla weight in KG
    home: str
        where can the gorilla be found
    age: int
        number of years
    sex: str
        is the gorilla male or female
    hours: int
        how many hours a gorilla sleep

    Methods
    -----
    sleep(hours=None)
        How many hours does my gorilla sleep
    """
    def __init__(self, name, weight, age, sex, hours, home = "Viruga"):
        self.name = name
        self.weight = weight
        self.home = home
        self.age = age
        self.sex = sex
        self.hours = hours
```

How to add method new class?

```
def sleep(self, hours = None):
    """
    Tells you how many hours the gorilla sleeps.

    If the argument `hours` isn't passed in, the default Gorilla
    hours is used.

    Parameters
    -----
    sleep(hours=None)
        How many hours does my gorilla sleep

    Raises
    -----
    NotImplementedError
        If no hour is set for the gorilla or passed in as a
        parameter.
    """

    if self.hours is None and hours is None:
        raise NotImplementedError("Gorillas need to sleep at some stage in the day!")
    out_hours = self.hours if hours is None else hours
    information = "I am a {sex} gorilla called {name},\nI weight {weight}Kg,\nI am from {home} and \nI love to sleep {hours} hours a day!"
    print(information.format(sex = self.sex, name = self.name, weight = self.weight, home = self.home, hours = out_hours))
```

What else can a gorilla do?

15:00

Lets now see how we built the Gorilla!

```
mygorilla = Gorilla(name = "hanjo",  
                    weight = 200,  
                    age = 20,  
                    sex = "male",  
                    hours= 10)
```

```
>>> mygorilla.sleep(hours = 0)
```

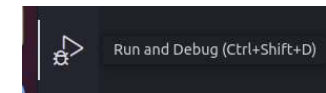
```
>>> mygorilla.sleep(hours = 12)  
# I am a male gorilla called hanjo,  
# I weight 200Kg,  
# I am from Viruga and  
# I love to sleep 12 hours a day!
```

