Engineering for Data Scientists

Valohai

Valohai

Contents

About the author

Juha Kiili

Senior Software Developer, Product Owner at Valohai

Senior Software Developer with gaming industry background shape-shifted into a full-stack ninja. I have the biggest monitor.

Foreword

Software engineering has come a long way. It's no longer just about getting a functioning piece of code on a floppy disk; it's about the craft of making software. There's a good reason for it too. Code lives for a long time.

Thus there are a lot of strong opinions about good engineering practices that make developing software for the long haul possible and more enjoyable. I think enjoyability is an important word here because most software developers know the pain of fixing poorly developed and poorly documented legacy software.

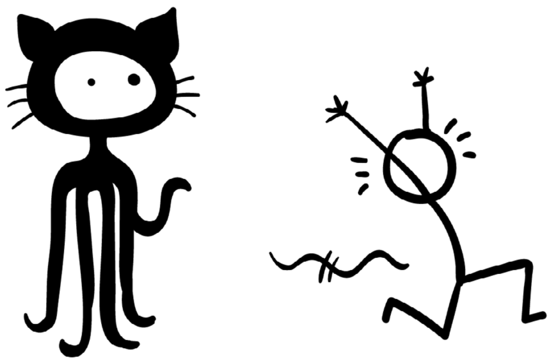
Data scientists are also entering this world because machine learning is becoming a core part of many products. While a heterogenous bunch with various backgrounds, data scientists are more commonly from academia and research than software engineering. The slog of building and maintaining software isn't as familiar as it is to most developers, but it will be soon enough. It's better to be prepared with a solid foundation of best practices, so it'll be easier to work with software engineers, and it'll be easier to maintain what you build.

This eBook is to help pick up engineering best practices with simple tips. I hope that we can teach even the most seasoned pros something new and get you talking with your team on how you should be building things. Remember, as machine learning becomes a part of software products, it too will live for a long time.

This eBook isn't about Valohai – although there is a section about our MLOps platform at the end – but good engineering is close to our heart.

Git

What is Git?



Git is a version control system designed to track changes in a source code over time.

When many people work on the same project without a version control system it's total chaos. Resolving the eventual conflicts becomes impossible as none has kept track of their changes and it becomes very hard to merge them into a single central truth. Git and higher-level services built on top of it (like Github) offer tools to overcome this problem.

Usually, there is a single central repository (called "origin" or "remote") which the individual users will clone to their local machine (called "local" or "clone"). Once the users have saved meaningful work (called "commits"), they will send it back ("push" and "merge") to the central repository.

What is the difference between Git & GitHub?

Git is the underlying technology and its command-line client (CLI) for tracking and merging changes in a source code.

GitHub is a web platform built on top of git technology to make it easier. It also offers additional features like user management, pull requests, automation. Other alternatives are for example GitLab and Sourcetree.

Terminology

- **Repository** – "Database" of all the branches and commits of a single project
- **Branch** – Alternative state or line of development for a repository.
- **Merge** – Merging two (or more) branches into a single branch, single truth.
- **Clone** – Creating a local copy of the remote repository.
- **Origin** – Common alias for the remote repository which the local clone was created from

- **Main / Master** – Common name for the root branch, which is the central source of truth.
- **Stage** – Choosing which files will be part of the new commit
- **Commit** – A saved snapshot of staged changes made to the file(s) in the repository.
- **HEAD** – Shorthand for the current commit your local repository is currently on.
- **Push** – Pushing means sending your changes to the remote repository for everyone to see
- **Pull** – Pulling means getting everybody else's changes to your local repository
- **Pull Request** – Mechanism to review & approve your changes before merging to main/master

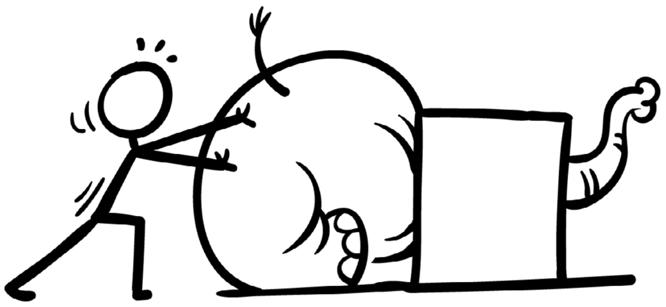
Basic commands

- `git init` (Documentation) – Create a new repository on your local computer.
- `git clone` (Documentation) – Start working on an existing remote repository.
- `git add` (Documentation) – Choose file(s) to be saved (staging).
- `git status` (Documentation) – Show which files you have changed.
- `git commit` (Documentation) – Save a snapshot (commit) of the chosen file(s).
- `git push` (Documentation) – Send your saved snapshots (commits) into the remote repository.
- `git pull` (Documentation) – Pull recent commits made by others into your local computer.
- `git branch` (Documentation) – Create or delete branches.
- `git checkout` (Documentation) – Switch branches or undo changes made to local file(s).
- `git merge` (Documentation) – Merge branches to form a single truth.

Rules of thumb for Git

Don't push datasets

Git is a version control system designed to serve software developers. It has great tooling to handle source code and other related content like configuration, dependencies, documentation. It is not meant for training data. Period. Git is for code only.



In software development, code is king and everything else serves the code. In data science, this is no longer the case and there is a duality between data and code. It doesn't make sense for the code to depend on data any more than it makes sense for data to depend on code. They should be decoupled and this is where the code-centric software development model fails you. **Git shouldn't be the central point of truth for a data science project.**

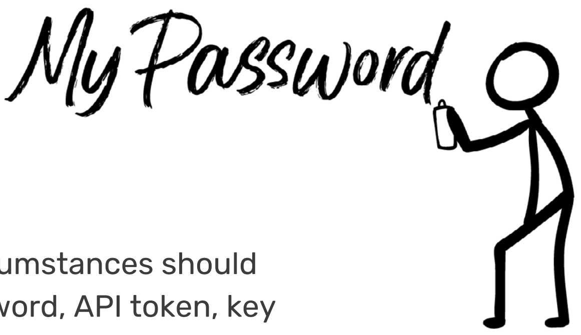
There are extensions like LFS that refer to external datasets from a git repository. While they serve a purpose and solve some of the technical limits (size, speed), they do not solve the core problem of a code-centric software development mindset rooted in git.

You will always have datasets floating around in your local directory though. It is quite easy to accidentally stage and commit them if you are not careful. The correct way to make sure that you don't need to worry about datasets with git is to use the `.gitignore` config file. Add your datasets or data folder into the config and never look back.

Example:

```
# ignore archives
*.zip
*.tar
*.tar.gz
*.rar

# ignore dataset folder and subfolders
datasets/
```

Don't push secrets



This should be obvious, yet the constant real-world mistakes prove to us it is not. It doesn't matter if the repository is private either. In no circumstances should anyone commit any username, password, API token, key code, TLS certificates, or any other sensitive data into git.

Even private repositories are accessible by multiple accounts and are also cloned to multiple local machines. This gives the hypothetical attacker exponentially more targets. Remember that private repositories can also become public at some point.

Decouple your secrets from your code and pass them using the environment instead. For Python, you can use the common `.env` file with which holds the environment variables, and the `.gitignore` file which makes sure that the `.env` file doesn't get pushed to the remote git repository. It is a good idea to also provide the `.env.template` so others know what kind of environment variables the system expects.

.env:

```
API_TOKEN=98789fsda789a89sdafsa9f87sda98f7sda89f7
```

.env.template:

```
API_TOKEN=
```

.gitignore:

```
.env
```

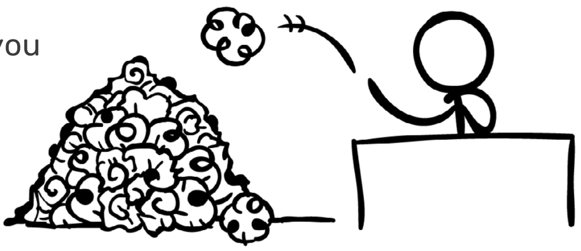
hello.py:

```
from dotenv import load_dotenv
load_dotenv()
api_token = os.getenv('API_TOKEN')
```

This still requires some manual copy-pasting for anyone cloning the repository for the first time. For more advanced setup, there are encrypted, access-restricted tools that can share secrets through the environment, such as Vault.

Note: If you already pushed your secrets to the remote repository, do not try to fix the situation by simply deleting them. It is too late as git is designed to be immutable. Once the cat is out of the bag, the only valid strategy is to change the passwords or disable the tokens.

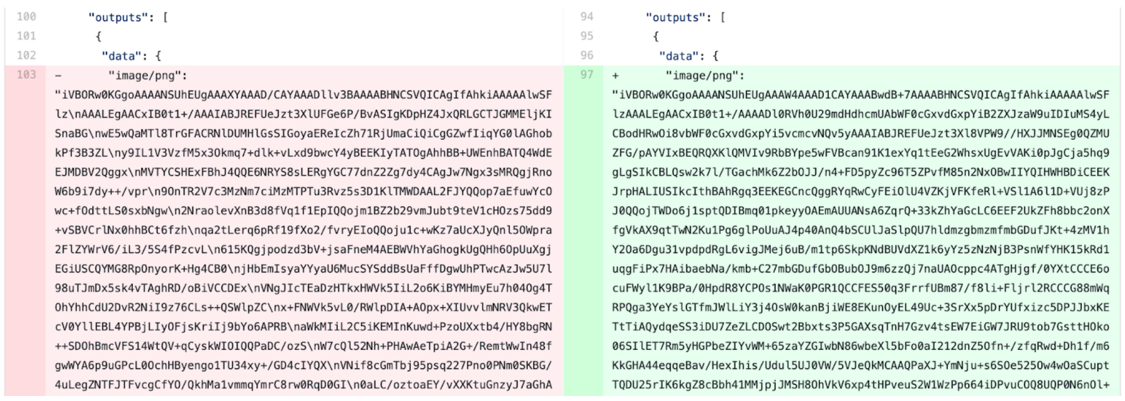
Don't push notebook outputs



Notebooks are cool because they let you not only store code but also the cell outputs like images, plots, tables. The problem arises when you commit and push the notebook with its outputs to git.

The way notebooks serialize all the images, plots, and tables is not pretty. Instead of separate files, it encodes everything as JSON gibberish into the `.ipynb` file. This makes git confused.

Git thinks that the JSON gibberish is equally important as your code. The three lines of code that you changed are mixed with three thousand lines that were changed in the JSON gibberish. Trying to compare the two versions becomes useless due to all the extra noise.



```
      "outputs": [
       {
        "data": {
-        "image/png":
"iVBORw0KGgoAAAANSUhEUgAAAXYAAAD/CAYAAADllv3BAAAABHNCSVQICAgIfAhkiAAAAAlwSF
lz\nAAALEgAACxIB0t1+/AAAIABJREFUeJzt3XlUFGe6P/BvASIgKDpHZ4JxQRLGCTJGMMEljKI
SnaBG\nwE5wQaMTl8TrGFACRNlDUMHlGsSIGoyaEReIcZh71RjUmaCiQiCgGZwfIiqYG0lAGhob
kPf3B3ZL\ny9IL1V3VzfM5x3Okmq7+dlk+vLxd9bwcY4yBEEKIyTATOgAhhBB+UWEnhBATQ4WdE
EJMDBV2Qggx\nMVTYCSHExFBhJ4QQE6NRYS8sLERgYGC77dnZ2Zg7dy4CAgJw7Ngx3sMRQgjRno
W6b9i7dy++/vpr\n9OnTR2V7c3MzNm7ciMzMTPTu3Rvz5s3D1KlTMWDAAL2FJYQQop7aEfuwYcO
wc+fOdttLS0sxbNgw\n2NraolevXnB3d8fVq1f1EpIQQojm1BZ2b29vmJubt9teV1cHOzs75dd9
+vSBVCrlNx0hhBCt6fzh\nqa2tLerq6pRf19fXo2/fvryEIoQQoju1c+wKz7aUcXJyQnl5Oe1Wpra
2FlZYWrV6/iL3/5S4fPzcvL\n615KQgjpodzd3bV+jtaFncPhAACys7Px8NEBWVhYaGhogkUgQHh6OpUuXgj
EGiUSCQYMG8RpOnyorK+Hg4CB0\njHbEmIsyaYYyaU6MucSYSddBsUaFffDgwUhPTwcAzJw5U7l
98uTJmDx5sk4vTAghRD/oBiVCCDEx\nVNgJIcTEaDzHTkxHWVk5IiL2o6KiBYMHmyEu7h04Og4T
OhYhhCdU2DvR2NiIII9z76CLs++QSWlpZC\nx+FNWVk5vL0/RWlpDIA+AOpx+XIwcsNRV3QkwET
cV0YllEBL4YPBjLIyOFjsKriIj9bYo6APRB\naWkMIiL2C5iKEMInKuwd+PzoUXxtb4/HY8bgRN
++SDOhBmcVFS14WpN7cQl52Nh+PAHwAeTpiA2G+/RemtWwIn48f
gwWYA6p9uGPcL0OchHByengo1TU34xy+/GD4/xAQX\nVNif8cGmTbj95psPno0PNm0SKBG/
4uLegYNjMEquG8bTKFqUKVuT31neDGs/2NVMPt7aH\nRYrnNDwcm8RSAxHH8/GvaEnw
QrDMTcy5yBm33cUaXNDA+8fmpYKhpGzzhhGbDx/uMMmj5Dc8hp/\n0aLC/oztoaEYe4vQ4AoehJ
```

```
      "outputs": [
       {
        "data": {
+        "image/png":
"iVBORw0KGgoAAAANSUhEUgAAAW4AAAD1CAYAAABwdB+7AAAABHNCSVQICAgIfAhkiAAAAAlwSF
lzAAALEgAACxIB0t1+/AAAADl0RVh0U29mdHdhcmUAbWF0cGxvdGxpYiB2ZXJzaW9uIDIuMS4yL
CBodHRwOi8vbWF0cGxvdGxpYi5vcmcvNQv5yAAAIABJREFUeJzt3Xl8VPW9//HXJJMNSEg0QZMU
ZFG/pAYVIxBEQRQXKlQMVIv9RbBYpe5wFVBcan91K1exYq1tEeG2WhsxUgEvVAKi0pJgCja5hq9
gLgSIkCBLQsw2k7l/TGachMk6Z2bOJJ/n4+FD5pyZc96T5ZPvfM85n2NxOBwIIYQIHWHBDiCEEK
JrpHALIUSIkcIthBAhRgq3EEKGCncQggRYqRwCyFEiOlU4VZKjVFKfeRl+VSl1A6l1D+VUj8zP
J0QQojTWDo6j1sptQDIBmq01pkeyyyOAEmAUANsA6ZqrQ+33bkZhYaGcGLC6EEF2UkZFh8bbc2onX
fgVkAX9qtTwN2Ku1Pg6glPoUAJ4p40AnQ4bSCUlJaSlpQU7hldmzgbmzmfmbGDufJKt+4zMV1h
Y2Oa6Dgu31vpdpdB/m1B1vpdpdRgL6vigJMej6uBn6B1w5WB3PsnWfYHkVdXZ1k6yYYz5zNjB3PsnWfYHK15kRd1
uqgFiPx7HAibaebNa/kmb+C27mbGDufGbOBubOJ9m6zzQj7naUAucqpeKr0ac4AaTgHjgf/0YXtCCCE6o
cuFWyl1E/ORqcPGPR8YBegc1Nwak0PGR1QCCFES50q3FrrH9B6a5u9e02/r8ekm+DSdzrDxdUTRf8Zq3Bm+Fljrl2RCCCE6
RPQga3YeYslGTZRqG+l7/5nrH38fDgocfZ8+Xxfjlo+0RSqWlKEmIoZSaEKXcC+7eOrDGtfIbzoCuAkpmUxDi
06SIlET7Rm5yHGPbeZIYvWM+65zaBIwbN86bwZXl5bFo0aIl212dnZB+/zfqRwd+Dh1f/m6
KkGHA44qqeBav/HexIhis/Udul5UJ0VW/5VJeQkMCAAQPaXJ+YmNf5ZJ3tu50SW/pSqO...
TQDU25rIK6kgZ8cBbh41MMjpjJMSH8OhVkV6xp4tHPveuS2W1WzPp664iDPvuCOQ8UQP0N6nOl+
```

Source: ReviewNB Blog

It becomes even more confusing if you have changed some code after the outputs were generated. Now the code and outputs that are stored in the version control do not match anymore.

