ML Ops Module

🧑‍🏫 lecture starter for teacher

🚗 Welcome to `WagonCab` 🚗

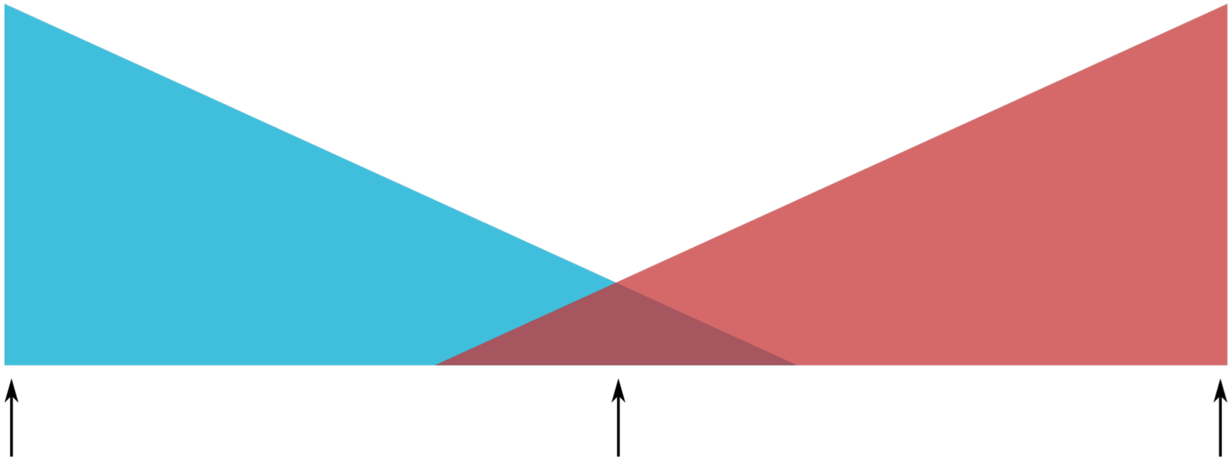
🤝 You've just been hired by `WagonCab` as a **Machine Learning Engineer**

But what is an ML Engineer, exactly?

💡 It's a new role that is emerging as data jobs **specialize**

Data Scientist

Data Engineer



Core Comptencies

Adv. Math/Statistics

ML/AI

Adv. Analytics

Overlapping Skills

Analysis

Programming

Big Data

Core Comptencies

Adv. Programming

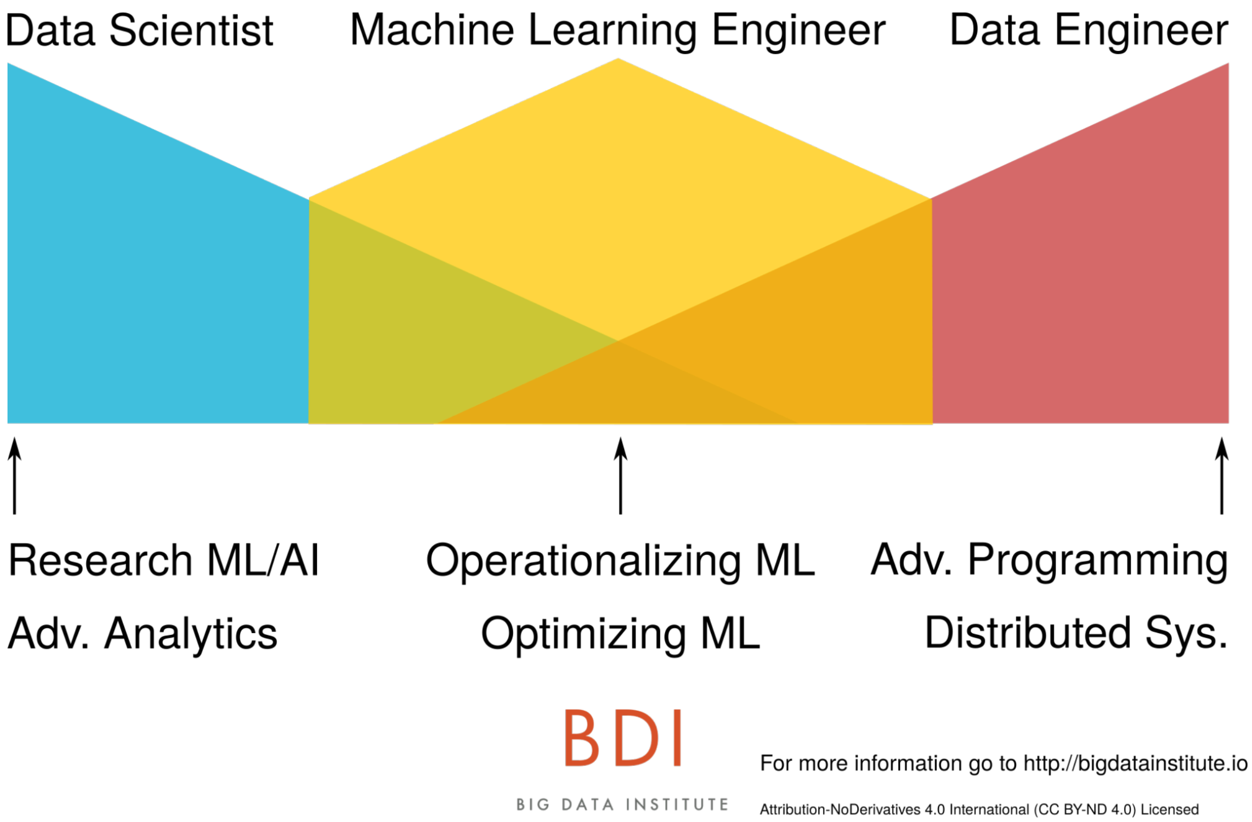
Distributed Sys.