```
class Gorilla(builtins.object)  
|   Gorilla(name, weight, age, sex, hours, home='Viruga')  
|  
|   A class used to represent a Gorilla  
Attributes
name : str
the name of the gorilla
weight : int
how much does the gorilla weight in KG
home: str
where can the gorilla be found
age: int
number of years
sex: str
is the gorilla male or female
hours: int
how many hours a gorilla sleep
Methods
-----
sleep(hours=None)
How many hours does my gorilla sleep
Methods defined here:
__init__(self, name, weight, age, sex, hours, home='Viruga')
Initialize self. See help(type(self)) for accurate signature.
sleep(self, hours=None)
Tells you how many hours the gorilla sleeps.
If the argument `hours` isn't passed in, the default Gorilla
hours is used.
Parameters
-----
sleep(hours=None)
How many hours does my gorilla sleep
Raises
-----
NotImplementedError
If no hour is set for the gorilla or passed in as a parameter.
-----
Data descriptors defined here:
__dict__
dictionary for instance variables (if defined)
```

Debugging

There are going to be moments (100% sure of this), when you will have to deal with a code in one of your pieces of code. This is when we need going to jump into that function using a debugger and be able to explore the state of the environment as it is at that point. You can also use the keyboard shortcut `Ctrl+Shift+D`.

- Continue / Pause `F5`
- Step Over `F10`
- Step Into `F11`
- Step Out `Shift+F11`
- Restart `Ctrl+Shift+F5`
- Stop `Shift+F5`

A screenshot of a Python IDE (likely VS Code) showing a debugger interface. The main editor displays a Python class named 'Gorilla' with a 'sleep' method. The code is partially highlighted in green, indicating the current execution point. A red arrow points to the 'Run and Debug' button in the top right corner. Another red arrow points to the 'TERMINATE' button in the bottom right corner of the IDE. The interface also shows a 'VARIABLES' pane on the left, a 'WATCH' pane, and a 'CALL STACK' pane. The bottom of the IDE shows the 'TERMINATE' button and a 'HELP' button.

05:00