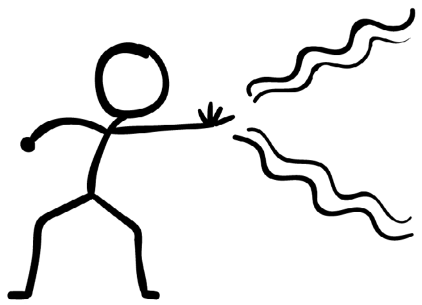
There are two options at your disposal.

You can manually clear the outputs from the main menu (Cells -> All Output -> Clear) before creating your git commit.

You can set up a pre-commit hook for git that clears outputs automatically

We highly recommend investing to option #2 as manual steps that you need to remember are destined to fail eventually.

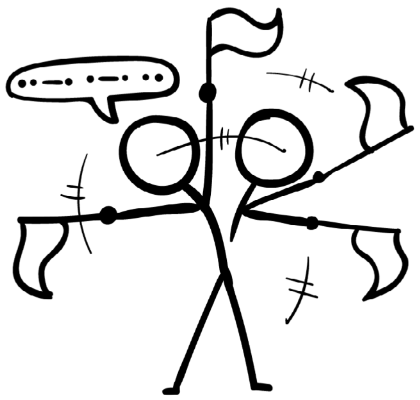
Don't use the --force



Sometimes when you try to push to the remote repository, git tells you that something is wrong and aborts. The error message might offer you an option to "use the force" (the `-f` or `--force`). Don't do it! Even if the error message calls for your inner Jedi, just don't. It's the dark side.

Obviously, there are reasons why the `--force` exists and it serves a purpose in some situations. None of those arguments apply to you young padawan. Whatever the case, read the error message, try to reason what could be the issue, ask someone else to help you if needed, and get the underlying issue fixed.

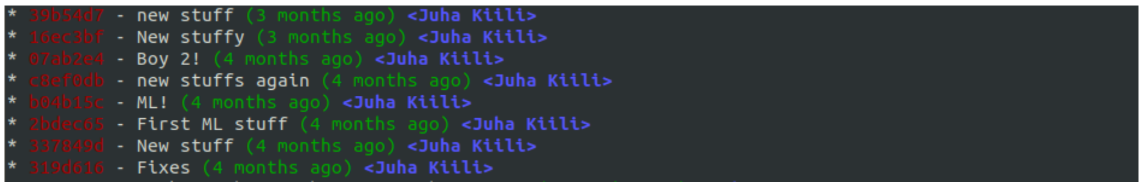
Do small commits with clear descriptions



Inexperienced users often fall into the trap of making huge commits with nonsensical descriptions. A good rule of thumb for any single git commit is that it should only do one thing. Fix one bug, not three. Solve one issue, not twelve. Remember that issues can often be split into smaller chunks, too. The smaller you can make it, the better.

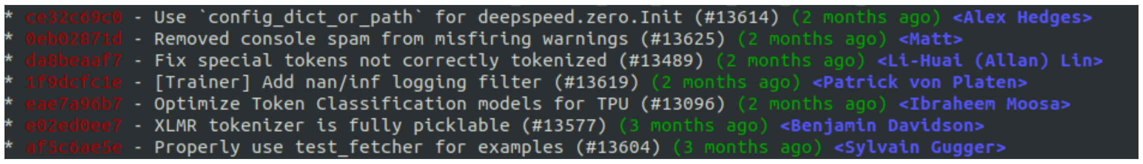
The reason you use version control is that someone else can understand what has happened in the past. If your commit fixes twelve bugs and the description says "Model fixed", it is close to zero value two months later. The commit should only do one thing and one thing only. The description should communicate the thing was. You don't need to make the descriptions long-winded novels if the commits are small. In fact, a long description for a commit message implies that the commit is too big and you should split it into smaller chunks!

Example #1: a bad repository



```
* 39b54d7 - new stuff (3 months ago) <Juha Kiili>
* 16ec3bf - New stuffy (3 months ago) <Juha Kiili>
* 07ab2e4 - Boy 2! (4 months ago) <Juha Kiili>
* c8ef0db - new stuffs again (4 months ago) <Juha Kiili>
* b04b15c - ML! (4 months ago) <Juha Kiili>
* 2bdec65 - First ML stuff (4 months ago) <Juha Kiili>
* 337849d - New stuff (4 months ago) <Juha Kiili>
* 319d616 - Fixes (4 months ago) <Juha Kiili>
```

Example #2: a good repository



```
* ce32c69c0 - Use `config_dict_or_path` for deepspeed.zero.Init (#13614) (2 months ago) <Alex Hedges>
* 0eb02871d - Removed console spam from misfiring warnings (#13625) (2 months ago) <Matt>
* da8beaaf7 - Fix special tokens not correctly tokenized (#13489) (2 months ago) <Li-Huai (Allan) Lin>
* 1f9dcfc1e - [Trainer] Add nan/inf logging filter (#13619) (2 months ago) <Patrick von Platen>
* eae7a96b7 - Optimize Token Classification models for TPU (#13096) (2 months ago) <Ibraheem Moosa>
* e02ed0ee7 - XLMR tokenizer is fully picklable (#13577) (3 months ago) <Benjamin Davidson>
* af5c6ae5e - Properly use test_fetcher for examples (#13604) (3 months ago) <Sylvain Gugger>
```

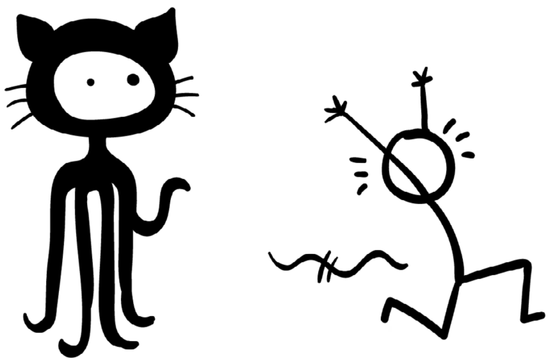
In real life you often make all kinds of ad-hoc things and end up in the situation #1 on your local machine. If you haven't pushed anything to the public remote yet, you can still fix the situation. We recommend learning how to use the interactive rebase.

Simply use:

```
git rebase -i origin/main
```

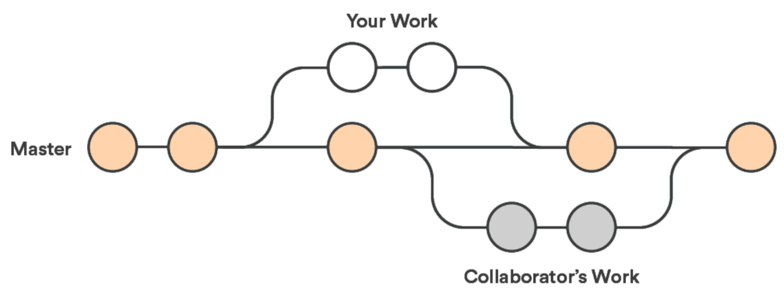
The interactive mode offers many different options for tweaking the history, rewording commit messages, and even changing the order. Learn more about the interactive rebase from here.

Don't be afraid of branching & pull requests

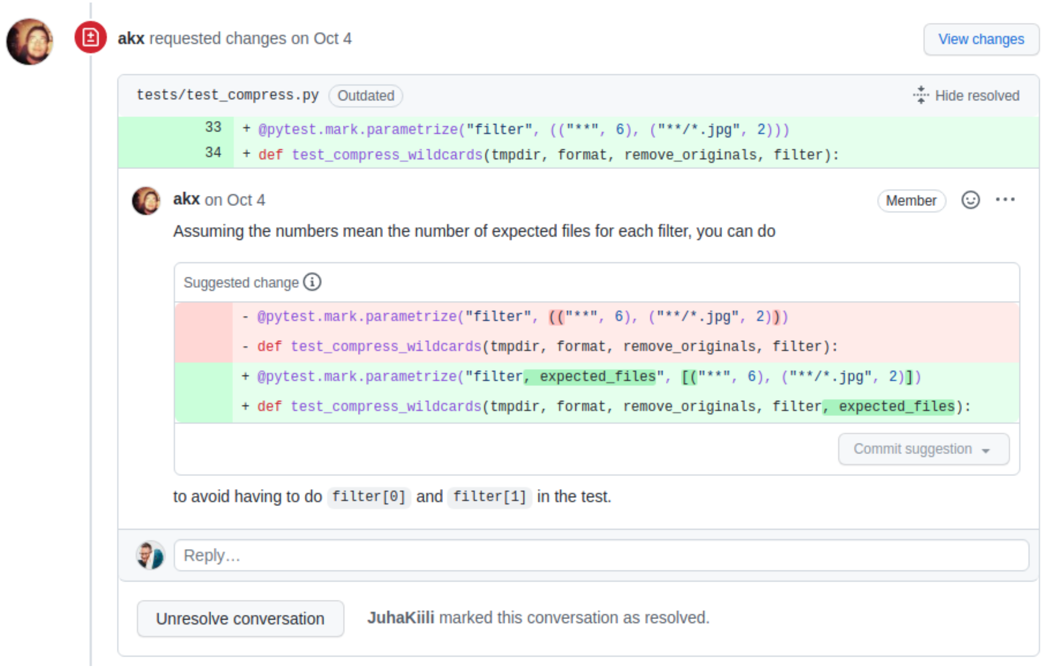


Branching and especially pull requests are slightly more advanced and not everyone's cup of tea, but if your data science project is mature, in production, and constantly touched by many different people, pull requests may be just the thing that is missing from your process.

When you create a new git repository, it will start with just a single branch called main (or master). The main branch is considered as the "central truth". Branching means that you will branch out temporarily to create a new feature or a fix to an old one. In the meantime, someone else can work in parallel on their own branch. This is commonly referred to as feature branch workflow.



The idea with branches is to eventually merge back to the `main` branch and update "the central truth". This is where pull requests come into play. The rest of the world doesn't care about your commits in your own branch, but merging to `main` is where your branch becomes the latest truth. That is when it's time to make a pull request.


akx requested changes on Oct 4

tests/test_compress.py Outdated

```
+ @pytest.mark.parametrize("filter", (("**", 6), ("**/*.jpg", 2)))
+ def test_compress_wildcards(tmpdir, format, remove_originals, filter):
```

akx on Oct 4

Assuming the numbers mean the number of expected files for each filter, you can do

Suggested change

```
- @pytest.mark.parametrize("filter", (("**", 6), ("**/*.jpg", 2)))
- def test_compress_wildcards(tmpdir, format, remove_originals, filter):
+ @pytest.mark.parametrize("filter, expected_files", [("**", 6), ("**/*.jpg", 2)])
+ def test_compress_wildcards(tmpdir, format, remove_originals, filter, expected_files):
```

to avoid having to do `filter[0]` and `filter[1]` in the test.

JuhaKiili marked this conversation as resolved.

Pull requests are not a git concept, but a GitHub concept. They are a request for making your branch the new central truth. Using the pull request, other users will check your changes before they are allowed to become the new central truth. GitHub offers great tools to make comments, suggest their modifications, signal approval, and finally apply the merge automatically.

Python dependencies

What is dependency management anyway?

Dependency management is the act of managing all the external pieces that your project relies on. It has the risk profile of a sewage system. When it works, you don't even know it's there, but it becomes excruciating and almost impossible to ignore when it fails.

Every project is built on someone else's sweat and tears. Those days when an engineer woke up, made coffee, and started a new project by writing a bootloader – the program that boots up your computer from scratch – are history. There are massive stacks of software and libraries beneath us. We are simply sprinkling our own thin layer of sugar on top.

My computer has a different stack of software than yours. Not only are the stacks different, but they are forever changing. It is amazing how *anything* works, but it does. All thanks to the sewage system of dependency management and a lot of smart people abstracting the layers so that we can just call our favorite pandas function and get predictable results.

Basics of Python dependency management

Let's make one thing clear. Simply Installing and upgrading Python packages is not dependency management. Dependency management is documenting the required environment for your project and making it easy and deterministic for others to reproduce it.

You could write installation instructions on a piece of paper. You could write them in your source code comments. You could even hardcode the install commands straight into the program. Dependency management? Yes. Recommended? Nope.

The recommended way is to decouple the dependency information from the code in a standardized, reproducible, widely-accepted format. This allows version pinning and easy deterministic installation. There are many options, but we'll describe the classic combination of pip and requirements.txt file in this article.

But before we go there, let's first introduce the atomic unit of Python dependency: the package.

What is a package?

"Package" is a well-defined term in Python. Terms like library, framework, toolkit are not. We will use the term "package" for the remainder of this article, even for the things that some refer to as libraries, frameworks, or toolkits.

A module is everything defined in a single Python file (classes, functions, etc.).

A package is a collection of modules.

Pandas is a package, Matplotlib is a package, print()-function is not a package. The purpose of a package is to be an easily distributable, reusable, and versioned collection of modules with well-defined dependencies to other packages.

You are probably working with packages every day by referring to them in your code with the Python `import` statement.

The art of installing packages

While you could install packages by simply downloading them manually to your project, the most common way to install a package is via PyPi (Python Package Index) using the famous `pip install` command.

Note: Never use `sudo pip install`. Never. It is like running a virus. The results are unpredictable and will cause your future self major pain.

Never install Python packages globally either. Always use virtual environments.

What are virtual environments?

Python virtual environment is a safe bubble. You should create a protective bubble around all the projects on your local computer. If you don't, the projects will hurt each other. Don't let the sewage system leak!

If you call `pip install pandas` outside the bubble, it will be installed globally. This is bad. The world moves forward and so do packages. One project needs the Matplotlib of 2019 and the other wants the 2021 version. A single global installation can't serve both projects. So protective bubbles are a necessity. Let's look at how to use them.

Go to your project root directory and create a virtual environment:

```
python3 -m venv mybubble
```

Now we have a bubble, but we are not in it yet. Let's go in!

```
source mybubble/bin/activate
```

Now we are in the bubble. Your terminal should show the virtual environment name in parenthesis like this:

```
(mybubble) johndoe@hello:~/myproject$
```

Now that we are in the bubble, installing packages is safe. From now on, any pip install command will only have effects inside the virtual environment. Any code you run will only use the packages inside the bubble.