Data Pipelines

BDI

BIG DATA INSTITUTE

For more information go to http://bigdatainstitute.io



For more information go to http://bigdatainstitute.io

💡 It's also emerging as **tools** become more mature

🤯 The full modern ML stack is way too big to cover in this bootcamp (source)

🚗 Back to `WagonCab` 🚗

Your company is launching a new ML product called `TaxiFare`

🎯 Learn to predict the price of conventional taxi rides in New York

🎯 Show in-app 📲 how much users would save using `WagonCab`🚗 instead of Taxis 🚕!

vs
.

WagonCab has the huge, public NYC Trip Record Dataset at its disposal, which is around 170GB in size, and looks as follows

	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	4.5	2009-06-15 17:26:21 UTC	-73.8443	40.7213	-73.8416	40.7123	1
1	16.9	2010-01-05 16:52:16 UTC	-74.016	40.7113	-73.9793	40.782	1
2	5.7	2011-08-18 00:35:00 UTC	-73.9827	40.7613	-73.9912	40.7506	2
3	7.7	2012-04-21 04:30:42 UTC	-73.9871	40.7331	-73.9916	40.7581	1

A team of **Data Scientists** has already created an ML model to predict the price of a ride 🏋️‍♀️

However:

- Work has been done in a isolated context (Notebook)

- The dataset used to train was a smaller, more manageable subset of the NYC dataset (100K rows)

🎯 Your goal as an **ML engineer** will be to:

- train the model at scale 💪
- train the model in the cloud ☁️
- deploy the model in production 🚢
- manage model lifecycle (performance monitoring, re-training on new data, etc.) 🔁
- provide a user interface to access it 🎨

Unit 1) Train at Scale

🎯 Today's goals:

- Understand data scientists' notebooks
- **Package** your Python code (👩‍🏫 lecture)
- Master your **IDE** (👩‍🏫 lecture)
- Learn **incremental** processing techniques to handle GBs worth of data

1) Packaging & Virtual Env 101

1.1) What is a Package?

👉 Re-usable code from one project to another (`from ... import ...`)

A package allows you to:

👉 Share it with others

- Install from PyPI: `pip install <package_name>`
- Install from GitHub: `pip install git+https://...`

👉 Deploy in production (on Linux servers)

👉 Track code (git) and collaborate on it!