Its go time



bnrUtils ✕

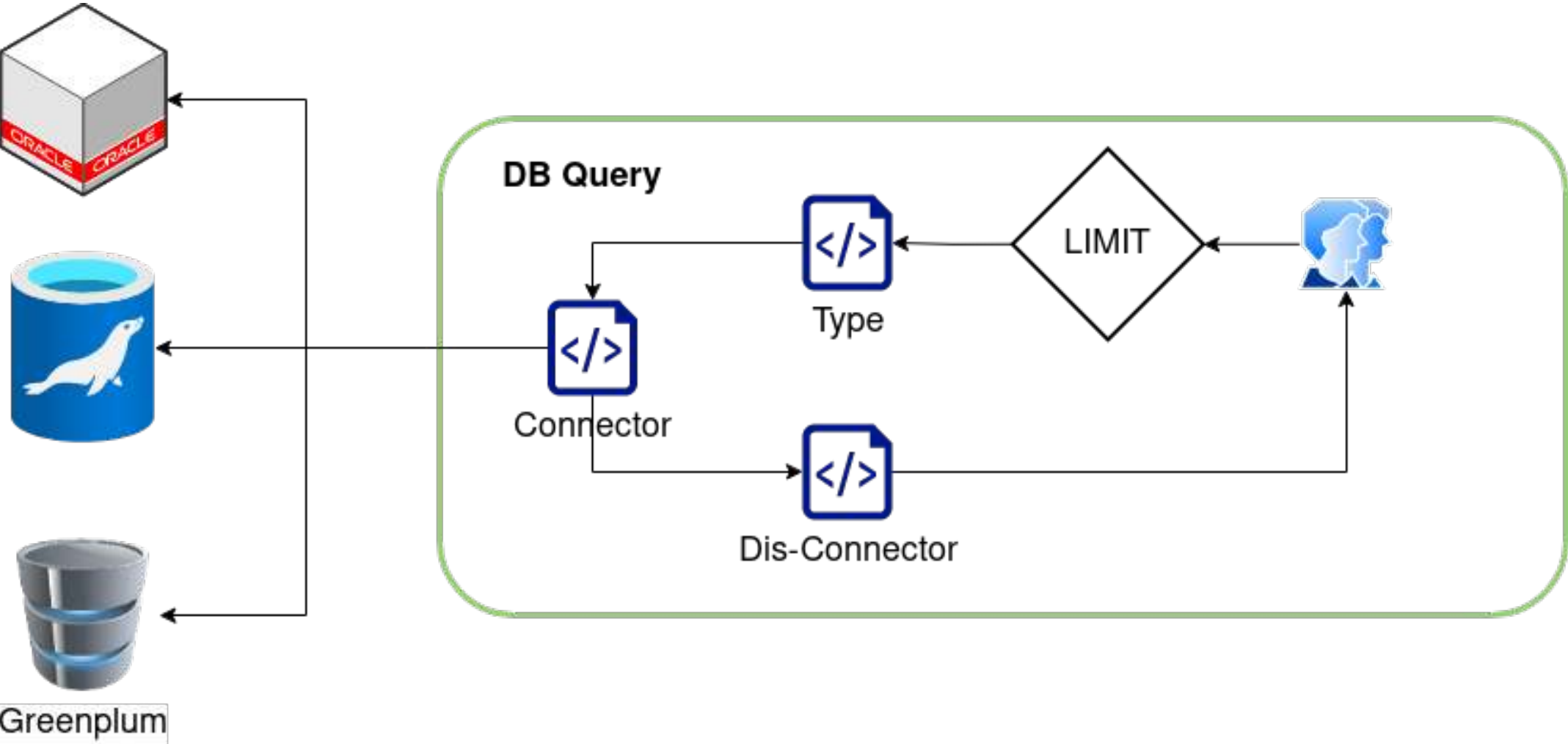
Function over form

- Start by building out small functions that do **one** thing and does this well. To do this, break down the steps for yourself in graphical format.

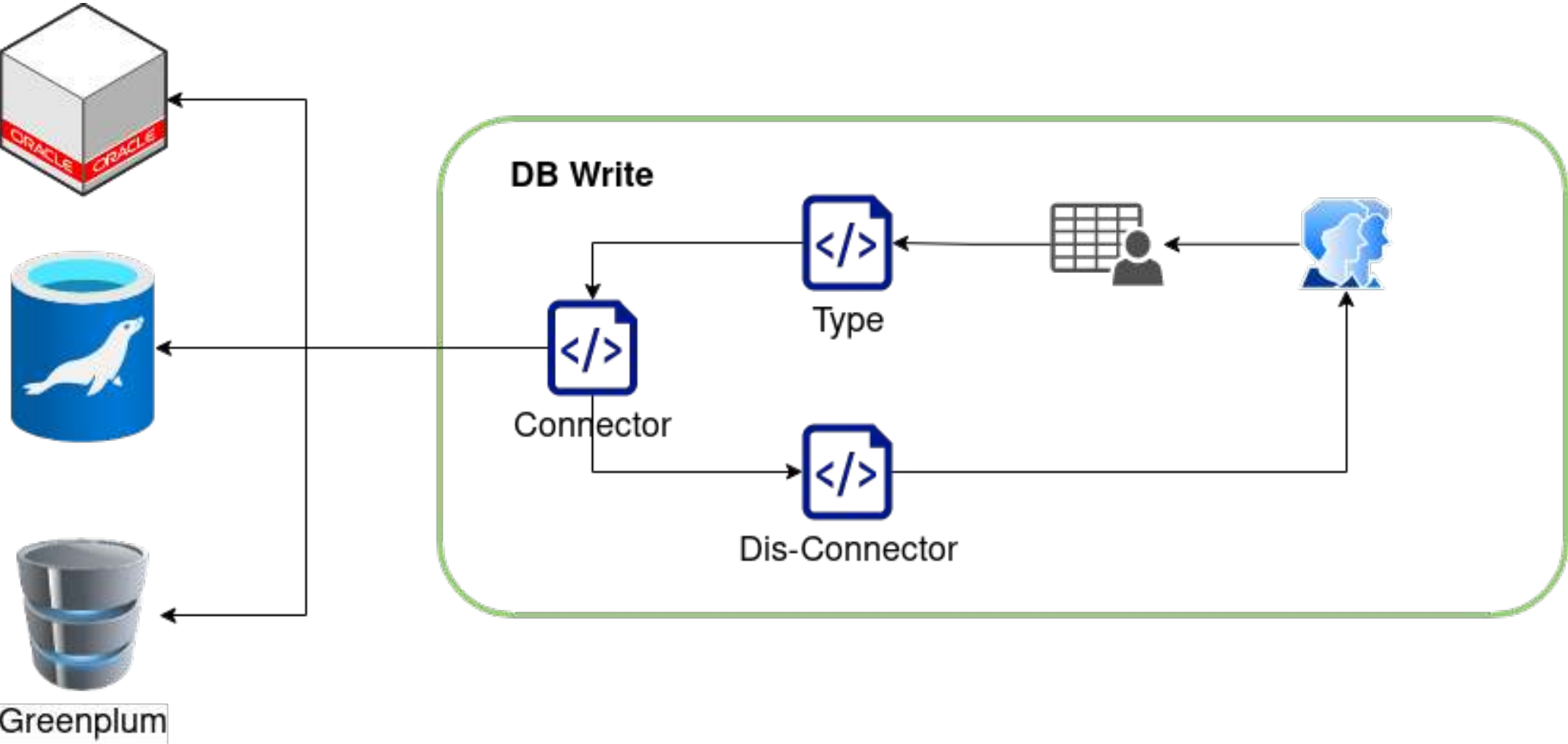
To do this, take 10:00 and draft a design in [Draw.io](#) for a `python` package that connects to different databases:

10:00

DB Write and Query



DB Write and Query



Final Functions

After drawing up some basic functions I also realised that I might want some other utilities that could give me an idea on table column info etc. So my final function package looked like. In this workshop we gonna focus on one method only - `query`:¹

- Class: `Query`
 - `query`
 - `show_tables`
 - `table_info`
 - `write_to_db`

These collection of `classes` and `methods` would make up the basic building blocks of my package. In this workshop today we will focus on `MariaDb` as our primary `DB`. Then next week I will assist in expanding the package for in-house use cases.

Most importantly, we will only use a `logger` to keep track of processing times. This will become very useful in production settings!

¹ I would recommend you try the other two methods on your own. It can only benefit in the long run.

Putting it all into a package

To put all of your functions into a package, there are some simple steps to follow. Some of them make your life easier, others are mandatory:

1) Use virtual environments (life) 2) Create a folder where your package will live 3) Create `setup.py` file to configure package 4) Create a make file to build your package 5) Create `requirements.txt` 6) Create `__init__.py` file 7) Document the package!

Using environments for package development

At its core, the main purpose of Python virtual environments is to create an isolated environment for Python projects. This means that each project can have its own dependencies, regardless of what dependencies every other project has. Its a good way to make sure your namespacing is working correctly.