If you list the installed packages you should see a very short list of currently installed default packages (like the pip itself).

```
pip list

Package       Version
------------- -------
pip           20.0.2
pkg-resources 0.0.0
setuptools    44.0.0
```

This listing is no longer for all the Python packages in your machine, but all the Python packages inside your virtual environment. Also, note that the Python version used inside the bubble is the Python version you used to create the bubble.

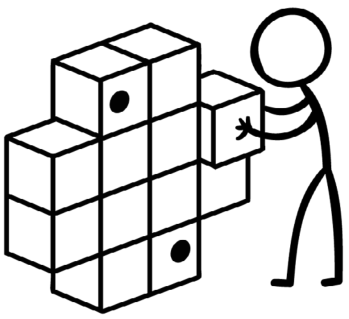
To leave the bubble, simply call `deactivate` command.

Always create virtual environments for all your local projects and run your code inside those bubble(s). The pain from conflicting package versions between projects is the kind of pain that makes people quit their jobs. Don't be one of those people.

What is version pinning?

Imagine you have a project that depends on Pandas package and you want to communicate that to the rest of the world (and your future self). Should be easy, right?

First of all, it is risky to just say: "You need Pandas".



The less risky option is "You need Pandas 1.2.1", but even that is not always enough.

Let's say you are correctly pinning the Pandas version to 1.2.1. Pandas itself has a dependency for numpy, but unfortunately doesn't pin the dependency to an exact numpy version. Pandas itself just says "You need numpy" and does not pin to an exact version.

At first, everything is fine, but after six months, a new numpy version 1.19.6 is released with a showstopper bug.

Now if someone installs your project, they'll get pandas 1.2.1 with buggy numpy 1.19.6, and probably a few gray hairs as your software spits weird errors. The sewage system is leaking. The installation process was not deterministic!

The most reliable way is to pin everything. Pin the dependencies of the dependencies of the dependencies of the dependencies, of the... You'll get the point. Pin'em as deep as the rabbit hole goes. Luckily there are tools that make this happen for you.

Note: If you are building a reusable package and not a typical project, you should not pin it so aggressively (this is why Pandas doesn't pin to the exact Numpy version). It is considered best practice for the end-user of the package to decide what and how aggressively pin. If you as a package creator pin everything, then you close that door from the end-user.

How do I pin Python dependencies?

Whenever you call `pip install` to get some hot new package into your project, you should stop and think for a second. This will create a new dependency for your project. How do I document this?

You should write down new libraries and their version number to a requirements.txt file. It is a format understood by pip to install multiple packages in one go.

# requirements.txt

```
pandas==1.2.1
matplotlib==3.4.0
```

# Install

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

This is already much better than most data science projects that one encounters, but we can still do better. Remember the recursive dependency rabbit hole from the previous chapter about version pinning. How do we make the installation more deterministic?

The answer is `pip-compile` command and `requirements.in` text file.

# Requirements.in

```
matplotlib==3.4.0
```

# Auto-generate requirements.txt

```
pip-compile requirements.in
```

# Generated requirements.txt

```
cycler==0.11.0
    # via matplotlib
kiwisolver==1.3.2
    # via matplotlib
matplotlib==3.4.0
    # via -r requirements.in
numpy==1.22.0
    # via
    #   matplotlib
    #   pandas
pandas==1.2.1
    # via -r requirements.in
pillow==9.0.0
    # via matplotlib
pyparsing==3.0.6
    # via matplotlib
```

```
python-dateutil==2.8.2
    # via
    #   matplotlib
    #   pandas
pytz==2021.3
    # via pandas
six==1.16.0
    # via python-dateutil
```

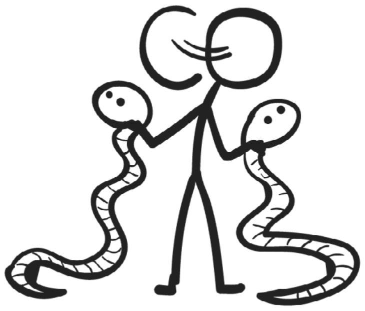
In the requirements.in you should only put your direct dependencies.

The pip-compile will then generate the perfect pinning of all the libraries into the requirements.txt, which provides all the information for a deterministic installation. Easy peasy! Remember to commit both files into your git repository, too.

How to pin the Python version?

Pinning the Python version is tricky. There is no straightforward way to pin the version dependency for Python itself (without using e.g conda).

You could make a Python package out of your project, which lets you define the Python version in the `setup.py` or `setup.cfg` with the key `python_requires>=3.9`, but that is overkill for a typical data science project, which usually doesn't have the characteristics of a reusable package anyway.



If you are really serious about pinning to specific Python, you could also do something like this in your code:

```
import sys
if sys.version_info < (3,9):
    sys.exit("Python >= 3.9 
required.")
```

The most bullet-proof way to force the Python version is to use Docker containers, which we will talk about in the next chapter!

Main takeaways

Don't avoid dependency management – Your future self will appreciate the documented dependencies when you pour coffee all over your MacBook.

Always use virtual environments on your local computer – Trying out that esoteric Python library with 2 GitHub stars is no big deal when you are safely inside the protective bubble.

Pinning versions is better than not pinning – Version pinning protects from packages moving forward when your project is not.

Packages change a lot, Python not so much – Even a single package can have dozens of nested dependencies and they are constantly changing, but Python is relatively stable and future-proof.

What about the cloud?

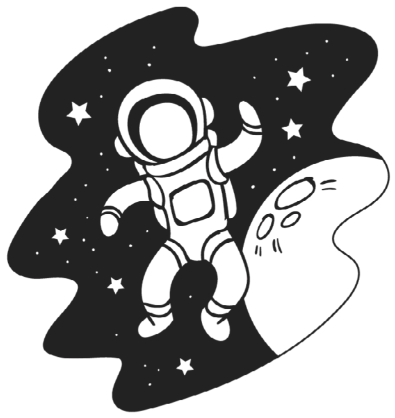
When your project matures enough and elevates into the cloud and into production, you should look into pinning the entire environment and not just the Python stuff.

This is where Docker containers are your best friend as they not only let you pin the Python version but anything inside the operating system. It is like a virtual environment but on a bigger scale.

Docker

What is Docker?

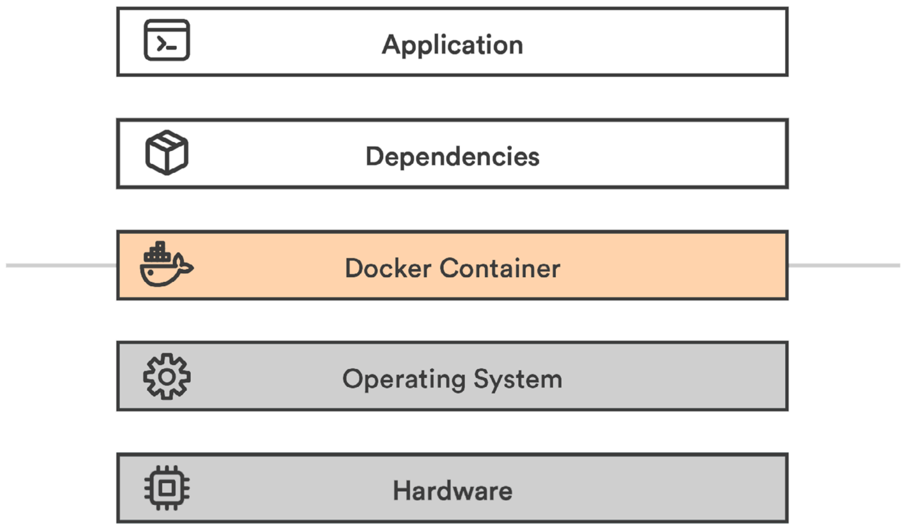
Imagine being an astronaut on a space station and planning to go outside and enjoy the view. You'd be facing hostile conditions. The temperature, oxygen, and radiation are not what you were built for. Human beings require a specific environment to thrive. To properly function in any other scenario, like deep in the sea or high up in space, we need a system to reproduce that environment. Whether it is a spacesuit or a submarine, we need isolation and something that ensures the levels of oxygen, pressure, and temperature we depend on.



In other words, we need a container.

Any software faces the same problem as the astronaut. As soon as we leave home and go out into the world, the environment gets hostile, and a protective mechanism to reproduce our natural environment is mandatory. The Docker container is the spacesuit of programs.

Docker isolates the software from all other things on the same system. A program running inside a "spacesuit" generally has no idea it is wearing one and is unaffected by anything happening outside.



The containerized stack

Application: High-level application (your data science project)

Dependencies: Low-level generic software (think Tensorflow or Python)

Docker container: The isolation layer

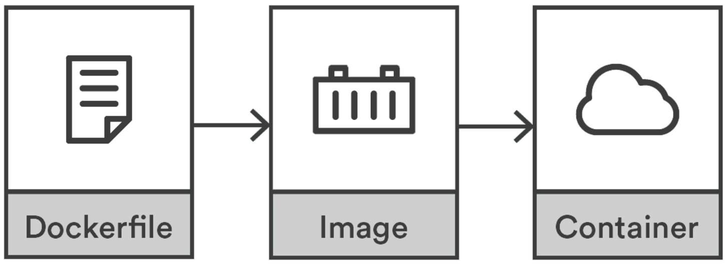
Operating system: Low-level interfaces and drivers to interact with the hardware

Hardware: CPU, Memory, Hard disk, Network, etc.

The fundamental idea is to package an application and its dependencies into a single reusable artifact, which can be instantiated reliably in different environments.

How to create a container?

The flow to create Docker containers:



1. Dockerfile: Instructions for compiling an image

2. Image: Compiled artifact

3. Container: An executed instance of the image

Dockerfile

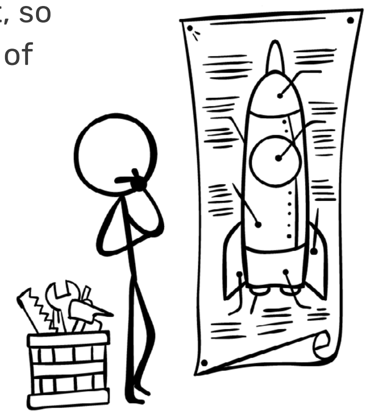
First, we need instructions.

We could define the temperature, radiation, and oxygen levels for a spacesuit, but we need instructions, not requirements. Docker is instruction-based, not requirement-based. We will describe the *how* and not the *what.* To do that, we create a text file and name it `Dockerfile`.

# Dockerfile

```
FROM python:3.9
RUN pip install tensorflow==2.7.0
RUN pip install pandas==1.3.3
```

The `FROM` command describes a base environment, so we don't need to start from scratch. A treasure trove of base images can be found from the DockerHub or via google searches.



The `RUN` command is an instruction to change the environment.

Note: While our example installs Python libraries one by one, that is not recommended. The best practice is to utilize `requirements.txt`, which defines the Python dependencies. Follow the best practices from our previous chapter to create one.

# Dockerfile with requirements.txt

```
FROM python:3.9
COPY requirements.txt /tmp
RUN pip install -r /tmp/requirements.txt
```

The COPY command copies a file from your local disk, like the `requirements.txt`, into the image. The `RUN` command here installs all the Python dependencies defined in the `requirements.txt` in one go.

Note: All the familiar Linux commands are at your disposal when using RUN.

Docker image

Now that we have our `Dockerfile`, we can compile it into a binary artifact called an image.

The reason for this step is to make it faster and reproducible. If we didn't compile it, everyone needing a spacesuit would need to find a sewing machine and painstakingly run all the instructions for every spacewalk. That is too slow but also indeterministic. Your sewing machine might be different from mine.

The tradeoff for speed and quality is that images can be quite large, often gigabytes, but a gigabyte in 2022 is peanuts anyway.

To compile, use the build command:

```
docker build . -t myimage:1.0
```

This builds an image stored on your local machine. The `-t` parameter defines the image name as “myimage” and gives it a tag “1.0”. To list all the images, run:

This builds an image stored on your local machine. The -t parameter defines the image name as “myimage” and gives it a tag “1.0”. To list all the images, run:

```
docker image list

REPOSITORY   TAG     IMAGE ID      CREATED       SIZE
<none>       <none>  85eb1ea6d4be  6 days ago    2.9GB
myimagename  1.0     ff732d925c6e  6 days ago    2.9GB
myimagename  1.1     ff732d925c6e  6 days ago    2.9GB
myimagename  latest  ff732d925c6e  6 days ago    2.9GB
python       3.9     f88f0508dc46  13 days ago   912MB
```

Docker container

Finally, we are ready for our spacewalk. Containers are the real-life instances of a spacesuit. They are not really helpful in the wardrobe, so the astronaut should perform a task or two while wearing them.

The instructions can be baked into the image or provided just in time before starting the container. Let’s do the latter.

```
docker run myimagename:1.0 echo "Hello world"
```

This starts the container, runs a single `echo` command, and closes it down.

Now we have a reproducible method to execute our code in any environment that supports Docker. This is very important in data science, where each project has many dependencies, and reproducibility is at the heart of the process.

Containers close down automatically when they have executed their instructions, but containers can run for a long time. Try starting a very long command in the background (using your shell’s & operator):

```
docker run myimagename:1.0 sleep 100000000000 &
```

You can see our currently running container with:

```
docker container list
```

To stop this container, take the container ID from the table and call:

```
docker stop <CONTAINER ID>
```

This stops the container, but its state is kept around. If you call

```
docker ps -a
```

You can see that the container is stopped but still exists. To completely destroy it:

```
docker rm  <CONTAINER ID>
```

The single command combining both stopping and removing:

```
docker rm -f <CONTAINER_ID>
```

To remove all stopped leftover containers:

```
docker container prune
```

Tip: You can also start a container with an interactive shell:

```
$ docker run -it myimagename:1.0 /bin/bash
root@9c4060d0136e:/# echo "hello"
hello
root@9c4060d0136e:/# exit
exit
$ <back in the host shell>
```

It is great for debugging the inner workings of an image when you can freely run all the Linux commands interactively. Go back to your host shell by running the `exit` command.

Terminology and Naming

Registry = Service for hosting and distributing images. The default registry is the Docker Hub.

Repository = Collection of related images with the same name but different tags. Usually, different versions of the same application or service.

Tag = An identifier attached to images within a repository (e.g., 14.04 or stable)

ImageID = Unique identifier hash generated for each image

The official documentation declares:

An image name is made up of slash-separated name components, optionally prefixed by a registry hostname.

It means that you can encode registry hostname and a bunch of slash-separated "name components" into the name of your image. Honestly, this is quite convoluted, but such is life.

The fundamental format is:

```
<name>:<tag>
```

But in practice it is:

```
<registry>/<name-component-1>/
<name-component-2>:<tag>
```

It may vary per platform. For Google Cloud Platform (GCP) the convention is:

```
<registry>/<project-id>/
<repository-name>/<image>@<image-digest>:<tag>
```

It is up to you to figure out the correct naming scheme for your case.

Note: The `latest` tag will be used if you pull an image without any tags. Never use this `latest` tag in production. Always use a tag with a unique version or hash instead since someone inevitably will update the "latest" image and break your build. What is the latest today is no longer the latest tomorrow! The astronaut doesn't care about the latest bells and whistles. They just want a spacesuit that fits them and keeps them alive.

Docker images and secrets

Just like it is a terrible practice to push secrets into a git repository, you shouldn't bake them into your Docker images either!