🎯 **Lecture's goal**: create a package called `toto` that you will be able to install on any machine
pip install toto

Anatomy of a Minimal Python Package

```
.                       # project directory
├── setup.py            # lists package name and dependencies
└── toto                # package directory
    ├── __init__.py     # defines toto as a package
    └── lib.py          # your code
```

- A module is a single python file inside a package
- A package is directory of python modules that contains an additional `__init__.py`

👉 `__init__.py` allows you to write `toto.lib`, for instance

```
# CLI
python -m toto.lib # executes it as module
python toto/lib.py # executes it as file
```

```
# python file
from toto.lib import a_function
```

👉 `__init__.py`'s content (often empty) is executed at each import line

💻 **LIVECODE: minimal package**

```
mkdir project-toto
cd project-toto
mkdir toto
touch toto/lib.py
touch toto/__init__.py
touch setup.py
code .
```

```
# toto/lib.py
def who_am_i():
    print("Hello my name is Jean")

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

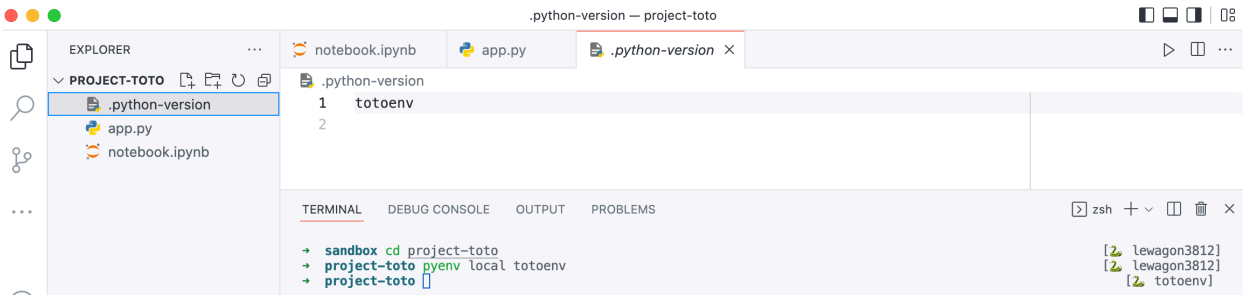
1.2) Virtual Environments

💻 **Let's create a dedicated virtual env `totoenv` for this project**

💡 One `venv` per project is a good practice!

```
# Create new totoenv inside python 3.8.12
pyenv virtualenv 3.8.12 totoenv

# In project-toto, create `.python-version` that activates totoenv when present
pyenv local totoenv
```



```
totoenv
```

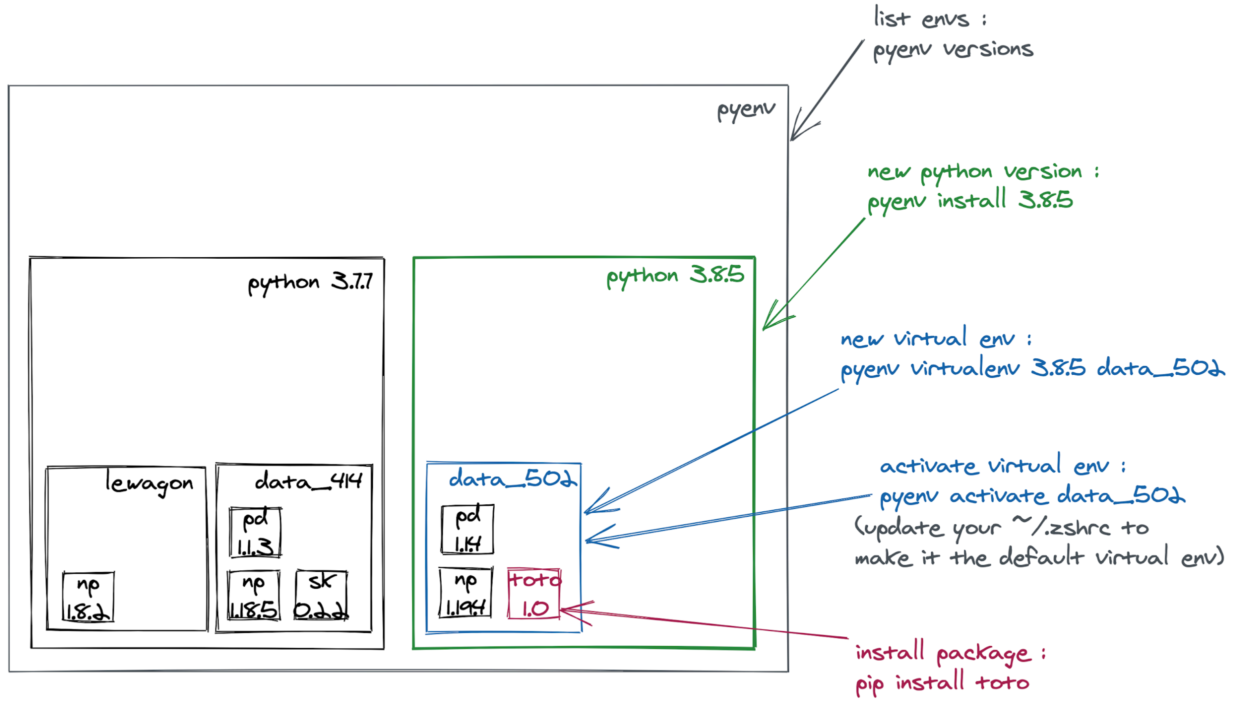
```
➜  sandbox cd project-toto                          [🐍 lewagon3812]
➜  project-toto pyenv local totoenv                 [🐍 lewagon3812]
➜  project-toto                                        [🐍 totoenv]
```

Install minimal packages for this demo lecture