```
pip install virtualenv
sudo apt-get install python3-venv
```

Next create a directory to store your environments, create the virtual environment and activate it:

```
cd ~
mkdir python-virtual-environments && cd python-virtual-environments
python3 -m venv bnr_utils
source bnr_utils/bin/activate
# deactivate
```

Create working directory

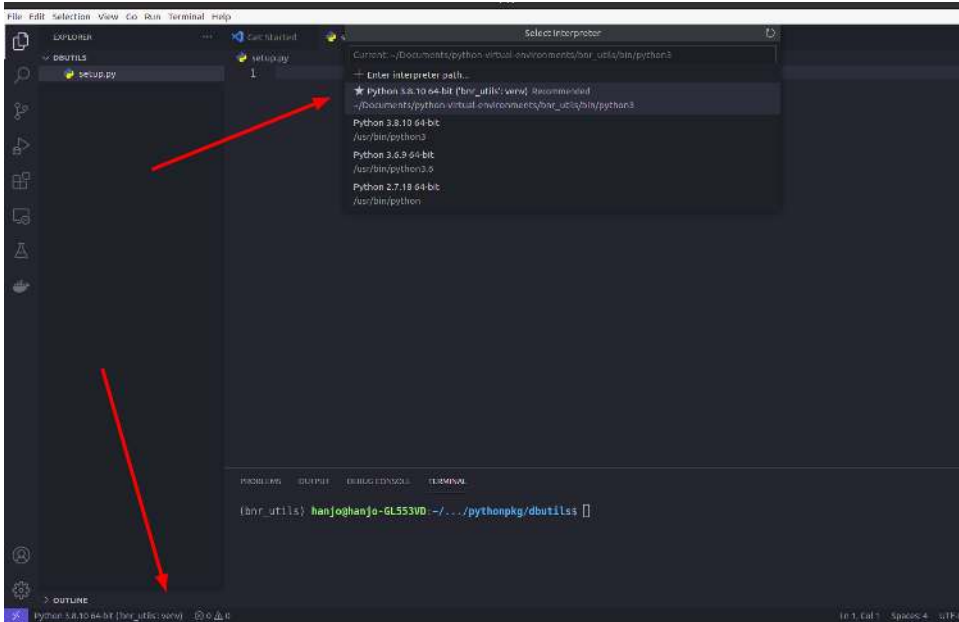
Its a good idea for you to keep your packages in a single directory. That way you can easily jump between developments:

```
cd ~  
mkdir -p pythonpkg/dbutils && cd pythonpkg/dbutils
```

Finally we can start by creating our `setup.py` file. When this file is present in a folder it gives an indication of the basic setup requirements as well as some basic information.

Setup.py

Open the folder in `VScode` and remember to activate the environment:



This file has to sit at the root directory and it starts off by containing basic meta information on the package:

```
import pathlib
from setuptools import setup, find_packages

HERE = pathlib.Path(__file__).parent

VERSION = '0.1.0'
PACKAGE_NAME = 'dbutils'
AUTHOR = 'Hanjo Odendaal'
AUTHOR_EMAIL = 'hanjo@71point4.com'
URL = 'XXX'

LICENSE = 'MIT'
DESCRIPTION = 'DB Utils Python Package to Connect to Da
LONG_DESCRIPTION = (HERE/"README.md").read_text()
LONG_DESC_TYPE = "text/markdown"
```

Setup.py

Next we start specifying what dependencies the package might have:

```
INSTALL_REQUIRES = [  
    "setuptools ≈ 47.1.1",  
    "pandas ≈ 1.0.5",  
    "pytest=6.0.1",  
    "requests ≈ 2.24.0",  
    "Sphinx=3.2.1",  
    "sphinx-rtd-theme=0.5.0",  
    "m2r2=0.2.5",  
    "sqlalchemy ≈ 1.3.20",  
    "pymysql ≈ 1.0.2",  
    "decouple=3.4.0",  
    "logger=1.4",  
]
```


Setup.py

Finally we push all this information into the setup function:

```
setup(name=PACKAGE_NAME,  
      version=VERSION,  
      description=DESCRIPTION,  
      long_description=LONG_DESCRIPTION,  
      long_description_content_type=LONG_DESC_TYPE,  
      author=AUTHOR,  
      license=LICENSE,  
      author_email=AUTHOR_EMAIL,  
      url=URL,  
      install_requires=INSTALL_REQUIRES,  
      packages=find_packages()  
    )
```

Setup.py

What the final `setup.py` file should look like accompanied by a `makefile` file:

```
# Use bump2version before creating a new release.  
#  
release:  
    python3 setup.py sdist bdist_wheel
```

```
import pathlib  
from setuptools import setup, find_packages  
  
HERE = pathlib.Path(__file__).parent  
  
VERSION = '0.1.0'  
PACKAGE_NAME = 'dbutils'  
AUTHOR = 'Hanjo Odendaal'  
AUTHOR_EMAIL = 'hanjo@71point4.com'  
URL = 'XXX'  
  
LICENSE = 'MIT'  
DESCRIPTION = 'DB Utils Python Package to Connect to Databases'  
LONG_DESCRIPTION = (HERE/"README.md").read_text()  
LONG_DESC_TYPE = "text/markdown"  
  
INSTALL_REQUIRES = [  
    "setuptools ≈ 47.1.1",  
    "pandas ≈ 1.0.5",  
    "pytest ≈ 6.0.1",  
    "requests ≈ 2.24.0",  
    "Sphinx ≈ 3.2.1",  
    "sphinx-rtd-theme ≈ 0.5.0",  
    "m2r2 ≈ 0.2.5",  
    "sqlalchemy ≈ 1.3.20",  
    "pymysql ≈ 1.0.2",  
    "python-decouple ≈ 3.5",  
    "logger ≈ 1.4",  
]  
  
setup(name=PACKAGE_NAME,  
      version=VERSION,  
      description=DESCRIPTION,  
      long_description=LONG_DESCRIPTION,  
      long_description_content_type=LONG_DESC_TYPE,  
      author=AUTHOR,  
      license=LICENSE,  
      author_email=AUTHOR_EMAIL,  
      url=URL,  
      install_requires=INSTALL_REQUIRES,  
      packages=find_packages()  
)
```

Requirements.txt

It is always a nice idea to have a `requirements.txt` file in the project. This makes setup of the environment a lot easier and `pip` has built in functions to install from these special files. At the same time you can ensure that the correct working package version is installed with the package.