Images are put into repositories and passed around carelessly. The correct assumption is that whatever goes into an image may be public at some point. It is not a place for your username, password, API token, key code, TLS certificates, or any other sensitive data.

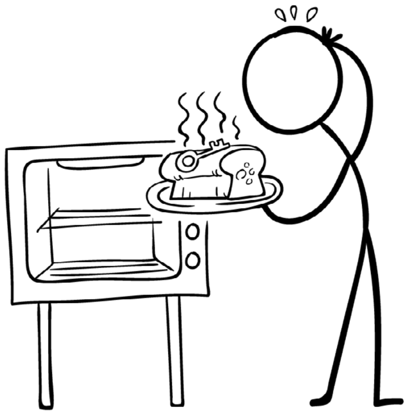
There are two scenarios with secrets and docker images:

1. You need a secret at build-time
2. You need a secret at runtime

Neither case should be solved by baking things permanently into the image. Let's look at how to do it differently.

Build-time secrets

If you need something private – say a private GitHub repository – to be pulled into the image at build time, you need to make sure that the SSH keys you are using do not leak into the image.



Do NOT use COPY instruction to move keys or passwords into the image! Even if you remove them afterward, they will still leave a trace!

Quick googling will give you many different options to solve this problem, like using multi-stage builds, but the best and most modern way is to use BuildKit. BuildKit ships with Docker but needs to be enabled for builds by setting up the environment variable `DOCKER_BUILDKIT`.

For example:

```
DOCKER_BUILDKIT=1 docker build .
```

BuildKit offers a mechanism to make secret files safely available for the build process.

Let's first create `secret.txt` with the contents:

```
TOP SECRET ASTRONAUT PASSWORD
```

Then create a new `Dockerfile`:

```
FROM alpine

RUN --mount=type=secret,
id=mypass cat /run/secrets/mypass
```

`The --mount=type=secret,id=mypass` is informing Docker that for this specific command, we need access to a secret called `mypass` (the contents of which we'll tell the Docker build about in the next step). Docker will make this happen by temporarily mounting a file `/run/secrets/mypass`.

The `cat /run/secrets/mypass` is the actual instruction, where `cat` is a Linux command to output the contents of a file into the terminal. We call it to validate that our secret was indeed available.

Let's build the image, adding `--secret` to inform `docker build` about where to find this secret:

```
DOCKER_BUILDKIT=1 docker build . -t myimage --secret
id=mypass,src=secret.txt
```

Everything worked, but we didn't see the contents of secret.txt printed out in our terminal as we expected. The reason is that BuildKit doesn't log every success by default.

Let's build the image using additional parameters. We add `BUILDKIT_PROGRESS=plain` to get more verbose logging and `--no-cache` to make sure caching doesn't ruin it:

```
DOCKER_BUILDKIT=1 BUILDKIT_PROGRESS=plain docker build .
--no-cache --secret id=mypass,src=secret.txt
```

Among all the logs printed out, you should find this part:

```
#5 [2/2] RUN --mount=type=secret,id=mypass cat /run/secrets/
mypass
#5 sha256:7fd248d616c172325af799b6570d2522d3923638ca41181fa
b438c29d0aea143
#5 0.248 TOP SECRET ASTRONAUT PASSWORD
```

It is proof that the build step had access to `secret.txt`.

With this approach, you can now safely mount secrets to the build process without worrying about leaking keys or passwords to the resulting image.

Runtime secrets

If you need a secret – say database credentials – when your container is running in production, you should use environment variables to pass secrets into the container.

Never bake any secrets straight into the image at build time!

```
docker run --env MYLOGIN=johndoe --env
MYPASSWORD=sdf4otwe3789
```

These will be accessible in Python like:

```
os.environ.get('MYLOGIN')
os.environ.get('MYPASSWORD')
```

Tip: You can also fetch the secrets from a secret store like Hashicorp Vault!

GPU support

Docker with GPUs can be tricky. Building an image from scratch is beyond the scope of this article, but there are five prerequisites for a modern GPU (NVIDIA) container.

Image:

- CUDA/cuDNN libraries
- GPU versions of your framework like Tensorflow (when needed)

Host machine:

- GPU drivers

- NVidia Docker Toolkit
- Docker `run` executed with `--gpus all`

The best approach is finding a base image with most prerequisites already baked in. Frameworks like Tensorflow usually offer images like `tensorflow/tensorflow:latest-gpu`, which are a good starting point.

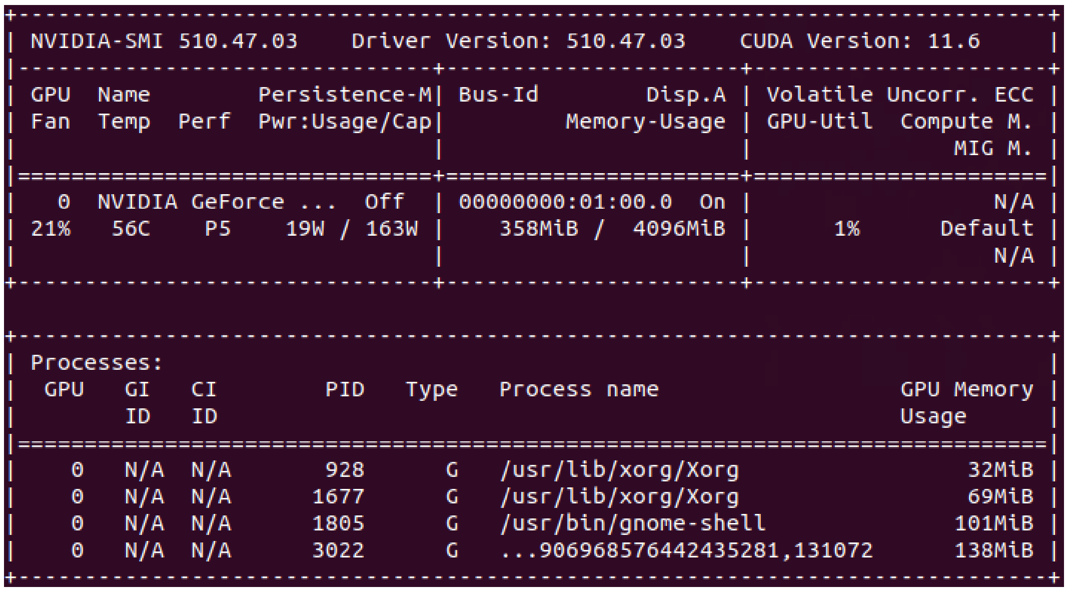
When troubleshooting, you can first try to test your host machine:

```
nvidia-smi
```

Then run the same command inside the container:

```
docker run --gpus all tensorflow/tensorflow:latest-gpu
nvidia-smi
```

You should get something like this for both commands:



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:01:00.0  On |                  N/A |
| 21%   56C    P5    19W / 163W |    358MiB /  4096MiB |      1%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A       928      G   /usr/lib/xorg/Xorg                 32MiB |
|    0   N/A  N/A      1677      G   /usr/lib/xorg/Xorg                 69MiB |
|    0   N/A  N/A      1805      G   /usr/bin/gnome-shell              101MiB |
|    0   N/A  N/A      3022      G   ...906968576442435281,131072      138MiB |
+-----------------------------------------------------------------------------+
```

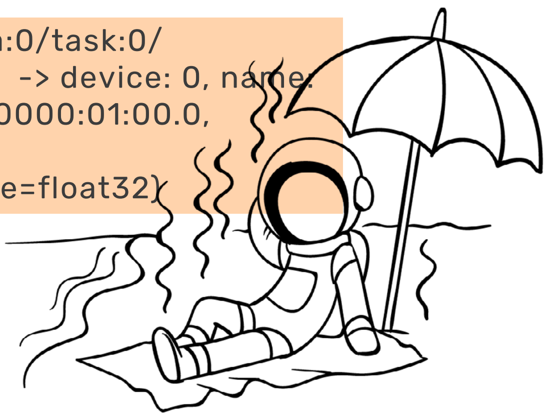
If you get an error from either, you'll have an idea whether the problem lies inside or outside the container.

It's also a good idea to test your frameworks. For example Tensorflow:

```
docker run --gpus all -it --rm tensorflow/
tensorflow:latest-gpu python -c "import
tensorflow as tf;print(tf.reduce_sum
(tf.random.normal([1000, 1000])))"
```

The output may be verbose and have some warnings, but it should end with something like:

```
Created device /job:localhost/replica:0/task:0/
device:GPU:0 with 3006 MB memory:  -> device: 0, name:
NVIDIA GeForce GTX 970, pci bus id: 0000:01:00.0,
compute capability: 5.2
tf.Tensor(-237.35098, shape=(), dtype=float32)
```



Docker containers vs. Python virtual environments

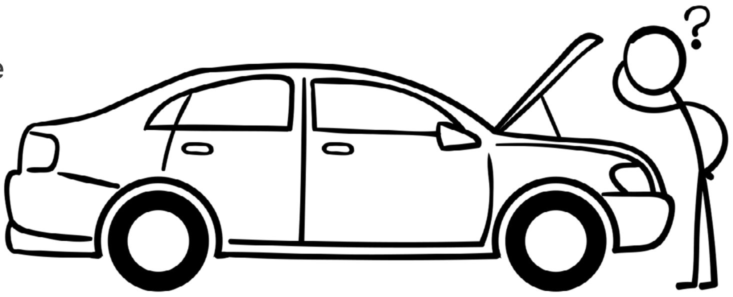
Our last chapter about Python dependency management talked about Python virtual environments and how they create a safety bubble between different Python projects in your local development environment. Docker containers solve a similar problem but on a different layer.

While a Python virtual environment creates the isolation layer between all Python-related things, a Docker container achieves this for the entire software stack. The use-cases for Python virtual environments and Docker containers are different. As a rule of thumb, virtual environments are enough for developing things on your local machine while Docker containers are built for running production jobs in the cloud.

To put it another way, for local development virtual environments are like wearing sunscreen on the beach, while Docker containers are like wearing a spacesuit – usually uncomfortable and mostly impractical.

What every data scientist should know about the command line

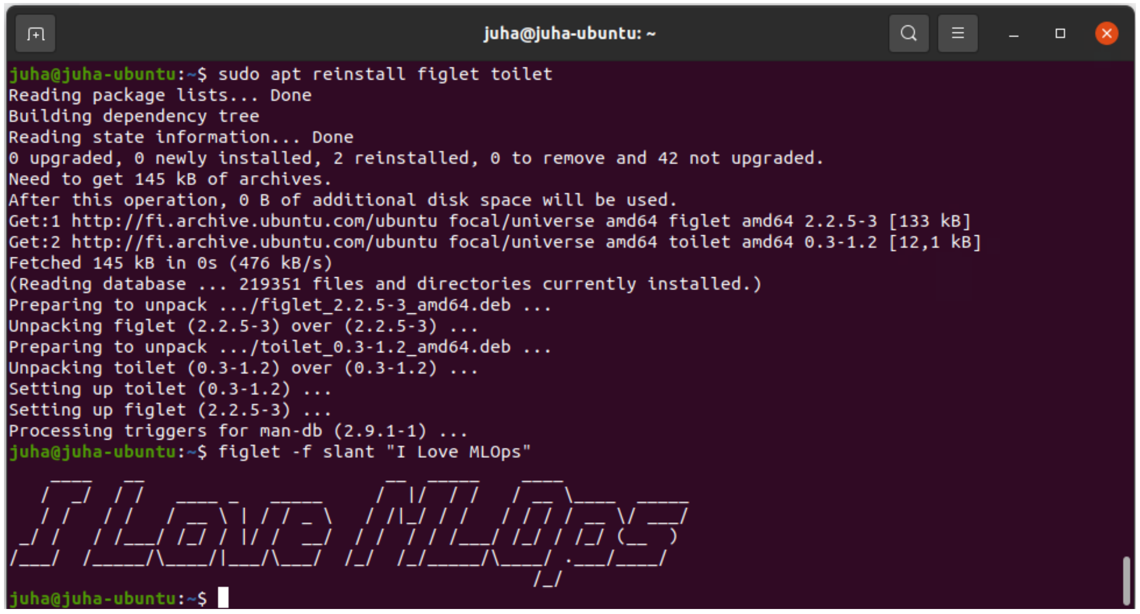
It is now 50 years old, and we still can't figure out what to call it. Command line, shell, terminal, bash, prompt, or console? We shall refer to it as the command line to keep things consistent.


Command line gives you sneak peek under the hood

The article will focus on the UNIX-style (Linux & Mac) command line and ignore the rest (like Windows's command processor and PowerShell) for clarity. We have observed that most data scientists are on UNIX-based systems these days.

What is it?

The command line is a text-based interface to your computer. You can think of it kind of as "popping the hood" of an operating system. Some people mistake it as just a relic of the past but don't be fooled. The modern command line is rocking like never before!



```
juha@juha-ubuntu:~$ sudo apt reinstall figlet toilet
Reading package lists... Done
Building dependency tree
Reading state information... Done
0 upgraded, 0 newly installed, 2 reinstalled, 0 to remove and 42 not upgraded.
Need to get 145 kB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 http://fi.archive.ubuntu.com/ubuntu focal/universe amd64 figlet amd64 2.2.5-3 [133 kB]
Get:2 http://fi.archive.ubuntu.com/ubuntu focal/universe amd64 toilet amd64 0.3-1.2 [12,1 kB]
Fetched 145 kB in 0s (476 kB/s)
(Reading database ... 219351 files and directories currently installed.)
Preparing to unpack .../figlet_2.2.5-3_amd64.deb ...
Unpacking figlet (2.2.5-3) over (2.2.5-3) ...
Preparing to unpack .../toilet_0.3-1.2_amd64.deb ...
Unpacking toilet (0.3-1.2) over (0.3-1.2) ...
Setting up toilet (0.3-1.2) ...
Setting up figlet (2.2.5-3) ...
Processing triggers for man-db (2.9.1-1) ...
juha@juha-ubuntu:~$ figlet -f slant "I Love MLOps"
    ____   __                       __  _____    ____
   /  _/  / /   ____  _   _____    /  |/  / /   / __ \____  _____
   / /   / /   / __ \| | / / _ \  / /|_/ / /   / / / / __ \/ ___/
 _/ /   / /___/ /_/ /| |/ /  __/ / /  / / /___/ /_/ / /_/ (__  )
/___/  /_____/\____/ |___/\___/ /_/  /_/_____/\____/ .___/____/
                                                  /_/
juha@juha-ubuntu:~$ 
```

Back in the day, text-based input and output were all you got (after punch cards, that is). Like the very first cars, the first operating systems didn't even have a hood to pop. Everything was in plain sight. In this environment, the so-called REPL (read-eval-print loop) methodology was the natural way to interact with a computer.

REPL means that you type in a command, press enter, and the command is evaluated immediately. It is different from the edit-run-debug or edit-compile-run-debug loops, which you commonly use for more complicated programs.

The command line generally follows the UNIX philosophy of "Make each program do one thing well", so basic commands are very straightforward. The fundamental premise is that you can do complex things by combining these simple programs. The old UNIX neckbeards refer to "having a conversation with the computer."

Why would I use it?

Almost any programming language in the world is more powerful than the command line, and most point-and-click GUIs are simpler to learn. Why would you even bother doing anything on the command line?

The first reason is speed. Everything is at your fingertips. For telling the computer to do simple tasks like downloading a file, renaming a bunch of folders with a specific prefix, or performing a SQL query on a CSV file, you really can't beat the agility of the command line. The learning curve is there, but it is like magic once you have internalized a basic set of commands.

The second reason is agnosticism. Whatever stack, platform, or technology you are currently using, you can interact with it from the command line. It is like the glue between all things. It is also ubiquitous. Wherever there is a computer, there is also a command line somewhere.

The third reason is automation. Unlike in GUI interfaces, everything done in the command line can eventually be automated. There is zero ambiguity between the instructions and the computer. All those repeated clicks in the GUI-based tools that you waste your life on can be automated in a command-line environment.

The fourth reason is extensibility. Unlike GUIs, the command line is very modular. The simple commands are perfect building blocks to create complex functionality for myriads of use-cases, and the ecosystem is still growing after 50 years. The command line is here to stay.

The fifth reason is that there are no other options. It is common that some of the more obscure or bleeding-edge features of a third party service may not be accessible via GUI at all and can only be used using a CLI (Command Line Interface).

How does it work?

There are roughly four layers in how the command-line works:

Terminal = The application that grabs the keyboard input passes it to the program being run (e.g. the shell) and renders the results back. As all modern computers have graphical user interfaces (GUI) these days, the terminal is a necessary GUI frontend layer between you and the rest of the text-based stack.

Shell = A program that parses the keystrokes passed by the terminal application and handles running commands and programs. Its job is basically to find where the programs are, take care of things like variables, and also provide fancy completion with the TAB key. There are different options like Bash, Dash, Zsh, and Fish, to name a few. All with slightly different sets of built-in commands and options.

Command = A computer program interacting with the operating system. Common examples are commands like `ls`, `mkdir`, and `rm`. Some are prebuilt into the shell, some are compiled binary programs on your disk, some are text scripts, and some are aliases pointing to another command, but at the end of the day, they are all just computer programs.

Operating system = The program that executes all other programs. It handles the direct interaction with all the hardware like the CPU, hard disk, and network.

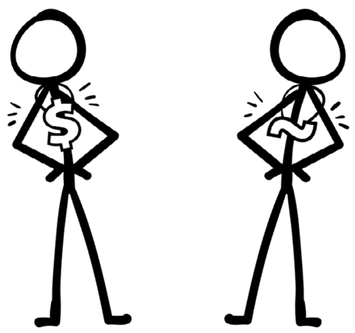
The prompt and the tilde

There is usually one thing common, though: prompt, likely represented by the dollar sign ($). It is a visual cue for where the status ends and where you can start typing in your commands.

On my computer, the command line says:

```
juha@ubuntu:~/hello$
```

The `juha` is my username, ubuntu is my computer name, and `~/hello` is my current working directory.



The command line tends to look slightly different for everyone.

And what's up with that tilde (~) character? What does it even mean that the current directory is `~/hello`?

Tilde is shorthand for the home directory, a place for all your personal files. My home directory is `/home/juha`, so my current working directory is `/home/juha/hello`, which shorthands to `~/hello`. (The convention ~username refers to someone's home directory in general; `~juha` refers to my home directory and so on.)

From now on, we will omit everything else except the dollar sign from the prompt to keep our examples cleaner.

The anatomy of a command

Earlier, we described commands simply as computer programs interacting with the operating system. While correct, let's be more specific.

When you type something after the prompt and press enter, the shell program will attempt to parse and execute it. Let's say:

```
$ generate million dollars
generate: command not found
```

The shell program takes the first complete word `generate` and considers that a command.

The two remaining words, `million` and `dollars`, are interpreted as two separate parameters (sometimes called arguments).

Now the shell program, whose responsibility is to facilitate the execution, goes looking for a `generate` command. Sometimes it is a file on a disk and sometimes something else. We'll discuss this in detail in our next chapter.

In our example, no such command called `generate` is found, and we end up with an error message (this is expected).

Let's run a command that actually works:

```
$ df --human-readable

Filesystem      Size  Used Avail Use% Mounted on
sysfs              0     0     0    - /sys
```

```
proc              0    0     0   - /proc
udev            16G    0   16G   0% /dev
...
```

Here we run a command "`df`" (short for disk free) with the "`--human-readable`" option.

It is common to use "-" (dash) in front of the abbreviated option and "--" (double-dash) for the long-form. (These conventions have evolved over time; see this blog post for more information.)

For example, these are the same thing:

```
$ df -h
$ df --human-readable
```

You can generally also merge multiple abbreviated option after a single dash.

```
df -h -l -a
df -hla
```

Note: The formatting is ultimately up to each command to decide, so don't assume these rules as universal.

Since some characters like space or backslash have a special meaning, it is a good idea to wrap string parameters into quotes. For bash-like shells, there is a difference between single (') and double-quotes ("), though. Single quotes take everything literally, while double quotes allow the shell program to interpret things like variables. For example:

```
$ testvar=13
$ echo "$testvar"
13
$ echo '$testvar'
$testvar
```

If you want to know all the available options, you can usually get a listing with the `--help` parameter:

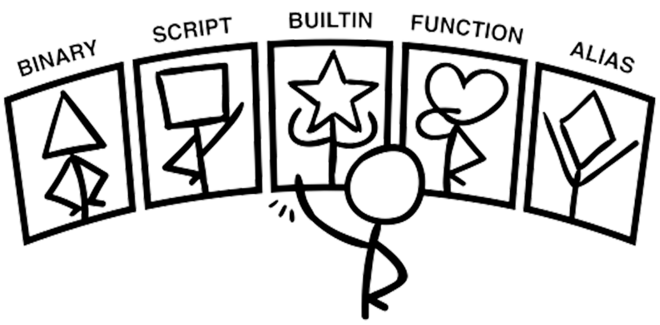
```
df --help
```

Tip: The common thing to type into the command line is a long file path. Most shell programs offer TAB key to auto-complete paths or commands to avoid repetitive typing. Try it out!

The different types of a command

We can split them into two categories, file-based and virtual.

Binary and script commands are file-based and executed by creating a new process (an operating system concept for a new program). File-based commands tend to be more complex and heavyweight.



There are five different types of commands: binary, script, builtin, function, and alias.

Builtins, functions, and aliases are virtual, and they are executed within the existing shell process. These commands are mostly simple and lightweight.

A binary is a classic executable program file. It contains binary instructions only understood by the operating system. You'll get gibberish if you try to open it with a text editor. Binary files are created by compiling source code into the executable binary file. For example, the Python interpreter command `python` is a binary executable.

For binary commands, the shell program is responsible for finding the actual binary file from the file system that matches the command name. Don't expect the shell to go looking everywhere on your machine for a command, though. Instead, the shell relies on an environment variable called `$PATH`, which is a colon-delimited (:) list of paths to iterate over. The first match is always chosen.

To inspect your current `$PATH`, try this:

```
$ echo $PATH
```

If you want to figure out where the binary file for a certain command is, you can call the which command.

```
$ which python
/home/juha/.pyenv/shims/python
```

Now that you know where to find the file, you can use the `file` utility to figure out the general type of the file.

```
$ file /home/juha/.pyenv/shims/pip
/home/juha/.pyenv/shims/pip: Bourne-Again shell script text
executable, ASCII text
$ file /usr/bin/python3.9
/usr/bin/python3.9: ELF 64-bit LSB executable, x86-64, version
1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.
so.2, for GNU/Linux 3.2.0, stripped
```

A script is a text file containing a human-readable program. Python, R, or Bash scripts are some common examples, which you can execute as a command.

Usually we do not execute our Python scripts as commands but use the interpreter like this:

```
$ python hello.py
Hello world
```

Here `python` is the command, and `hello.py` is just a parameter for it. (If you look at what `python --help` says, you can see it corresponds to the variation "file: program read from script file", which really does make sense here.)

But we can also execute hello.py as directly as a command:

```
$ ./hello.py
Hello world
```

For this to work, we need two things. Firstly, the first line of `hello.py` needs to define a script interpreter using a special `#!` Notation.

```
#!/usr/bin/env python3
print("Hello world")
```

The `#!` notation tells the operating system which program knows how to interpret the text in the file and has many cool nicknames like shebang, hashbang, or my absolute favorite the hash-pling!

The second thing we need is for the file to be marked executable. You do that with the `chmod` (change mode) command: `chmod u+x hello.py` will set the eXecutable flag for the owning User.

A builtin is a simple command hard-coded into the shell program itself. Commands like `cd`, `echo`, `alias`, and `pwd` are usually builtins.

If you run the `help` command (which is also a builtin!), you'll get a list of all the builtin commands.

A **function** is like an extra builtin defined by the user. For example:

```
$ hello() { echo 'hello, world'; }
```

Can be used as a command:

```
$ hello
hello, world
```

If you want to list all the functions currently available, you can call (in Bash-like shells):

```
$ declare -F
```

Aliases are like macro. A shorthand or an alternative name for a more complicated command.

For example, you want new command `showerr` to list recent system errors:

```
$ alias showerr="cat /var/log/syslog"
$ showerr
Apr 27 10:49:20 juha-ubuntu gsd-power[2484]: failed to
turn the kbd backlight off: GDBus.Error:org.freedesktop.
UPower.GeneralError: error writing brightness
...
```

Since functions and aliases are not physical files, they do not persist after closing the terminal and are usually defined in the so-called profile file `~/.bash_profile` or the `~/.bashrc` file, which are executed when a new interactive or login shell is started. Some distributions also support a `~/.bash_aliases` file (which is likely invoked from the profile file -- it's scripts all the way down!).

If you want to get a list of all the aliases currently active for your shell, you can just call the `alias` command without any parameters.

Combining commands together

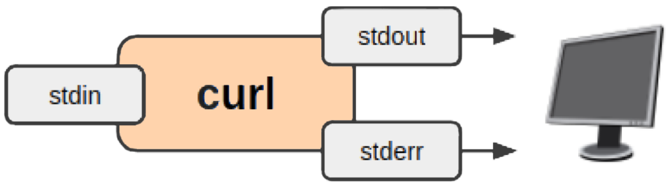
Pretty much anything that happens on your computer happens inside processes. Binary and script commands always start a new process. Builtins, functions, and aliases piggyback on the existing shell program's process.

A process is an operating system concept for running an instance of a command (program). Each process gets an ID, its own reserved memory space, and security privileges to do things on your system. Each process also has a standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) streams.

What are these streams? They are simply arbitrary streams of data. No encoding is specified, which means it can be anything. Text, video, audio, morse-code, whatever the author of the command felt appropriate. Ultimately your computer is just a glorified data transformation machine. Thus it makes sense that every process has an input and output, just like functions do. It also makes sense to separate the output stream from the error stream. If your output stream is a video, then you don't want the bytes of the text-based error messages to get mixed with your video bytes (or, in the 1970s, when the standard error stream was implemented after your phototypesetting was ruined by error messages being typeset instead of being shown on the terminal).

By default, the stdout and stderr streams are piped back into your terminal, but these streams can be redirected to files or piped to become an input of another process. In the command line, this is done by using special redirection operators (`|` , `>` , `<` , `>>`).

Let's start with an example. The `curl` command downloads an URL and directs its standard output back into the terminal as default.

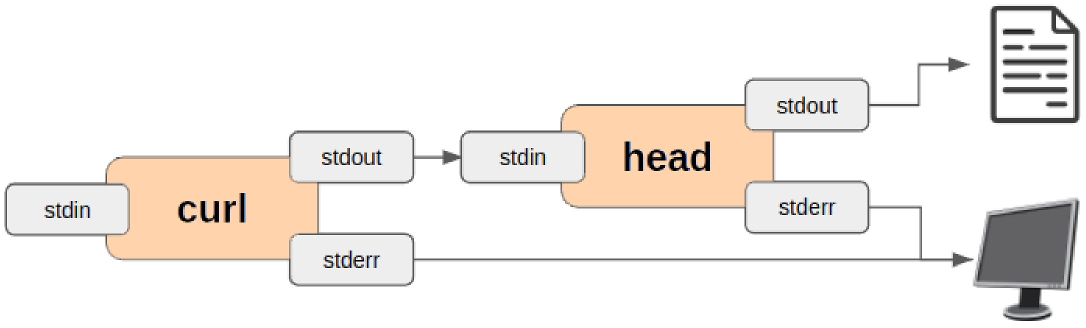


```
$ curl https://filesamples.com/samples/document/csv/sample1.csv
"May", 0.1, 0, 0, 1, 1, 0, 0, 0, 2, 0, 0, 0
"Jun", 0.5, 2, 1, 1, 0, 0, 1, 1, 2, 2, 0, 1
"Jul", 0.7, 5, 1, 1, 2, 0, 1, 3, 0, 2, 2, 1
"Aug", 2.3, 6, 3, 2, 4, 4, 4, 7, 8, 2, 2, 3
"Sep", 3.5, 6, 4, 7, 4, 2, 8, 5, 2, 5, 2, 5
```

Let's say we only want the first three rows. We can do this by piping two commands together using the piping operator (`|`). The standard output of the first command (`curl`) is piped as the standard input of the second (`head`). The standard output of the second command (`head`) remains output to the terminal as a default.

```
$ curl https://filesamples.com/samples/document/csv/sample1.csv
| head -n 3
"May", 0.1, 0, 0, 1, 1, 0, 0, 0, 2, 0, 0, 0
"Jun", 0.5, 2, 1, 1, 0, 0, 1, 1, 2, 2, 0, 1
"Jul", 0.7, 5, 1, 1, 2, 0, 1, 3, 0, 2, 2, 1
```

Usually, you want data on the disk instead of your terminal. We can achieve this by redirecting the standard output of the last command (`head`) into a file called `foo.csv` using the `>` operator.



```
$ curl https://filesamples.com/samples/document/csv/sample1.csv |
head -n 3 > foo.csv
```

Finally, a process always returns a value when it ends. When the return value is zero (0), we interpret it as successful execution. If it returns any other number, it means that the execution had an error and quit prematurely. For example, any Python exception which is not caught by try/except has the Python interpreter exit with a non-zero code.

You can check what the return value of the previously executed command was using the `$?` variable.

```
$ curl http://fake-url
curl: (6) Could not resolve hostmm
$ echo $?
6
```

Previously we piped two commands together with streams, which means they ran in parallel. The return value of a command is important when we combine two commands together using the `&&` operator. This means that we wait for the previous command to succeed before moving on to the next. For example:

```
cp /tmp/apple.png /tmp/usedA.png && cp /tmp/apple.png /tmp/usedB.
png && rm /tmp/apple.png
```

Here we try to copy the file `/tmp/apple` to two different locations and finally delete the original file. Using the `&&` operator means that the shell program checks for the return value of each command and asserts that it is zero (success) before it moves. This protects us from accidentally deleting the file at the end.

If you're interested in writing longer shell scripts, now is a good time to take a small detour to the land of the Bash "strict mode" to save yourself from a lot of headache.

Manage data science projects like a boss

Often when a data scientist ventures out into the command line, it is because they use the CLI (Command Line Interface) tool provided by a third party service or a cloud operator. Common examples include downloading data from the AWS S3, executing some code on a Spark cluster, or building a Docker image for production.

It is not very useful to always manually memorize and type these commands over and over again. It is not only painful but also a bad practice from a teamwork and version control perspective. One should always document the magic recipes.

For this purpose, we recommend using one of the classics, all the way from 1976, the make command. It is a simple, ubiquitous, and robust command which was originally created for compiling source code but can be weaponized for executing and documenting arbitrary scripts.

The default way to use `make` is to create a text file called `Makefile` into the root directory of your project. You should always commit this file into your version control system.

Let's create a very simple `Makefile` with just one "target". They are called targets due to the history with compiling source code, but you should think of target as a task.

Makefile

```
hello:
	echo "Hello world!"
```

Now, remember we said this is a classic from 1976? Well, it's not without its quirks. You have to be **very careful** to indent that `echo` statement with a tab character, not any number of spaces. If you don't do that, you'll get a "missing separator" error.

To execute our "hello" target (or task), we call:

```
$ make hello
echo "Hello world!"
Hello world!
```

Notice how make also prints out the recipes and not just the output. You can limit the output by using the -s parameter.