```
pip list
pip install --upgrade pip

pip install pandas
pip install ipython   # needed for ipython
pip install ipykernel # needed for notebooks
```

Reminder on pyenv vs. venv



list envs :
pyenv versions

new python version :
pyenv install 3.8.5

new virtual env :
pyenv virtualenv 3.8.5 data_502

activate virtual env :
pyenv activate data_502
(update your ~/.zshrc to
make it the default virtual env)

install package :
pip install toto

2) 💻 Installing & Using a Package

🎯 Goals

- `pip install` `toto` in virtual env `totoenv`

2.1) Install the Package

Fill in `setup.py`

```
# setup.py
from setuptools import setup

setup(name='toto',
    description="package description",
    packages=["toto"]) # You can have several packages, try it
```

Install

Sit next to `setup.py` and run:

```
pip install .
```

Verify that the package is installed

```
pip freeze
pip freeze | grep toto
pip freeze G toto            # this is an Oh-My-Zsh shortcut
```

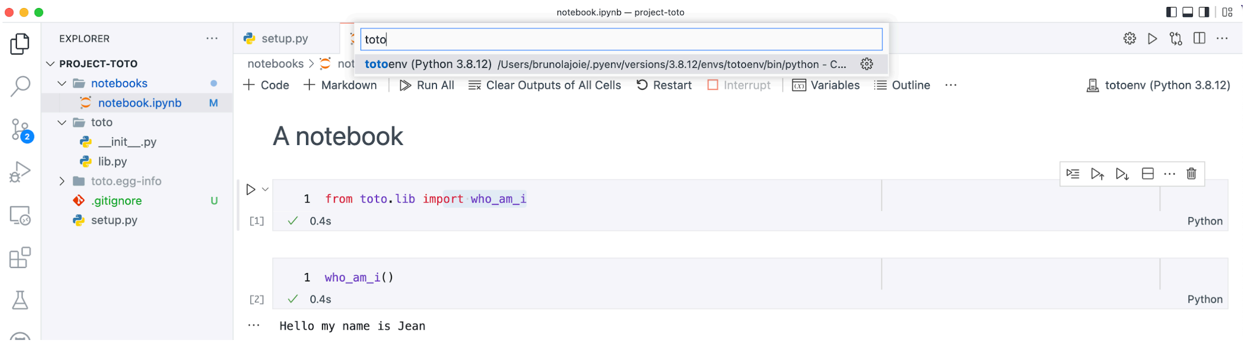
2.2) Run the Package from Anywhere (when `totoenv` is Activated)

For instance, from a notebook

```
mkdir notebooks
touch notebooks/notebook.ipynb
```

Open the notebook with VS Code, select `ipykernel=totoenv`, and you should be able to call

```
from toto.lib import who_am_i
who_am_i()
```



toto

totoenv (Python 3.8.12) /Users/brunolajoie/.pyenv/versions/3.8.12/envs/totoenv/bin/python - C...