```
pip install -r requirements.txt
```

```
setuptools ≈ 47.1.1  
pandas ≈ 1.0.5  
pytest=6.0.1  
requests ≈ 2.24.0  
Sphinx=3.2.1  
sphinx-rtd-theme=0.5.0  
m2r2=0.2.5  
sqlalchemy ≈ 1.3.20  
pymysql ≈ 1.0.2  
logger=1.4  
python-decouple=3.5  
wheel=0.37.1
```

Basics done!

What is the final ingredient to make this a package?

Files named **init.py** are used to mark directories on disk as Python package directories.

```
from .query import Query
```

This file should be in the folder where our modules will be stored:

```
mkdir dbutils && cd dbutils && touch __init__.py
```

At this point your folder structure should look something like:

```
.
├── dbutils
│   ├── dbutils
│   │   └── __init__.py
│   ├── makefile
│   ├── requirements.txt
│   └── setup.py
```

2 directories, 4 files



dbutils 



Writing our main class: Query

In writing my main class, you will see that I am *pimping* my class quite a lot. The additional sets of information will be very useful for you to debug, as well as have standard outputs to keep logs when your package goes into production.

Create a file called `Query.py` in the `dbutils` folder and add these two lines:

```
import logging
from pkg_resources import get_distribution
logging.basicConfig(level=logging.DEBUG)
```

- This allows me to use logging in throughout my package.
 - Here I set it to log at the `DEBUG` level.
- I will always have a variable saved to `self` to get the version of the package that produced the results.

Take 10min to write the *Boilerplate* for a class called `Query` **boilerplate**

10:00

Writing our main class: Query

```
import logging
from pkg_resources import get_distribution
logging.basicConfig(level=logging.DEBUG)

class Query():
    """
    Initialization method of the :code:`Query` class.

    Attributes
    -----
    db : str
        The name of the database.
    db_host : str
        Host of DB. [USE ENVIRONMENT VARIABLES]
    db_port : str
        Port where DB listening.[USE ENVIRONMENT VARIABLES]
    db_user : str
        Username [USE ENVIRONMENT VARIABLES].
    db_pass : str
        Password. [USE ENVIRONMENT VARIABLES]

    Methods
    -----
    Fill this space
    """
    def __init__(self, **kwargs):
        __version__ = get_distribution('dbutils').version
        self.db = kwargs["db"]
        self.db_host = kwargs["db_host"]
        self.db_port = kwargs["db_port"]
        self.db_user = kwargs["db_user"]
        self.db_pass = kwargs["db_pass"]
```

Building the package for shipping

We are finally ready to test whether our package builds successfully.

The final bit of information that is needed to finalize it all is... you guessed it, more documentation!

So create a `README.md` you should properly populate this file to explain what the package does and how it functions. I am not going to go in detail into `markdown` in this workshop.

For now make the file and add a very simple header. In `README.md`:

```
# Hello
```

Finally we can test! Navigate to your parent directory where the package lives: `/pathtopackage/pythonpkg/`

```
make -C dbutils/
```


Install package for testing

Having created a `setup.py`, test the install with `pip`. In the folder `pythonpkg/dbutils`:

```
pip install .
```

If you have an existing install, and want to ensure package and dependencies are updated use `--upgrade` along with `pip`:

```
pip install --upgrade .
```

To uninstall (use package name):

```
pip uninstall dbutils
```

Building our connector

It is time to now finally use our Query class to connect to the database. The `connect` utility will be the workhorse of the package and will facilitate the connection between python and which ever database you have.

Create a file called `_utils.py`.¹

```
import logging
log = logging.getLogger(__name__)
from sqlalchemy import create_engine
import pymysql

def _connect(self, params):
    """Execute query"""
    log.debug(f"Connecting to {self.db_host}")

    db_connection_str = f"mysql+pymysql://{self.db_user}:{self.db_pass}@localhost/{self.db}"
    db_connection = create_engine(db_connection_str)

    return db_connection
```

¹ I use underscore to denote auxiliary functions OR functions that are suppose to be hidden to user purely because it should not concern the user. Please don't judge me.

Using connector in a module

```
import pandas as pd
from ._utils import _connect
import logging

log = logging.getLogger(__name__)

def sql_query(self, **kwargs):
    try:
        db_connection = _connect(self, kwargs)

        log.debug(db_connection)
        txt = kwargs['sql']

        if "limits" in kwargs:
            txt = txt + f" limit {kwargs['limits']}"
            log.debug(txt)

        df = pd.read_sql_query(txt, con = db_connection)

        return(df)

    except Exception:
        raise
    finally:
        db_connection.dispose()
```

Importing your module into class

```
import logging
from pkg_resources import get_distribution
logging.basicConfig(level=logging.DEBUG)

class Query():
    """
    Initialization method of the :code:`Query` class.