```
$ make -s hello
Hello world!
```

Next, let's add something useful like downloading our training data.

Makefile

```
hello:
	echo "Hello world!"

get-data:
	mkdir -p .data

	curl <https://filesamples.com/samples/document/csv/sample1.csv>
	> .data/sample1.csv
	echo "Downloaded .data/sample1.csv"
```

Now we can download our example training data with:

```
$ make -s get-data
Downloaded .data/sample1.csv
```

(Aside: The more seasoned Makefile wizards among our readership would note that `get-data` should really be named `.data/sample1.csv` to take advantage of Makefile's shorthands and data dependencies.)

Finally, we'll look at an example of what a simple `Makefile` in a data science project could look like so we can demonstrate how to use variables with make and get you more inspired:

Makefile

```
DOCKER_IMAGE := mycompany/myproject
VERSION := $(shell git describe --always --dirty --long)

default:
    echo "See readme"

init:
    pip install -r requirements.txt
    pip install -r requirements-dev.txt
    cp -u .env.template .env

build-image:
    docker build .
        -f ./Dockerfile
        -t $(DOCKER_IMAGE):$(VERSION)

push-image:
    docker push $(DOCKER_IMAGE):$(VERSION)

pin-dependencies:
    pip install -U pip-tools
    pip-compile requirements.in
    pip-compile requirements-dev.in

upgrade-dependencies:
    pip install -U pip pip-tools
    pip-compile -U requirements.in
    pip-compile -U requirements-dev.in
```

This example `Makefile` would allow your team members to initialize their environment after cloning the repository, pin the dependencies when they introduce new libraries, and deploy a new docker image with a nice version tag.

If you consistently provide a nice `Makefile` along with a well-written readme in your code repositories, it will empower your colleagues to use the command line and reproduce all your per-project magic consistently.

What every data scientist should know about programming tools

There are many ways to give instructions to computers, but writing long text-based recipes is one of the most challenging and versatile ways to command our silicon-based colleagues. We call this approach programming, and most data scientists accept that it is a part of their profession, but unfortunately, many underestimate the importance of tooling for it.

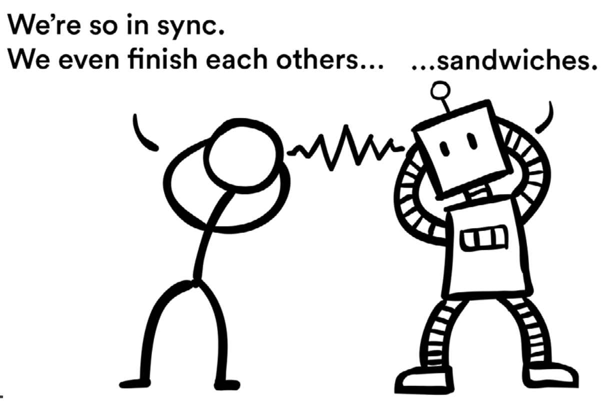
The minimum tooling is a simple text editor and the ability to execute your programs. Most operating systems come with an editor (like Notepad in Windows) and the ability to run code (Mac & Linux ship with a c++ compiler). Programming in this minimalistic way went out of fashion in the 90s.

Notebooks (like Jupyter) are often the first contact with programming for any data scientist. There is absolutely nothing wrong with notebooks, and they are fantastic for many use-cases, but they are not the only option for writing programs. Too many get stuck in the vanilla notebook and do not realize what they are missing out on.

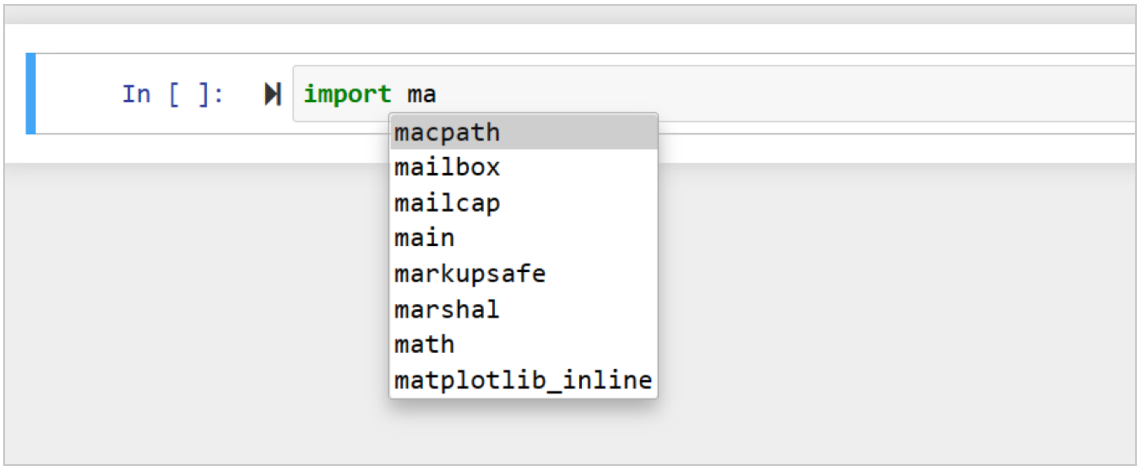
There are many tools for writing, refactoring, navigating, debugging, analyzing, and profiling source code. Most tools are stitched together into a single program called IDE (Integrated Development Environment), but some remain as separate stand-alone programs. Most modern IDEs (like VSCode and PyCharm) also have a vibrant plugin ecosystem to extend the built-in capabilities, and the same can be said about the notebooks too.

Code Completion

The programmer needed to memorize all the syntax and methods by heart back in the day. The simple text editor wouldn't offer any suggestions, and the internet didn't really exist yet. If you were lucky, you had some programming books on your shelves. These days the best IDEs type the code for you. You start a "sentence," and the tooling finishes it for you. This is called code completion.



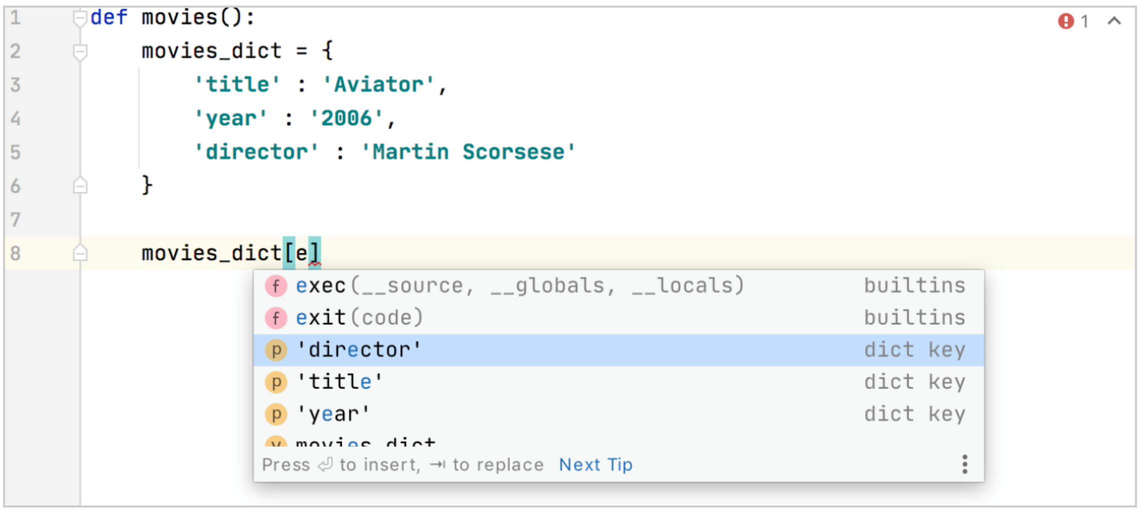
The funny thing is that I always thought code completion is something that only IDEs do and Jupyter doesn't, but it does! Start writing some code in your notebook and press the TAB key. It's magic.



```
In [ ]:  import ma
         macpath
         mailbox
         mailcap
         main
         markupsafe
         marshal
         math
         matplotlib_inline
```

Code completion in Jupyter notebook

Code completion is a bit smoother in IDEs, though. There is no need to keep firing the TAB key, and the popups offer more context like method signatures, documentation and tips.

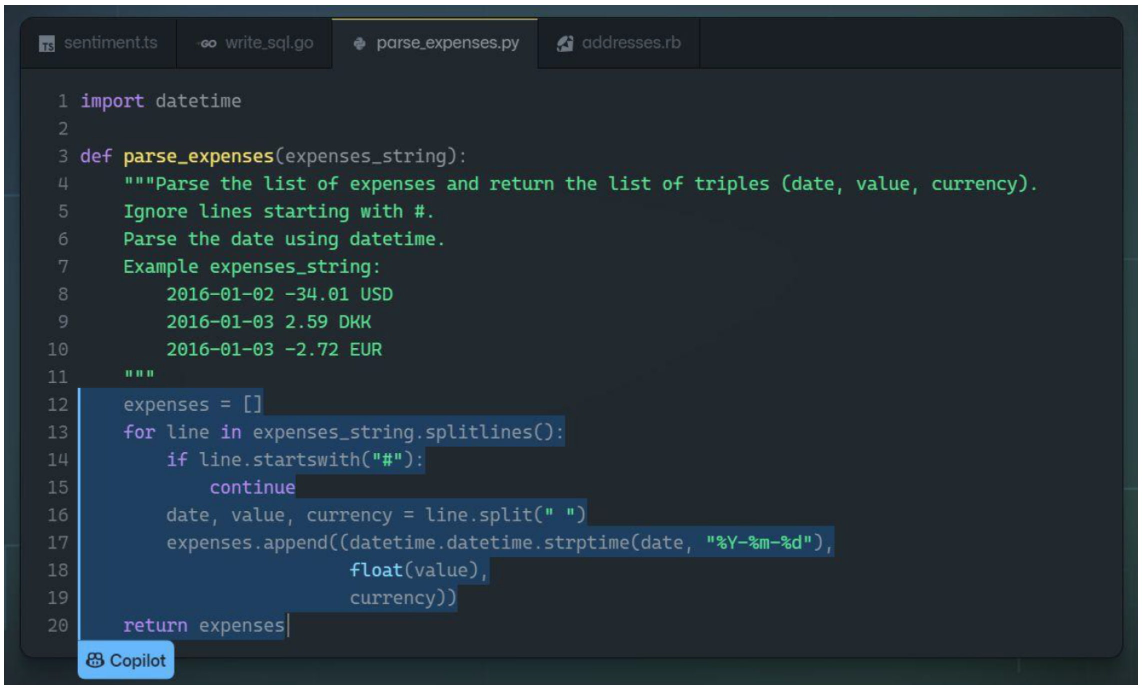


```
def movies():
    movies_dict = {
        'title' : 'Aviator',
        'year' : '2006',
        'director' : 'Martin Scorsese'
    }

    movies_dict[e]
```

```
f exec(__source, __globals, __locals)      builtins
f exit(code)                               builtins
p 'director'                               dict key
p 'title'                                  dict key
p 'year'                                   dict key
v movies_dict
Press ⏎ to insert, ⇥ to replace  Next Tip
```

Code completion in PyCharm

The latest game-changer in code completion is GitHub CoPilot. It is a plugin that doesn't just finish your "sentences" but offers the entire "chapters" based on your typing. Future programmers will use more and more AI-assisted code editors like GitHub CoPilot to write code, just like I'm using AI-assisted natural language tools like Grammarly to write this article. This progression is inevitable.

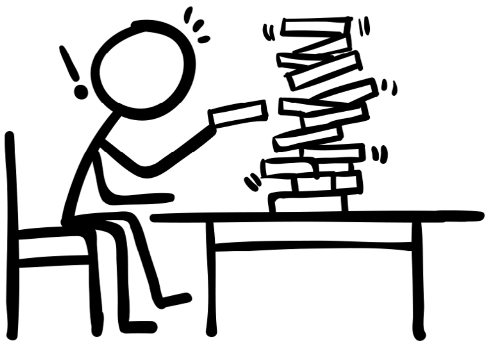