```
[1]  from toto.lib import who_am_i
     ✓ 0.4s

[2]  who_am_i()
     ✓ 0.4s

...  Hello my name is Jean
```

❓ **Why Does it Work?**

🔎 Well, remember that Python's `import` always looks at your `PYTHONPATH`

```
import sys
sys.path
```

Which pyenv always appends with site-packages:

👉 *~/.pyenv/versions/3.8.12/envs/**totoenv**/lib/python3.8/**site-packages***

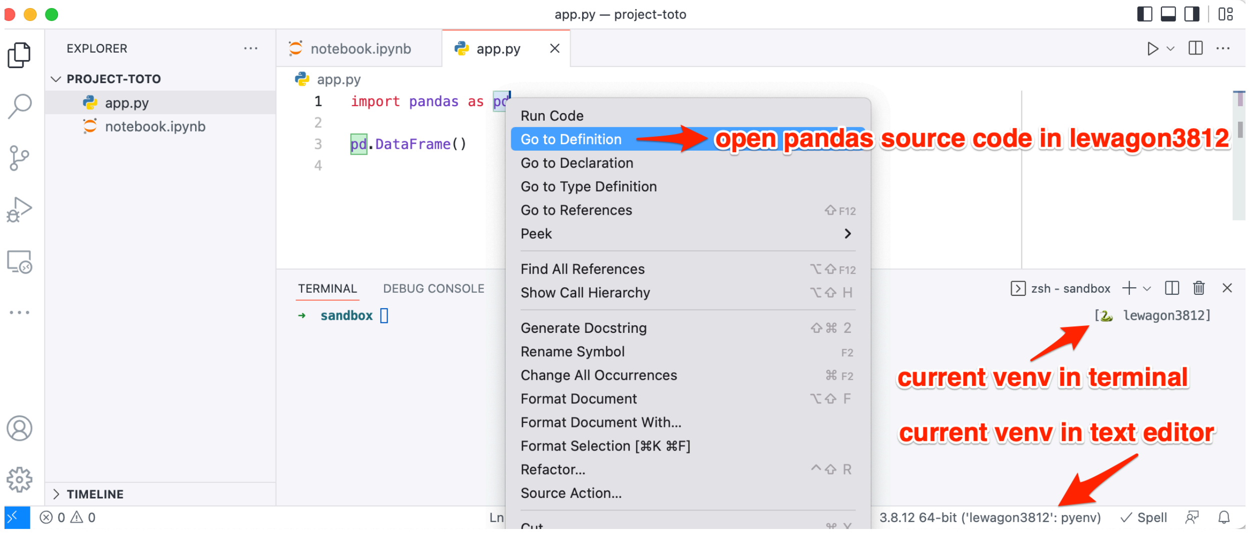
🔎 ...and where is `toto` located?

```
import toto
toto.__file__
```

👉 ~/.pyenv/versions/3.8.12/envs/**totoenv**/lib/python3.8/**site-packages**/toto/__init__.py

✅ `pip install .` creates a **COPY** of your project folder inside site-packages site-packages contains all your pip packages.

🔎 Use it to explore third-party libraries!



```
import pandas as pd

pd.DataFrame()
```

```
→ sandbox
```

```
[lewagon3812]
```

3.8.12 64-bit ('lewagon3812': pyenv)

open pandas source code in lewagon3812

current venv in terminal

current venv in text editor

🔥 Hot-Reload on the Package?

```
def who_am_i():
    print("Hello my name is Jean UPDATED")
```

👉 Is the modification visible?

❌ No, because `pip install .` only creates a **COPY** of your project in site-packages

✅ `pip install -e .` (editable) for hot-reloading

- First, `pip uninstall toto` (move away from `setup.py`'s root to do it properly)
- Then `pip install -e .`

```
import toto
toto.__file__ # '~/code/sandbox/project-toto/toto/__init__.py'
```

PS.: for notebooks and `ipython`, don't forget the magic command to avoid closing/reopening `ipython` every time

```
%load_ext autoreload
%autoreload 2
```

🔥 **Pro Tip**: setup `autoreload` as default action

```
//settings.json
```

```
"jupyter.runStartupCommands": [
  "%load_ext autoreload",
  "%autoreload 2"
],
```

2.3) Dependencies

👉 **Let's say we want the `termcolor` package to be installed along with `toto`**

```
# toto/lib.py
from termcolor import colored

def who_am_i():
    print(colored("Hello my name is Jean", "blue"))
```

✅ **Solution: create `requirements.txt` and update `setup.py`**

```
# Terminal
touch requirements.txt
echo termcolor >> requirements.txt
```

...or specify the version to be used (`termcolor==1.1.0`, for example)