    Attributes
    -----
    db : str
        The name of the database.
    db_host : str
        Host of DB. [USE ENVIRONMENT VARIABLES]
    db_port : str
        Port where DB listening.[USE ENVIRONMENT VARIABLES]
    db_user : str
        Username [USE ENVIRONMENT VARIABLES].
    db_pass : str
        Password. [USE ENVIRONMENT VARIABLES]

    Methods
    -----
    Fill this space
    """
    def __init__(self, **kwargs):
        __version__ = get_distribution('dbutils').version
        self.db = kwargs["db"]
        self.db_host = kwargs["db_host"]
        self.db_port = kwargs["db_port"]
        self.db_user = kwargs["db_user"]
        self.db_pass = kwargs["db_pass"]

    from ._sql_query import sql_query
```

Using package in a main application

We can test our new package by creating a file called `dev.py` and a `.env` file.

* `.env`

- This file will store all your environment variables that you might not want to expose to the outside world. This functionality comes from the `python-decouple` package and is a must when developing code. I will 🔥 your laptop down if I find plain text passwords.¹
 - `dev.py`
- This file be our little sandbox where we test out certain functionality.
- Once you get more comfortable with software development, you should start writing unit-tests within your code. I do not cover it today, even though its best practice, purely because of time constraint.

¹ Next time we meet I want to see that all of you have a LastPass account. If you save your passwords in a plain text file, I will report you to management.

Using package in a main application

The `dev.py` file will be broken down into three distinct sections: (1) Imports, (2) Logger setup and (3) Main:

```
import logging
from decouple import config
from dbutils import Query
import pandas as pd

def setup_logger():
    # create logger
    logger = logging.getLogger('dbutils')
    # logger.setLevel(logging.DEBUG)
    logger.setLevel(logging.INFO)

    # create console handler and set level to debug
    ch = logging.StreamHandler()
    ch.setLevel(logging.DEBUG)

    # create formatter
    formatter = logging.Formatter('%(asctime)s [%(levelname)s] %(name)s: %(message)s')

    # add formatter to ch
    ch.setFormatter(formatter)

    # add ch to logger
    logger.addHandler(ch)
```

Using package in a main application

The final piece of the script contains our `main` function:

```
def main():
    setup_logger()

    database = Query(
        db_type = 'mysql',
        db_name = 'workshop',
        db_user = config('db_user'),
        db_pass = config('db_pass'),
        db_host = config('db_host'),
        db_port = config('db_port')
    )

    print(database.sql_query(sql = "SELECT * FROM user", limits = 5))
    print(database.sql_query())

if __name__ == '__main__' and __package__ is None:
    print(f"Running main file {__name__}")
    main()
```

Using package in a main application

If all went well, you should see:

```
>>> if __name__ == '__main__' and __package__ is None:
...     print(f"Running main file {__name__}")
...     main()
...
Running main file __main__
      Host      User          Password  ...  is_role  default_role  max_statement_time
0  localhost   root  *3CD53EE62F8F7439157DF288B55772A2CA36E60C  ...      N
1  localhost  ubuntu *A3167888E0A634C04BFDE00EF3FD267117402DBA  ...      N
[2 rows x 46 columns]
```


See you in 30min



LoofandTimmy.com

Bringing your documentation to life

One of the nice things about documenting your code as you write it, is that once you have finished, you can use tools such as `sphinx` to automatically build a website document for you!