```python
import datetime

def parse_expenses(expenses_string):
    """Parse the list of expenses and return the list of triples (date, value, currency).
    Ignore lines starting with #.
    Parse the date using datetime.
    Example expenses_string:
        2016-01-02 -34.01 USD
        2016-01-03 2.59 DKK
        2016-01-03 -2.72 EUR
    """
    expenses = []
    for line in expenses_string.splitlines():
        if line.startswith("#"):
            continue
        date, value, currency = line.split(" ")
        expenses.append((datetime.datetime.strptime(date, "%Y-%m-%d"),
                         float(value),
                         currency))
    return expenses
```

Code completion with GitHub CoPilot

The bottom line is that if you have never used code completion before, you should start doing that today. It will change your life!

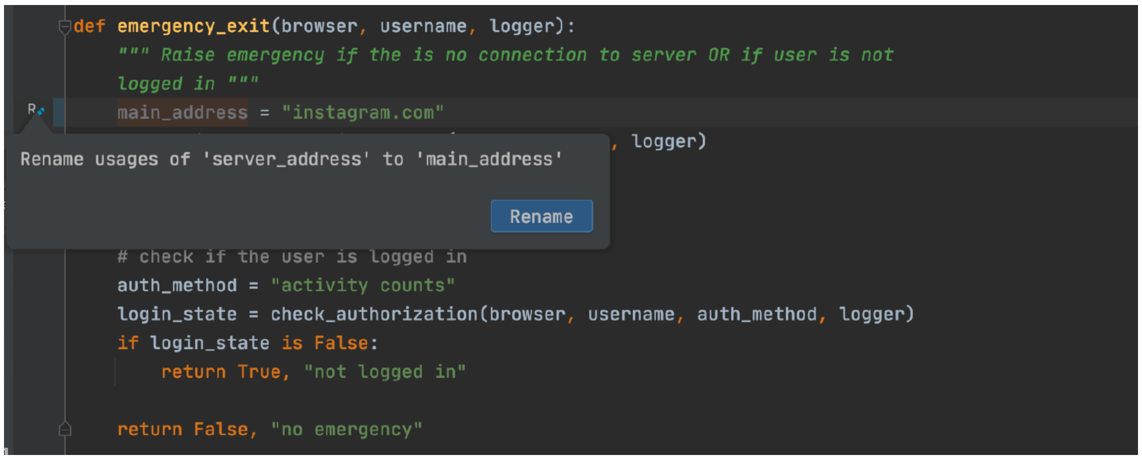
Refactoring



Imagine writing some code and having a variable called `table`. You use the variable all over the place and later realize that you should've named it `customers_table` instead, as the original name is too vague.

In a Jupyter notebook, you could do a “find and replace” operation, but it only covers a single notebook and can be slightly dangerous. For example, your code will break if you have used the word `table` in any other context.

Modern IDE is context-aware and truly understands code. It knows what a method is, and the rename operation isn't just a dummy string operation but safely and robustly renames all usages across the entire codebase.



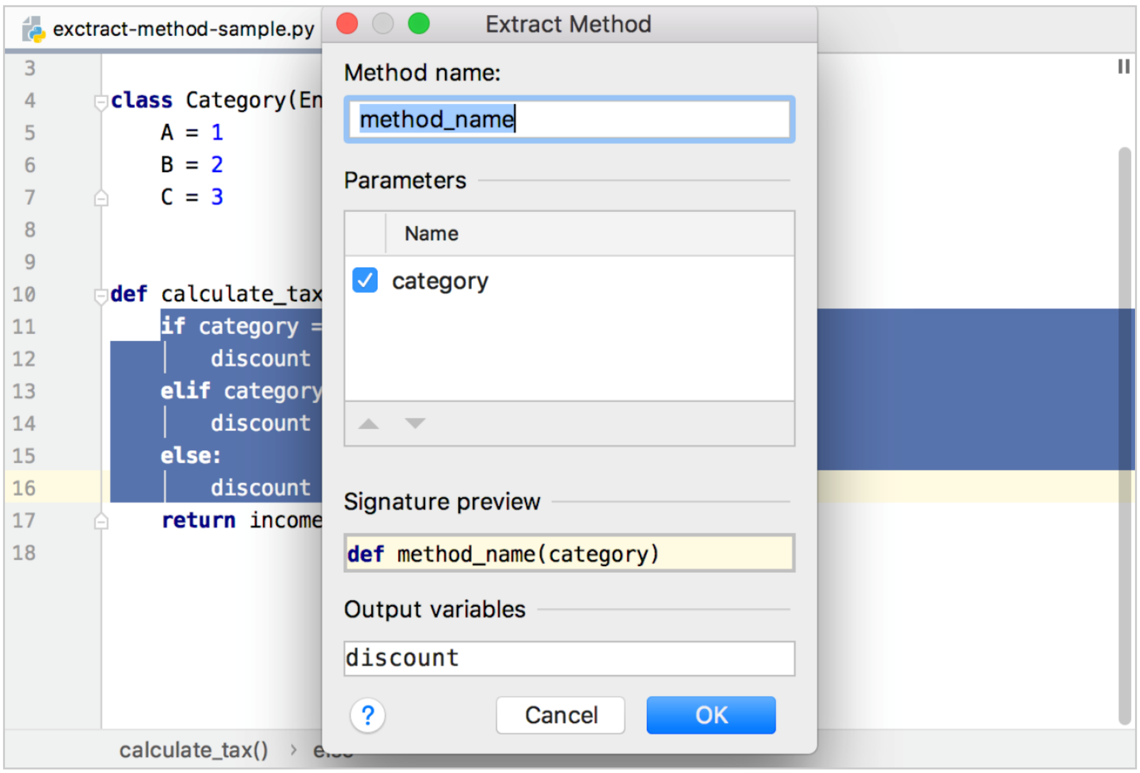
```
def emergency_exit(browser, username, logger):
    """ Raise emergency if the is no connection to server OR if user is not
    logged in """
    main_address = "instagram.com"
                                                  , logger)
```

Rename usages of 'server_address' to 'main_address'

```
    # check if the user is logged in
    auth_method = "activity counts"
    login_state = check_authorization(browser, username, auth_method, logger)
    if login_state is False:
        return True, "not logged in"

    return False, "no emergency"
```

Renaming a variable in PyCharm

Renaming a method or a variable is a classic, but there are dozens of useful little tools out there like adding imports, extracting methods, auto-updating class initializers, and commenting a large chunk of code to name a few.



```
class Category(En
    A = 1
    B = 2
    C = 3


def calculate_tax
    if category =
        discount
    elif category
        discount
    else:
        discount
    return income
```

Method name: method_name

Parameters

	Name
✓	category

Signature preview

```
def method_name(category)
```

Output variables

```
discount
```

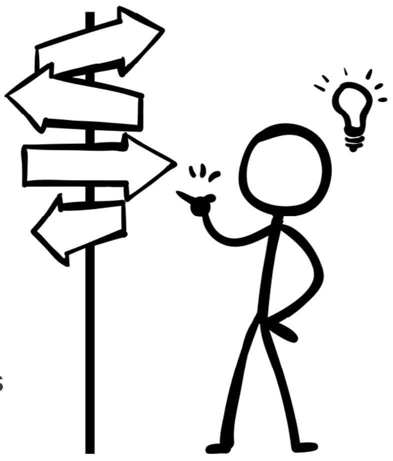
Extracting a method in PyCharm

If you want more inspiration, check out the documentation for PyCharm & VSCode
https://www.jetbrains.com/help/pycharm/refactoring-source-code.html
https://code.visualstudio.com/docs/editor/refactoring

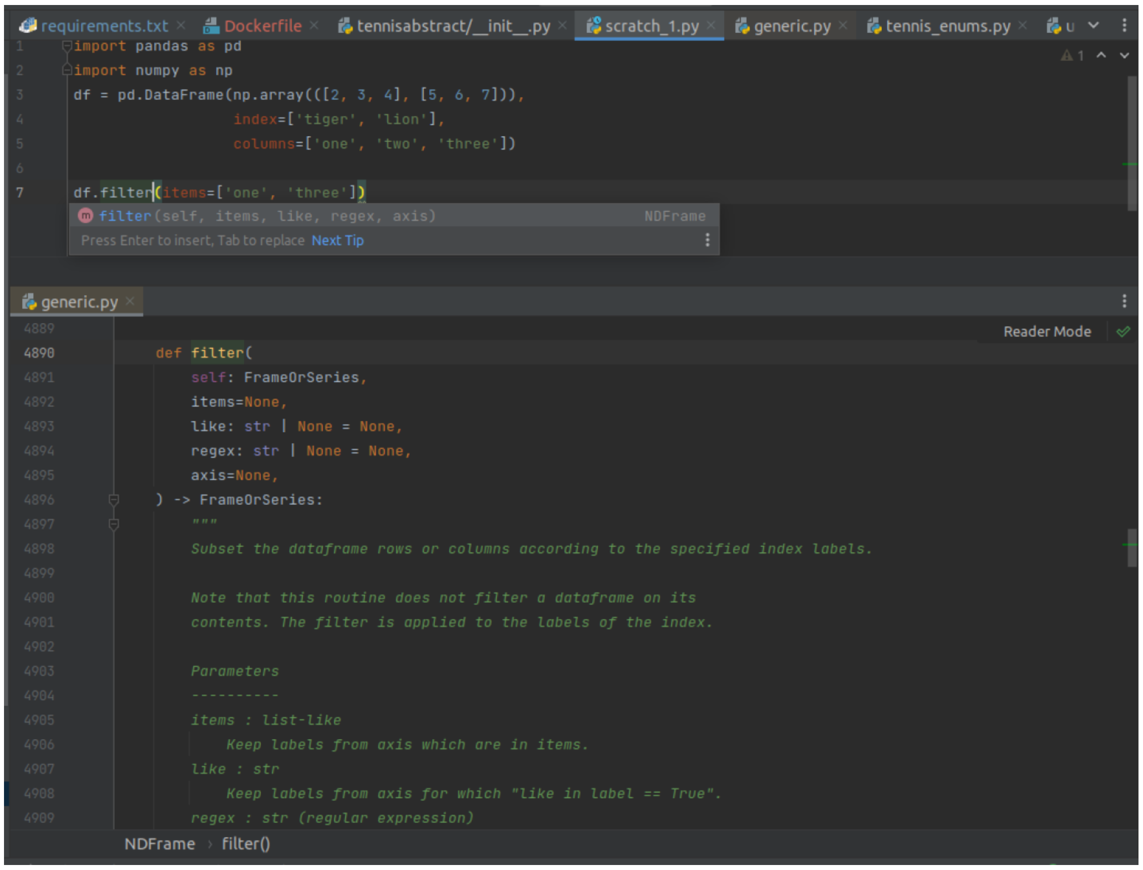
Navigation

Code navigation usually happens when you are figuring things out. You ask questions like, "What does this method do?" and "Where is this variable introduced again?"

One might think that while editing a single notebook with only 50 lines of code, there isn't much to navigate around, but that is a fallacy. You are always using 3rd party libs like pandas or matplotlib, which have 1000x more code than your notebook.



The great thing about using an IDE is that you can dive into the source code of 3rd party package. Want to know what filter() method in Pandas actually does under the hood? Just CTRL+click it and see the implementation yourself! The source code for Pandas is not some next-level voodoo. It is vanilla Python code written by a flesh-and-blood programmer just like you. Don’t be afraid to dive in!



```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.array(([2, 3, 4], [5, 6, 7])),
                  index=['tiger', 'lion'],
                  columns=['one', 'two', 'three'])

df.filter(items=['one', 'three'])
```

```
filter(self, items, like, regex, axis)    NDFrame
Press Enter to insert, Tab to replace  Next Tip
```

```
    def filter(
        self: FrameOrSeries,
        items=None,
        like: str | None = None,
        regex: str | None = None,
        axis=None,
    ) -> FrameOrSeries:
        """
        Subset the dataframe rows or columns according to the specified index labels.

        Note that this routine does not filter a dataframe on its
        contents. The filter is applied to the labels of the index.

        Parameters
        ----------
        items : list-like
            Keep labels from axis which are in items.
        like : str
            Keep labels from axis for which "like in label == True".
        regex : str (regular expression)
```

Diving into the source code of Pandas filter()

Navigation tools are great at putting everything at your fingertips. Almost every IDE has a generic search tool, which is like having a google search engine for your project. “What was the name of that method again?” and “I need to edit the Dockerfile now” are just hotkey away from getting solved.

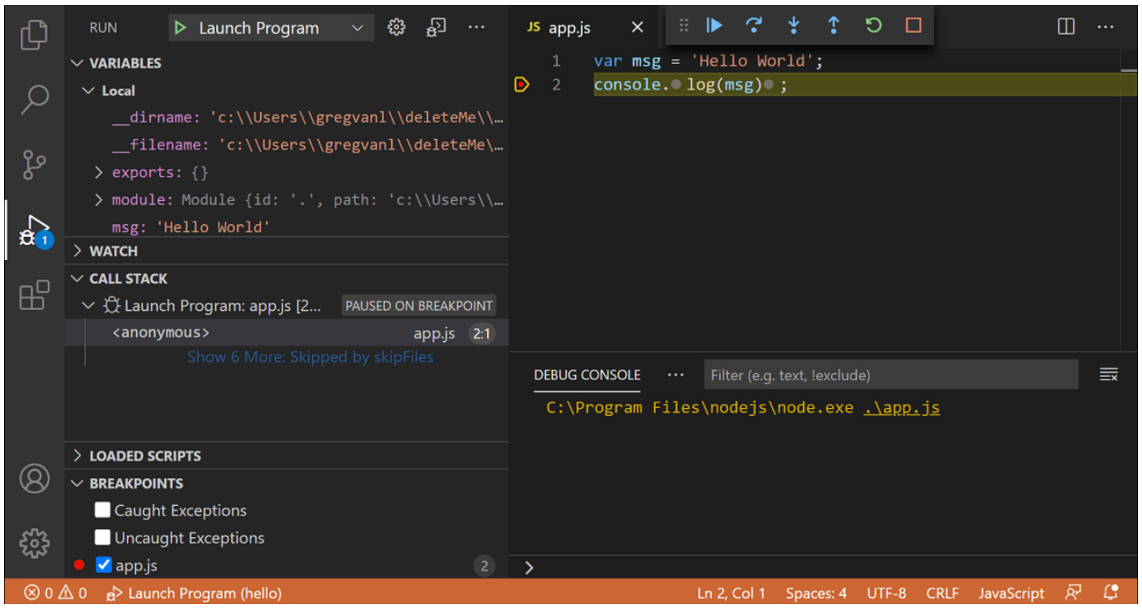
Learning all the hotkeys for navigation feels like a burden at first, but jumping around in code becomes second nature once you have internalized them. Navigation is one aspect where the notebooks are unfortunately quite lacking, perhaps due to being designed for a single piece of code and not a large codebase.

Debugger

Let's face it, every piece of code out there has bugs, and when you are writing something new, your program is broken pretty much all of the time. Debugging is the act of finding out why the darned thing doesn't do what you expect. Someone once said that debugging is like being the detective in a crime movie where you are also the murderer.

The easy and obvious bugs are squashed just by staring at the code. There is nothing wrong with that. If that doesn't work, the following approach is running the program with some extra logging, which is fine too. But once we get into the twilight zone of the more bizarre bugs, where nothing seems to make sense, you want to get yourself a debugger.

A debugger is a tool that lets you run the program and inspect its execution like you had one of those 10000 frames per second stop-motion cameras. You get to run the program step-by-step, see the value of every variable, and follow the execution down to the rabbit hole of method calls as deep as you need to go. You no longer need to guess what happens. The entire state of the program is at your fingertips.



```
var msg = 'Hello World';
console.log(msg);
```

```
__dirname: 'c:\\Users\\gregvanl\\deleteMe\\…
__filename: 'c:\\Users\\gregvanl\\deleteMe\…
> exports: {}
> module: Module {id: '.', path: 'c:\\Users\\…
msg: 'Hello World'
```

```
Launch Program: app.js [2...   PAUSED ON BREAKPOINT
<anonymous>   app.js   2:1
Show 6 More: Skipped by skipFiles
```

```
C:\Program Files\nodejs\node.exe .\app.js
```

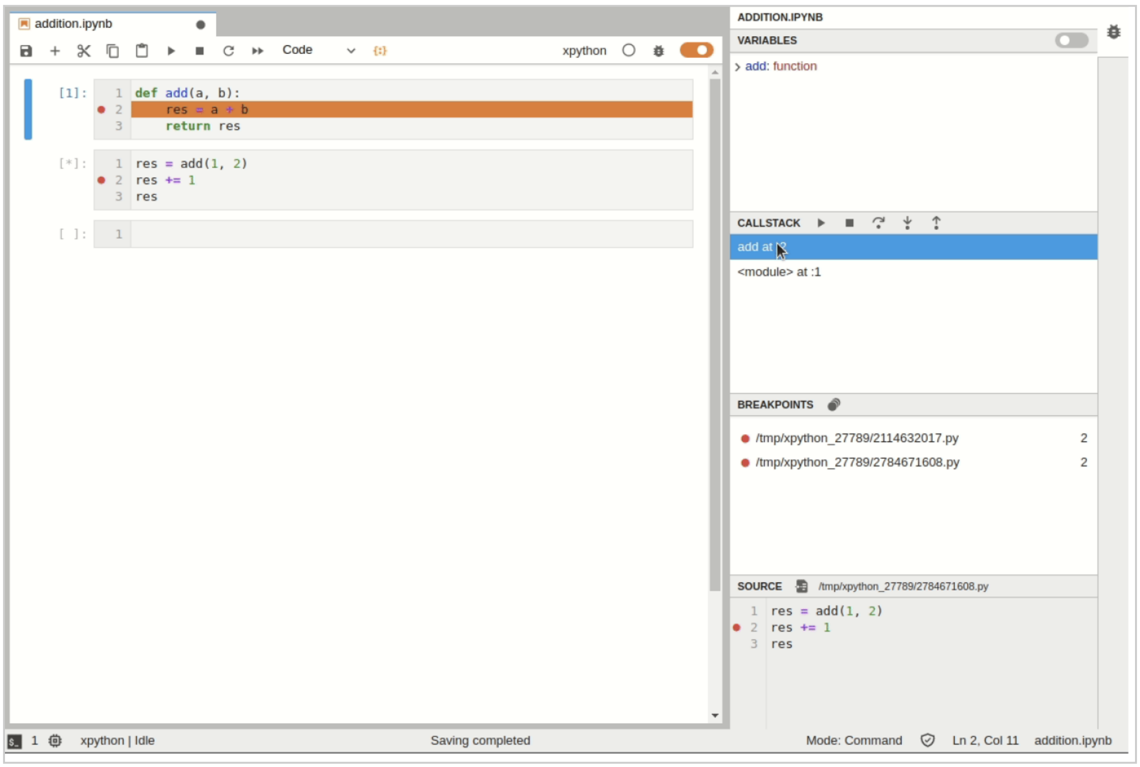
☐ Caught Exceptions
☐ Uncaught Exceptions
☑ app.js 2

⊗ 0 ⚠ 0 Launch Program (hello) Ln 2, Col 1 Spaces: 4 UTF-8 CRLF JavaScript

VSCode debugger inspecting a running program

As data scientists, we often run our production code in the cloud, and the most bizarre bugs tend to thrive in these situations. When your production environment (cloud) slightly deviates from your development environment (laptop), you are in for some painful moments. It is where debuggers shine, as they let you debug remotely and reliably compare the two environments.

Python ships with a built-in command-line debugger `pdb` and Jupyter lets you use it with the `%debug` magics, but we highly recommend using visual debuggers in the IDEs like PyCharm and VSCode. Jupyterlab also has a visual debugger available as an extension (https://github.com/jupyterlab/debugger).


addition.ipynb