```
# setup.py
from setuptools import setup
from setuptools import find_packages

# list dependencies from file
with open('requirements.txt') as f:
    content = f.readlines()
requirements = [x.strip() for x in content]

setup(name='toto',
      description="package description",
      packages=find_packages(), # NEW: find packages automatically
      install_requires=requirements) # NEW
```

🏁 Then, **pip install -e .** to update `totoenv` with `requirements.txt`

```
.
├── notebooks
|     └── notebook.ipynb     # your code
├── toto                     # package directory
|     ├── __init__.py        # defines toto as a package
|     └── lib.py             # your code
├── requirements.txt         # lists the dependencies
├── setup.py                 # lists package name and dependencies
├── .python-version          # stores name of virtual env
├── .gitignore               # files not to track with git
```

2.4) Adding a `Makefile` to Create Simple CLI Commands ⚡

⌨️ syntax: `make <some_action>`

```
.
├── notebooks
|     └── notebook.ipynb     # your code
├── toto                     # package directory
|     ├── __init__.py        # defines toto as a package
|     └── lib.py             # your code
├── Makefile                 # command line directive manager
├── requirements.txt         # lists the dependencies
├── setup.py                 # lists package name and dependencies
├── .python-version          # stores name of virtual env
├── .gitignore               # files not to track with git
```

Makefile Sample

`Makefile`
directive_name:
<tab>some command with all its arguments
<tab>@this command will not print out before being executed
<tab>-the command after this one will run no matter what
<tab>-@the markers can be combined

A Simple Makefile

`Makefile`

```
install:
	@pip install -e .

clean:
	@rm -f */version.txt
	@rm -f .coverage
	@rm -f */.ipynb_checkpoints
	@rm -Rf build
	@rm -Rf */__pycache__
	@rm -Rf */*.pyc

all: install clean
```

💻 Let's run a directive; while sitting next to the `Makefile`:

```
tree
make clean
tree
```

Let's run several directives; while sitting next to the `Makefile`:

```
make install clean # = make all
```

❗ The `Makefile` is **highly sensitive**:

- Its name is case-sensitive; call it `makefile` and nothing works
- The commands inside of the directives must be indented exclusively using **tabulations**; use one or more spaces instead and nothing works

3) Testing your Package 🧪

Why should we bother with tests?

- ensures robustness of the project in case of changes
- allows teams to work on the same project without breaking each other's code
- with code, you can "describe" what your code should do better than with words 🔇 (TDD)

👉 **Test Driven Development** (TDD) consists of writing the tests **before** the actual code

👉 Most software teams hire full-time testers!

💻 **Test Example**

```python
# toto/divide.py
def divide_without_raising(x:float, y:float) -> float:
    '''
    divides x by y, but instead of raising errors when y equals 0, returns:
    - inf if x positive
    - -inf if x negative
    - nan if x equals 0
    '''
    pass # YOUR CODE HERE
```

💡 Small parentheses regarding `inf` and `nan`

```python
inf = float('inf')
assert type(inf) == float

# We can do arithmetic on infinity!
assert inf == inf
assert inf + inf == inf
assert inf * inf == inf
assert inf * -2 == -inf
```

```python
# BUT these operations on inf are undefined ❌
inf - inf
inf / inf
```

nan

```python
nan = float('nan')

assert type(nan) == float
assert nan != nan # nan is the ONLY FLOAT that does NOT equal itself

import math
assert math.isnan(nan) # check for "nanism"
assert math.isnan(nan + 2) # ANY operation on nan is nan
```

Let's do **TDD**: write tests before coding the function

```python
import math
from toto.divide import divide_without_raising

def test_has_correct_arithmetic():
    assert divide_without_raising(2.0, 2.0) == 1.0, 'wrong basic arithmetic'

def test_handles_divide_by_zero_correctly():
    assert divide_without_raising(2., 0.) == float('inf')
    assert divide_without_raising(-2., 0.) == -1 * float('inf')
    assert math.isnan(divide_without_raising(0., 0.))
```

Launch your tests using the `pytest` framework 🔥

```
echo pytest >> requirements.txt
pip install -e .
pytest tests -v # verbose
```

You'll often see it written in the `Makefile` so you can `make test` your package

```
# Makefile
test:
    @pytest -v tests
```

Write all your tests the same place