```
sudo apt install python-sphinx -y
mkdir docs
cd docs
sphinx-quickstart
pip install sphinxcontrib-napoleon
```

Then go and edit:

```
.
├── Makefile
├── build
├── make.bat
└── source
    ├── _static
    ├── _templates
    ├── conf.py
    └── index.rst
```

4 directories, 4 files

conf.py

```
import os
import sys
sys.path.insert(0, os.path.abspath('..'))

extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.viewcode',
    'sphinxcontrib.napoleon'
]
```

The final step

This last step is purely optional, but I think it makes the project look very professional.

This Sphinx theme was designed to provide a great reader experience for documentation users on both desktop and mobile devices. This theme is commonly used with projects on Read the Docs but can work with any Sphinx project.

Some more [themes](#)

```
sudo pip install sphinx_rtd_theme
```

Again in `conf.py` add `html_theme = "sphinx_rtd_theme"`. Then the moment of truth:

```
sphinx-apidoc -o . ..  
make html
```

dbutils

Navigation

Quick search

Go

Welcome to dbutils's documentation!

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

©2022, Hanjo Odendaal. | Powered by [Sphinx 3.2.1](#) & [Alabaster 0.7.12](#) | [Page source](#)

The final step

This last step is purely optional, but I think it makes the project look very professional.

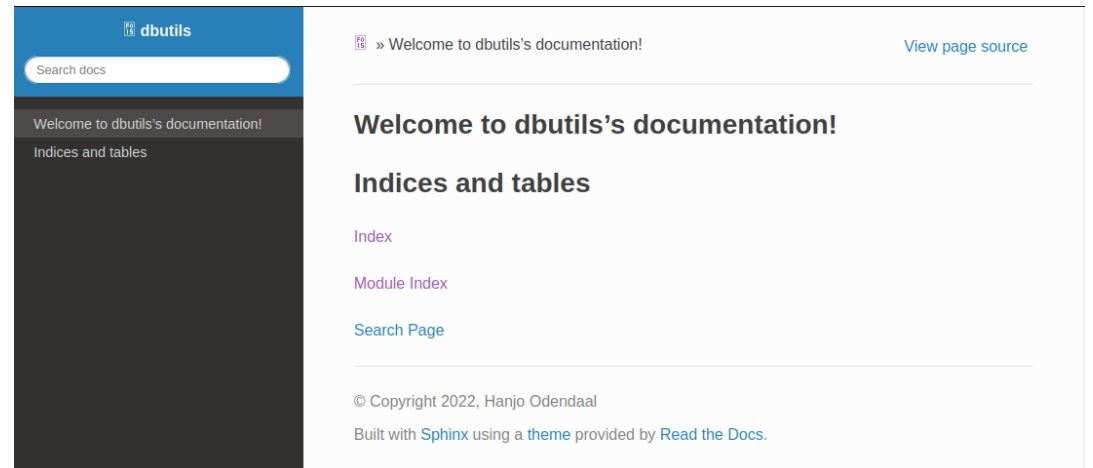
This Sphinx theme was designed to provide a great reader experience for documentation users on both desktop and mobile devices. This theme is commonly used with projects on Read the Docs but can work with any Sphinx project.

Some more [themes](#)

```
sudo pip install sphinx_rtd_theme
```

Again in `conf.py` add `html_theme = "sphinx_rtd_theme"`. Then the moment of truth:

```
sphinx-apidoc -o . ..  
make html
```



Be free!



Final task

- You love your new package, but dont always know what tables are available, can you create a new method to bring back a data frame with the available tables as rows?

```
import pandas as pd
from sqlalchemy import inspect
from ._utils import _connect
import logging

log = logging.getLogger(__name__)

def sql_show_tables(self, **kwargs):
    try:
        db_connection = _connect(self, kwargs)

        log.debug(db_connection)

        insp = inspect(db_connection)
        table_names = insp.get_table_names()
        out = pd.DataFrame(table_names, columns=['table_names'])

        return(out)

    except Exception:
        raise
    finally:
        db_connection.dispose()
```