```
[1]:
def add(a, b):
    res = a + b
    return res
```

```
[*]:
res = add(1, 2)
res += 1
res
```

```
[ ]:
```

ADDITION.IPYNB

VARIABLES

```
> add: function
```

CALLSTACK

```
add at :2
<module> at :1
```

BREAKPOINTS

```
● /tmp/xpython_27789/2114632017.py    2
● /tmp/xpython_27789/2784671608.py    2
```

SOURCE /tmp/xpython_27789/2784671608.py

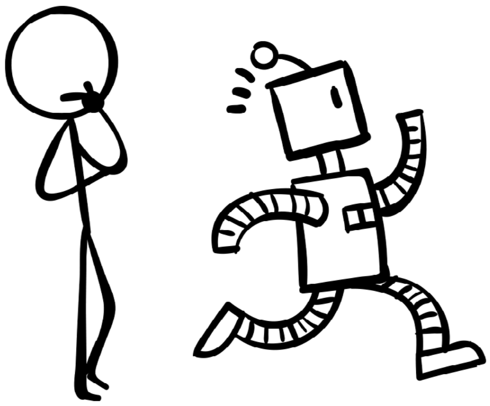
```
res = add(1, 2)
res += 1
res
```

```
xpython | Idle        Saving completed        Mode: Command    Ln 2, Col 11    addition.ipynb
```

JupyterLab debugger extension

A debugger might be an overkill for simple bugs, but the next time you find yourself staring at the code for more than an hour, you might want to consider trying out a debugger. You'd be surprised how much it changes your perspective.

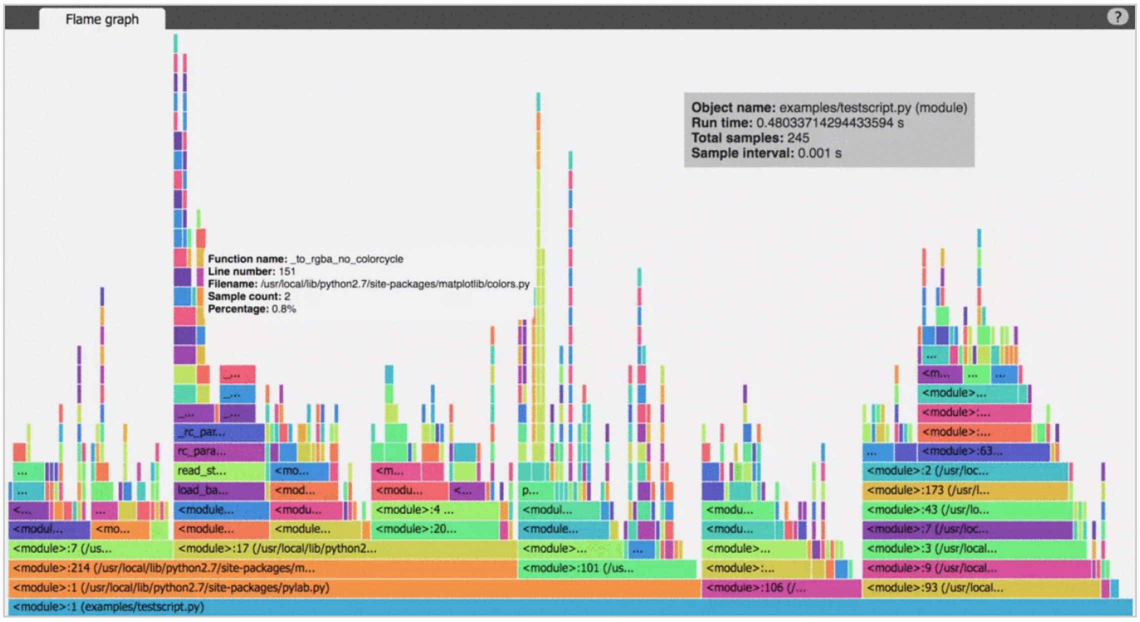
Profiler



The last dish on today's menu is a profiler. Sometimes you face a situation where your code isn't meeting the performance requirements. Perhaps the batch preprocessing step takes five hours, and it needs to happen in 30 minutes, or maybe you can only use two gigabytes of memory, and you are currently hoarding eight.

Often we start guessing blindly where the bottleneck is in our code. We might even manually write some ad-hoc logging to time our method calls. Human intuition can be pretty bad at this. We often end up micro-optimizing things that make no difference at all. It is better than nothing, but to be completely honest, you need a profiler.

A profiler is a tool that times everything and can also measure memory usage in great detail. You run your program using a profiler, and you know exactly where the processing power is spent and who hoarded the precious megabytes. Like the debugger in the previous chapter, you no longer need to guess. All the information is at your fingertips.



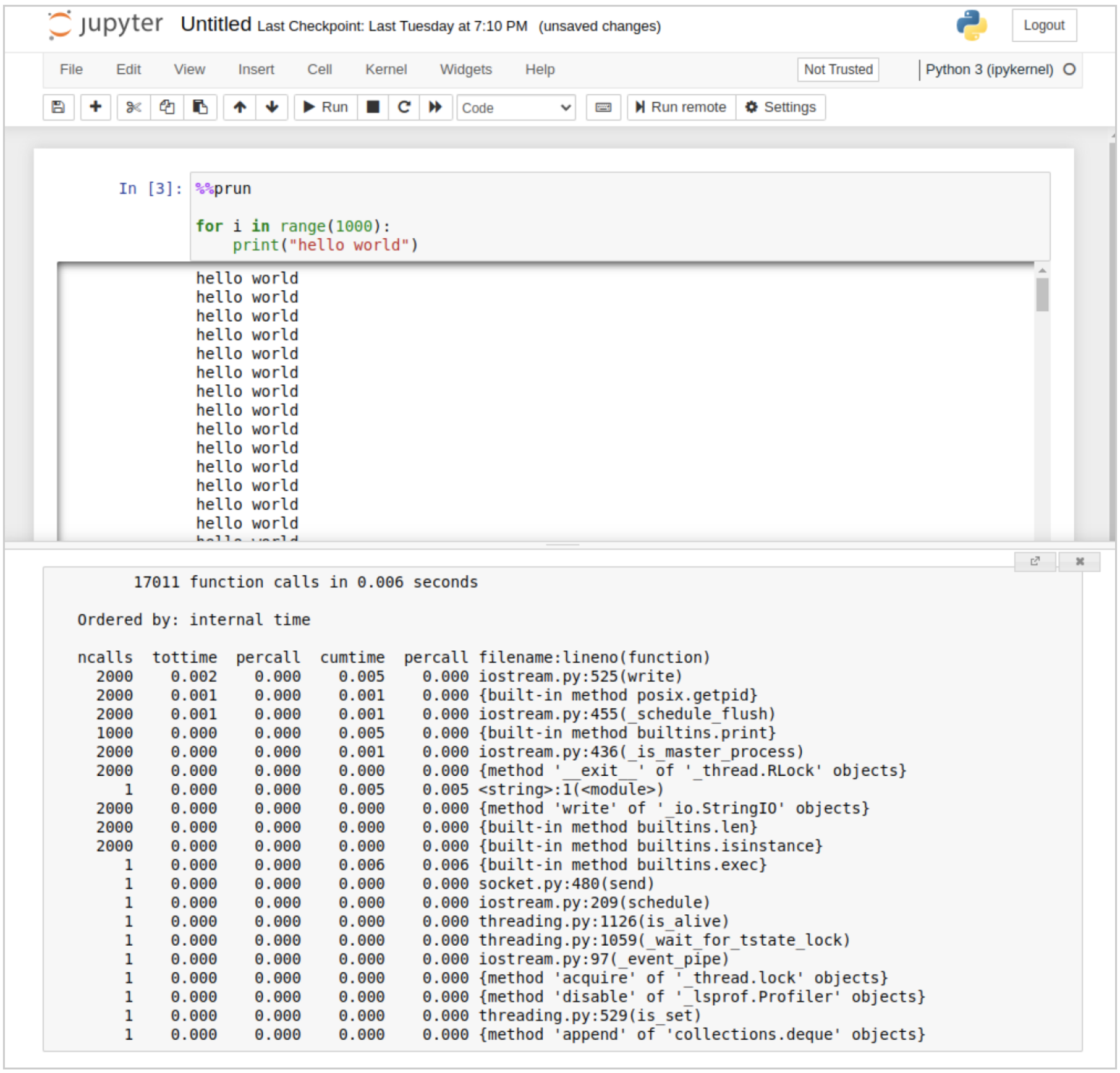
Object name: examples/testscript.py (module)
Run time: 0.48033714294433594 s
Total samples: 245
Sample interval: 0.001 s

Function name: _to_rgba_no_colorcycle
Line number: 151
Filename: /usr/local/lib/python2.7/site-packages/matplotlib/colors.py
Sample count: 2
Percentage: 0.8%

A flame graph visualizing the time spent between different parts of the program

In a typical data science crime scene, the murderer is a 3rd party library like Pandas. It is not that there is anything inherently wrong with it, but they are optimized for ease of use instead of making sure you get the best performance. Complicated things are hidden from you by design. The end result is code that works but is very slow. Profilers are an excellent tool for exposing this when needed. It is not uncommon to get a 100x speed-up by switching one Pandas method to another!

The best profilers are standalone programs or IDE plugins, but all is not lost in the notebook space. Jupyter notebook has built-in magic commands like `%time` and `%prun` which can tell you a lot, but are a bit lacking in the user experience when compared to their visual counterparts.



```
%%prun

for i in range(1000):
    print("hello world")
```

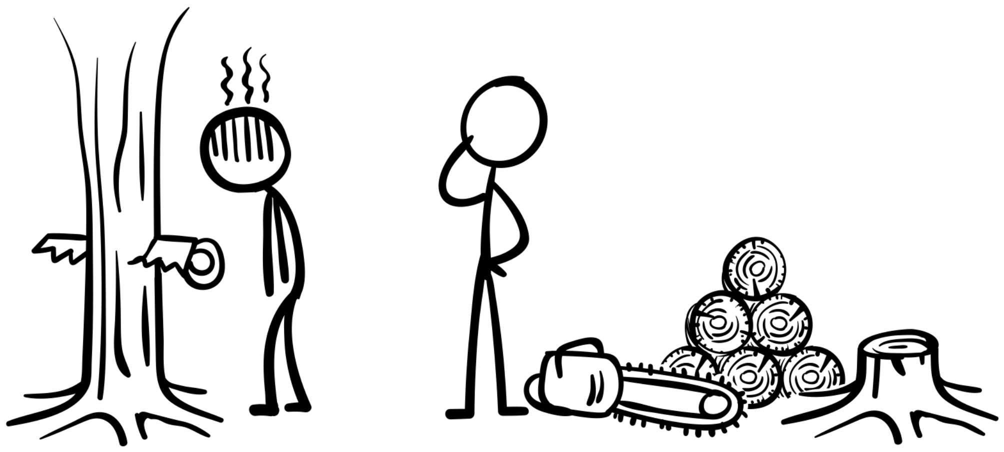
```
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
```

```
         17011 function calls in 0.006 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2000    0.002    0.000    0.005    0.000 iostream.py:525(write)
     2000    0.001    0.000    0.001    0.000 {built-in method posix.getpid}
     2000    0.001    0.000    0.001    0.000 iostream.py:455(_schedule_flush)
     1000    0.000    0.000    0.005    0.000 {built-in method builtins.print}
     2000    0.000    0.000    0.001    0.000 iostream.py:436(_is_master_process)
     2000    0.000    0.000    0.000    0.000 {method '__exit__' of '_thread.RLock' objects}
        1    0.000    0.000    0.005    0.005 <string>:1(<module>)
     2000    0.000    0.000    0.000    0.000 {method 'write' of '_io.StringIO' objects}
     2000    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     2000    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000    0.006    0.006 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 socket.py:480(send)
        1    0.000    0.000    0.000    0.000 iostream.py:209(schedule)
        1    0.000    0.000    0.000    0.000 threading.py:1126(is_alive)
        1    0.000    0.000    0.000    0.000 threading.py:1059(_wait_for_tstate_lock)
        1    0.000    0.000    0.000    0.000 iostream.py:97(_event_pipe)
        1    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 threading.py:529(is_set)
        1    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
```

Profiling a cell in Jupyter notebook with `%%prun`

While debugging can be meaningful without a debugger, optimizing should never be done without a profiler. We are so bad at guessing what makes our programs slow that having a profiler around is the only way to keep us honest while optimizing the performance.

Conclusion



A professional lumberjack doesn't cut down a forest with a rusty old handsaw, he uses a chainsaw because it gets the job done. In this regard, programming is pretty much like any other job. Programming in a vanilla notebook might be fine for small things, but engineering for production without proper tooling is not recommended, and the gap is widening every day in the wake of new AI-assisted programming tools. I hope this article has inspired data scientists to explore what is out there.

Final takeaways

MLOps is the term used for operating a machine learning project in production. It really comes down to running reproducible machine learning workloads with confidence. This ebook teaches the fundamentals of Git, Docker, Python dependencies, and Bash, all requirements for getting your MLOps right to do pioneering machine learning.

There is more to MLOps than these, though. The chapters in this book are just building blocks. It is not enough to build robust Docker images, operate clean git repositories, and master the command line. All those images, repositories, and scripts need to come together. Something needs to glue these things to create a meaningful whole.

The glue could be well-written documentation, a central code repository, a great cloud provider, or an MLOps platform. We actually think it is all of these. As soon as you have more than three models in production or more than five data scientists on payroll, the engineering fundamentals are not enough. You'll need something to keep it all together. The choice is yours.