```
.
├── notebooks
|     └── notebook.ipynb      # your code
├── tests
|     ├── test_lib.py         # a test file for lib.py
├── toto                      # package directory
|     ├── __init__.py         # defines toto as a package
|     └── lib.py              # your code
├── Makefile                  # command line directive manager
├── requirements.txt          # lists the dependencies
├── setup.py                  # lists package name and dependencies
├── .python-version           # stores name of virtual env
├── .gitignore                # files not to track with git
```

```
# SOLUTION
def divide_without_raising(x:float, y:float) -> float:
    '''
    divides x by y, but instead of raising errors when y equals 0, returns:
    - inf if x positive
    - -inf if x negative
    - nan if x equals 0
    '''
    if y != 0.:
        return x/y
    else:
        if x > 0.:
            return float('inf')
        if x < 0.:
            return -1 * float('inf')
        if x == 0.:
            return float('nan')
```

🏁 Lastly: add a `README.md` to help reproduce your work!

`README.md`

```
# How to install
pip install toto

# How to reproduce results
from toto.lib import who_am_i
who_am_i()

# How to run tests
make tests
```

4) Data Engineering Tips 💡

4.1) Become a Debugging Master!

💻 Live Demo:

- add call to `who_am_i()` inside `divide_without_raising()`
- change color from "blue" to ""

Learn how to read your **stack trace**

```
Traceback (most recent call last):
  File "toto/divide.py", line 22, in <module>
    divide_without_raising(2.,0.)
  File "toto/divide.py", line 10, in divide_without_raising
    who_am_i()
  File "/Users/brunolajoie/code/sandbox/project-toto/toto/lib.py", line 5, in who_am_i
    print(colored("Hello my name is Jean", ""))
  File "/Users/brunolajoie/.pyenv/versions/totoenv/lib/python3.8/site-packages/termcolor.py", line 105, in colored
```

```
    text = fmt_str % (COLORS[color], text)
KeyError: ''
```

🔥 **Pro Tip**: use `Option-Click` to navigate to the line
🔥 **Pro Tip 2**: use `ipdb.set_trace()` (<=> `breakpoint()`) instead of `print()`

`pip install ipdb` (do not add it to `requirements.txt`, it's a *dev-only* package)

ipdb navigation

- `s` (step into)
- `n` (next = step over)
- `c` (continue to next error or `ipdb.set_trace()`)
- `u` (up stack trace)
- `d` (down stack trace)
- `return` (continue until current function's `return`)
- `l` (provide more context)
- `ll` (provide a lot more context)
- `q` (quit; `exit` also works)

🔥 **Pro Tip 3**: automatically set a trace where your code stopped

```
if __name__ == '__main__':
    try:
        divide_without_raising(2., 0.)
    except:
        import ipdb, traceback, sys

        extype, value, tb = sys.exc_info()
        traceback.print_exc()
        ipdb.post_mortem(tb)
```

☝️ Then just use `u` or `d` to get up/down the stack trace until you find your codebase!

4.2) Master your IDE

VS Code Shortcuts (macOS)

- Command palette `⌘-⇧-P`
- Toggle Terminal: `Ctrl-Backtick`
- Split screens with `Ctrl-⌘-⇨`
- Toggle file bar `⌘-B`
- Move panel position (palette)
- Go to file `⌘-P`
- Navigate to symbols globally `⌘-⇧-R`

- Navigate file-to-file `⌥-Click`
- Search `⌘-⇧-F`
- Replace `⌘-⇧-H`
- Rename symbols across all your files (Right-Click)
- Create your own shortcuts (palette)
- Create your own snippets (palette)
- Learn your `settings.json`

Notebooks

- magic commands `# %%` to have Jupyter-like code cells in any Python file
- convert from `.ipynb` to `.py` file
- setup `autoreload` as default action

```
//settings.json
"jupyter.runStartupCommands": [
  "%load_ext autoreload",
  "%autoreload 2"
],
```

4.3) Master your Command Line

❄️**`manual`**

- `man git` full manual for Git CLI

🔥**`--help`**: shorter, and works for sub-commands

- `git --help`
- `git pull --help`

🔥🔥 **tldr** is an even shorter summary

- `brew install tldr`
- `tldr git pull`

Aliases & Commands:

- Customize your aliases in `code ~/.aliases`
- `alias hi='echo hello world'`
- `which hi` lists the location of a command or the command behind an alias

Your Turn! 🚀

You are an **ML Engineer** at `WagonCab` now, working on putting the `TaxiFare` price predictor in production!

Challenge of the Day

- Understand Data Scientists' notebooks
- Package their code into a Python package
- Train it at scale with incremental processing techniques

Appendix

(No live lecture)

A.1) Memory Optimization

For large dataset, it may be useful to compress data by "downcasting" dtypes to **the smallest possible values** according to existing ranges in your dataset

```
s_int = pd.Series([1, 2, 134])
s_int
```

```
0      1
1      2
2    134
dtype: int64
```

pd.Series.astype allows to specify a particular numpy data type

```
s_int = s_int.astype(np.int16)
s_int
```

```
0      1
1      2
2    134
dtype: int16
```

👇 **Beware of the haircut!**

```
s_int = s_int.astype(np.int8)
s_int
```

```
0      1
1      2
2   -122
dtype: int8
```

What happened?

`np.int8` uses 8 bits to represent numbers, of which one bit is used for the sign. So it can only handle integers between -128 and 127.

Let's see what happens once you go above 127.

```
edge_number = pd.Series([127], dtype=np.int8)
edge_number
```

```
0    127
dtype: int8
```

```
edge_number + 1   # This overflows
```

```
0   -128
dtype: int8
```

So we have to make sure that our range fits within the `np.int8` range before downcasting.

```
134 < 2**7   # This number does not fit within the np.int8 range
```

```
False
```

Floats

```
s_float = np.array([0.1234567890123456789, 2, 3], dtype=np.float16)
print("16bit: ", a[0])

s_float = np.array([0.1234567890123456789, 2, 3], dtype=np.float32)
print("32bit: ", b[0])

s_float = np.array([0.1234567890123456789, 2, 3], dtype=np.float64)
print("64bit: ", c[0])
```

👉 Be careful when playing with float precision, especially when **scaling**
☝️ `float32` should be enough for most of your first Data Science projects

☝️ `float16` is extremely uncommon

Downcast made easy with pd.to_numeric 😍

- Converts to the *smallest possible* `int` data type that do not change values
- Also turns `floats64` to `floats32`

```
s_int = pd.Series([1, 2, 134])
s_float = pd.Series([0.1234567890123456789, 2, 3], dtype=np.float64)

print(pd.to_numeric(s_int, downcast='integer'), '\n')
print(pd.to_numeric(s_float, downcast='float'))
```

```
0      1
1      2
2    134
dtype: int16

0    0.123457
1    2.000000
2    3.000000
dtype: float32
```

Assess memory usage

```
df = pd.DataFrame(dict(
    my_int=[123 for _ in range(100)], # int64
    my_float=[1.0 for _ in range(100)], # float64
    my_bool=[True for _ in range(100)])) # bool

df
```

```
df.memory_usage()
```

🎁 Keep that for later use 🎁

```
def compress(df, **kwargs):
    """
    Reduces size of dataframe by downcasting numerical columns
    """
    input_size = df.memory_usage(index=True).sum()/ 1024
    print("new dataframe size: ", round(input_size,2), 'kB')

    in_size = df.memory_usage(index=True).sum()
    for type in ["float", "integer"]:
        l_cols = list(df.select_dtypes(include=type))
        for col in l_cols:
            df[col] = pd.to_numeric(df[col], downcast=type)
    out_size = df.memory_usage(index=True).sum()
    ratio = (1 - round(out_size / in_size, 2)) * 100

    print("optimized size by {} %".format(round(ratio,2)))
    print("new dataframe size: ", round(out_size / 1024,2), " kB")

    return df
```

```
compress(df)
df.memory_usage()
```

```
df.dtypes
```

A.2) Scripts

⌨️ syntax: `do_something`

A **script** is an executable file that can run from **anywhere** on your terminal

💻 **Let's add our script in `scripts/toto-run`**

```
#!/usr/bin/env python
from toto.lib import who_am_i
who_am_i()
```

☝️ The *shebang* `#!` line indicates what interpreter to use

Tell the package to deploy the script in `setup.py`

```
setup(name='toto',
    ...
    scripts=['scripts/toto-run']) # NEW LINE
```

Then `pip install -e .` again

Use script anywhere (if `toto-env` is activated)

```
toto-run
```